

A Beginner's Guide
That Makes You Feel **SMART**

C++ WITHOUT FEAR

THIRD EDITION



- **Learn** programming basics fast
- **Understand** with well-illustrated figures and examples
- **Practice** with games, exercises, and puzzles
- **Write** your first C++ program
- **Refer** to summaries, appendices, and C++14 notes

BRIAN OVERLAND

Updated
for
C++14!

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



C++ Without Fear

Third Edition

This page intentionally left blank

C++ Without Fear

Third Edition

**A Beginner's Guide That
Makes You Feel Smart**

Brian Overland

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

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 author 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com

Library of Congress Cataloging-in-Publication Data

Overland, Brian, 1958– author.

C++ without fear : a beginner's guide that makes you feel smart / Brian Overland.—Third edition.

pages cm

Includes index.

ISBN 978-0-13-431430-3 (pbk. : alk. paper)—ISBN 0-13-431430-1 (pbk. : alk. paper)

1. C++ (Computer program language) I. Title.

QA76.73.C153O838 2016

005.13'3—dc23

2015033385

Copyright © 2016 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. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-431430-3

ISBN-10: 0-13-431430-1

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Once more, for Colin

This page intentionally left blank

Contents

	<i>Preface</i>	xxiii
	We'll Have Fun, Fun, Fun...	xxiii
	Why C and C++?	xxiv
	C++: How to "Think Objects"	xxiv
	Purpose of the Third Edition	xxiv
	Where Do I Begin?	xxv
	Icons and More Icons	xxvi
	Anything Not Covered?	xxvii
	A Final Note: Have Fun!	xxviii
	<i>Acknowledgments</i>	xxix
	<i>About the Author</i>	xxx
Chapter 1	<i>Start Using C++</i>	1
	Install Microsoft Visual Studio	1
	Create a Project with Microsoft	2
	Writing a Program in Microsoft Visual Studio	5
	Running a Program in Visual Studio	5
	Compatibility Issue #1: stdafx.h	6
	Compatibility Issue #2: Pausing the Screen	8
	If You're Not Using Microsoft	8
	Example 1.1. Print a Message	9
	How It Works	9

Exercises	11
<i>Interlude</i> What about the #include and using?	11
Advancing to the Next Print Line	12
Example 1.2. Print Multiple Lines	13
How It Works	14
Exercises	15
<i>Interlude</i> What Is a String?	15
Storing Data: C++ Variables	16
Introduction to Data Types	17
<i>Interlude</i> Why Double Precision, Not Single?	19
Example 1.3. Convert Temperatures	19
How It Works	21
Optimizing the Program	23
Exercises	25
A Word about Variable Names and Keywords	26
Exercise	26
Chapter 1 Summary	27
Chapter 2 <i>Decisions, Decisions</i>	29
But First, a Few Words about Data Types	29
Decision Making in Programs	31
<i>Interlude</i> What about Artificial Intelligence (AI)?	31
if and if-else	32
<i>Interlude</i> Why Two Operators (= and ==)?	35
Example 2.1. Odd or Even?	36
How It Works	37
Optimizing the Code	38
Exercise	39
Introducing Loops	39
<i>Interlude</i> Infinite Loopiness	42
Example 2.2. Print 1 to N	43
How It Works	44
Optimizing the Program	45
Exercises	46
True and False in C++	46
<i>Interlude</i> The bool Data Type	47
The Increment Operator (++)	48
Statements versus Expressions	49

Introducing Boolean (Short-Circuit) Logic	51
<i>Interlude</i> What Is “true”?	53
Example 2.3. Testing a Person’s Age	53
How It Works	54
Exercise	54
Introducing the Math Library	55
Example 2.4. Prime-Number Test	55
How It Works	57
Optimizing the Program	58
Exercise	58
Example 2.5. The Subtraction Game (NIM)	58
How It Works	61
Exercises	61
Chapter 2 Summary	62
Chapter 3 <i>And Even More Decisions!</i>	65
The do-while Loop	65
Example 3.1. Adding Machine	67
How It Works	68
Exercises	69
Introducing Random Numbers	69
Example 3.2. Guess-the-Number Game	72
How It Works	74
Optimizing the Code	76
Exercises	77
The switch-case Statement	77
Example 3.3. Print a Number	80
How It Works	81
Exercises	82
Chapter 3 Summary	83
Chapter 4 <i>The Handy, All-Purpose “for” Statement</i>	85
Loops Used for Counting	85
Introducing the “for” Loop	86
A Wealth of Examples	88
<i>Interlude</i> Does “for” Always Behave Like “while”?	90
Example 4.1. Printing 1 to N with “for”	90

How It Works	91
Exercises	92
Declaring Loop Variables “On the Fly”	92
Example 4.2. Prime-Number Test with “for”	93
How It Works	95
Exercise	96
Comparative Languages 101: The Basic “For” Statement	96
Chapter 4 Summary	97
Chapter 5	
<i>Functions: Many Are Called</i>	99
The Concept of Function	99
The Basics of Using Functions	101
Step 1: Declare (Prototype) the Function	101
Step 2: Define the Function	102
Step 3: Call the Function	102
Example 5.1. The avg() Function	103
How It Works	104
Function, Call a Function!	105
Exercises	106
Example 5.2. Prime-Number Function	106
How It Works	108
Exercises	109
Local and Global Variables	109
<i>Interlude</i> Why Global Variables at All?	111
Recursive Functions	112
Example 5.3. Prime Factorization	113
How It Works	115
<i>Interlude</i> Interlude for Math Junkies	117
Exercises	117
Example 5.4. Euclid’s Algorithm for GCF	118
How It Works	119
<i>Interlude</i> Who Was Euclid?	121
Exercises	121
<i>Interlude</i> Interlude for Math Junkies: Rest of the Proof	122
Example 5.5. Beautiful Recursion: Tower of Hanoi	122
How It Works	125
Exercises	126
Example 5.6. Random-Number Generator	127

	How It Works	128
	Exercises	129
	Games and More Games	129
	Chapter 5 Summary	131
Chapter 6	<i>Arrays: All in a Row...</i>	133
	A First Look at C++ Arrays	133
	Initializing Arrays	135
	Zero-Based Indexing	135
	<i>Interlude</i> Why Use Zero-Based Indexes?	136
	Example 6.1. Print Out Elements	137
	How It Works	137
	Exercises	138
	Example 6.2. How Random Is Random?	139
	How It Works	141
	Exercises	143
	Strings and Arrays of Strings	144
	Example 6.3. Print a Number (from Arrays)	145
	How It Works	147
	Exercises	147
	Example 6.4. Simple Card Dealer	148
	How It Works	150
	Exercises	152
	2-D Arrays: Into the Matrix	152
	Chapter 6 Summary	153
Chapter 7	<i>Pointers: Data by Location</i>	155
	What the Heck Is a Pointer, Anyway?	155
	The Concept of Pointer	156
	<i>Interlude</i> What Do Addresses Look Like?	157
	Declaring and Using Pointers	158
	Example 7.1. Print Out Addresses	161
	Example 7.2. The <code>double_it</code> Function	162
	How It Works	163
	Exercises	164
	Data Flow in Functions	165
	Swap: Another Function Using Pointers	165
	Example 7.3. Array Sorter	166

How It Works	170
Exercises	172
Reference Arguments (&)	172
Pointer Arithmetic	173
Pointers and Array Processing	175
Example 7.4. Zero Out an Array	177
How It Works	178
Writing More Compact Code	178
Exercises	179
Chapter 7 Summary	180
Chapter 8	
<i>Strings: Analyzing the Text</i>	181
Text Storage on the Computer	181
<i>Interlude</i> How Does the Computer Translate Programs?	182
It Don't Mean a Thing if It Ain't Got that String	183
String-Manipulation Functions	184
Example 8.1. Building Strings	186
How It Works	187
Exercises	189
<i>Interlude</i> What about Escape Sequences?	189
Reading String Input	190
Example 8.2. Get a Number	192
How It Works	193
Exercise	195
Example 8.3. Convert to Uppercase	195
How It Works	196
Exercises	197
Individual Characters versus Strings	197
Example 8.4. Breaking Up Input with strtok	198
How It Works	200
Exercises	201
The C++ String Class	201
Include String-Class Support	202
Declare and Initialize Variables of Class string	203
Working with Variables of Class string	203
Input and Output	205
Example 8.5. Building Strings with the string Class	205
How It Works	206

	Exercises	207
	Example 8.6. Adding Machine #2	207
	How It Works	208
	Exercises	209
	Other Operations on the string Type	209
	Chapter 8 Summary	210
Chapter 9	<i>Files: Electronic Storage</i>	213
	Introducing File—Stream Objects	213
	How to Refer to Disk Files	215
	Example 9.1. Write Text to a File	216
	How It Works	217
	Exercises	219
	Example 9.2. Display a Text File	219
	How It Works	221
	Exercises	222
	Text Files versus “Binary” Files	222
	<i>Interlude</i> Are “Binary Files” Really More Binary?	224
	Introducing Binary Operations	225
	Example 9.3. Random-Access Write	227
	How It Works	229
	Exercises	230
	Example 9.4. Random-Access Read	230
	How It Works	232
	Exercises	233
	Chapter 9 Summary	233
Chapter 10	<i>Classes and Objects</i>	237
	OOP, My Code Is Showing	237
	What’s an Object, Anyway?	238
	<i>Interlude</i> OOP...Is It Worth It?	240
	Point: A Simple Class	241
	<i>Interlude</i> Interlude for C Programmers: Structures and Classes	242
	Private: Members Only (Protecting the Data)	243
	Example 10.1. Testing the Point Class	246
	How It Works	247
	Exercises	248

Introducing the Fraction Class	248
Inline Functions	251
Find the Greatest Common Factor	253
Find the Lowest Common Denominator	254
Example 10.2. Fraction Support Functions	255
How It Works	256
Exercises	258
Example 10.3. Testing the Fraction Class	258
How It Works	260
<i>Interlude</i> A New Kind of #include?	261
Exercises	262
Example 10.4. Fraction Arithmetic: add and mult	262
How It Works	266
Exercises	267
Chapter 10 Summary	267
Chapter 11 <i>Constructors: If You Build It...</i>	269
Introducing Constructors	269
Multiple Constructors (Overloading)	270
C++11/C++14 Only: Initializing Members	271
The Default Constructor—and a Warning	272
<i>Interlude</i> Is C++ Out to Trick You with the Default Constructor?	273
C++11/C++14 Only: Delegating Constructors	274
Example 11.1. Point Class Constructors	275
How It Works	277
Exercises	277
Example 11.2. Fraction Class Constructors	278
How It Works	280
Exercises	281
Reference Variables and Arguments (&)	281
The Copy Constructor	282
<i>Interlude</i> The Copy Constructor and References	284
A Constructor from String to Fract	285
Chapter 11 Summary	286

Chapter 12	<i>Two Complete OOP Examples</i>	289
	Dynamic Object Creation	289
	Other Uses of new and delete	290
	Blowin' in the Wind: A Binary Tree App	291
	The Bnode Class	294
	The Btree Class	296
	Example 12.1. Names in Alpha Order	298
	How It Works	299
	Exercises	300
	<i>Interlude</i> Recursion versus Iteration Compared	301
	Tower of Hanoi, Animated	302
	After Mystack Class Design	304
	Using the Cstack Class	304
	Example 12.2. Animated Tower	305
	How It Works	308
	Exercises	311
	Chapter 12 Summary	311
Chapter 13	<i>Easy Programming with STL</i>	313
	Introducing the List Template	313
	<i>Interlude</i> Writing Templates in C++	314
	Creating and Using a List Class	315
	Creating and Using Iterators	316
	C++11/C++14 Only: For Each	318
	<i>Interlude</i> Pointers versus Iterators	319
	Example 13.1. STL Ordered List	319
	How It Works	320
	A Continually Sorted List	321
	Exercises	323
	Designing an RPN Calculator	323
	<i>Interlude</i> A Brief History of Polish Notation	325
	Using a Stack for RPN	325
	Introducing the Generalized STL Stack Class	327
	Example 13.2. Reverse Polish Calculator	329
	How It Works	330
	Exercises	332

	Correct Interpretation of Angle Brackets	333
	Chapter 13 Summary	333
Chapter 14	<i>Object-Oriented Monty Hall</i>	335
	What's the Deal?	335
	TV Programming: "Good Deal, Bad Deal"	337
	Example 14.1. The PrizeManager Class	339
	How It Works	340
	Optimizing the Code	341
	Exercises	342
	Example 14.2. The DoorManager Class	343
	How It Works	344
	Exercises	346
	Example 14.3. The Full Monty Program	347
	How It Works	349
	Exercises	350
	The Monty Hall Paradox, or What's Behind the Door?	351
	Improving the Prize Manager	353
	Chapter 14 Summary	356
Chapter 15	<i>Object-Oriented Poker</i>	359
	Winning in Vegas	359
	How to Draw Cards	361
	The Card Class	363
	The Deck Class	364
	Doing the Job with Algorithms	366
	Example 15.1. Primitive Video Poker	368
	How It Works	369
	Exercises	370
	The Vector Template	371
	Getting Nums from the Player	372
	Example 15.2. Draw Poker	373
	How It Works	376
	Exercises	378
	How to Evaluate Poker Hands	378
	Example 15.3. Draw-Poker Payout!	383

	How It Works	385
	Exercises	386
	Chapter 15 Summary	387
Chapter 16	<i>Polymorphic Poker</i>	389
	Multiple Decks	389
	Switching Decks at Runtime	391
	Polymorphism Is the Answer	392
	Example 16.1. A Virtual Dealer	396
	How It Works	397
	Exercises	399
	<i>Interlude</i> What Is the Virtual Penalty?	399
	“Pure Virtual” and Other Abstract Matters	401
	Abstract Classes and Interfaces	402
	Object Orientation and I/O	403
	cout Is Endlessly Extensible	404
	But cout Is Not Polymorphic	404
	Example 16.2. True Polymorphism: The IPrintable Class	405
	How It Works	408
	Exercises	409
	A Final Word (or Two)	410
	An (Even More) Final Word	411
	Chapter 16 Summary	412
Chapter 17	<i>New Features of C++14</i>	415
	The Newest C++14 Features	415
	Digit-Group Separators	416
	String-Literal Suffix	417
	Binary Literals	418
	Example 17.1. Bitwise Operations	421
	Exercises	421
	Features Introduced in C++11	422
	The long long Type	422
	<i>Interlude</i> Why a “Natural” Integer?	424
	Working with 64-Bit Literals (Constants)	424
	Accepting long long Input	425

Formatting long long Numbers	426
Example 17.2. Fibonacci: A 64-Bit Example	427
How It Works	430
Exercises	431
Localizing Numbers	431
<i>Interlude</i> Who Was Fibonacci?	432
Range-Based “for” (For Each)	433
Example 17.3. Setting an Array with Range-Based “for”	435
How It Works	437
Exercises	437
The auto and decltype Keywords	438
The nullptr Keyword	439
Strongly Typed Enumerations	440
enum Classes in C++11 Onward	442
Extended enum Syntax: Controlling Storage	442
Raw-String Literals	443
Chapter 17 Summary	444
Chapter 18 <i>Operator Functions: Doing It with Class</i>	447
Introducing Operator Functions	447
Operator Functions as Global Functions	450
Improve Efficiency with References	452
Example 18.1. Point Class Operators	454
How It Works	456
Exercises	457
Example 18.2. Fraction Class Operators	457
How It Works	460
Optimizing the Code	461
Exercises	462
Working with Other Types	463
The Class Assignment Function (=)	463
The Test-for-Equality Function (==)	465
A Class “Print” Function	466
Example 18.3. The Completed Fraction Class	467
How It Works	470
Exercises	471

A Really Final Word (about Ops)	471
Chapter 18 Summary	472
Appendix A <i>Operators</i>	475
The Scope (::) Operator	478
The sizeof Operator	478
Old- and New-Style Type Casts	479
Integer versus Floating-Point Division	480
Bitwise Operators (&, , ^, ~, <<, and >>)	480
Conditional Operator	481
Assignment Operators	482
Join (,) Operator	482
Appendix B <i>Data Types</i>	483
Precision of Data Types	484
Data Types of Numeric Literals	485
String Literals and Escape Sequences	486
Two's-Complement Format for Signed Integers	487
Appendix C <i>Syntax Summary</i>	491
Basic Expression Syntax	491
Basic Statement Syntax	492
Control Structures and Branch Statements	493
The if-else Statement	493
The while Statement	493
The do-while Statement	494
The for Statement	494
The switch-case Statement	495
The break Statement	496
The continue Statement	496
The goto Statement	497
The return Statement	497
The throw Statement	497

Variable Declarations	498
Function Declarations	500
Class Declarations	502
Enum Declarations	503
Appendix D <i>Preprocessor Directives</i>	505
The #define Directive	505
The ## Operator (Concatenation)	507
The defined Function	507
The #elif Directive	507
The #endif Directive	508
The #error Directive	508
The #if Directive	508
The #ifdef Directive	509
The #ifndef Directive	510
The #include Directive	510
The #line Directive	511
The #undef Directive	511
Predefined Constants	512
Appendix E <i>ASCII Codes</i>	513
Appendix F <i>Standard Library Functions</i>	517
String (C-String) Functions	517
Data-Conversion Functions	518
Single-Character Functions	519
Math Functions	520
Randomization Functions	521
Time Functions	521
Formats for the strftime Function	523

Appendix G	<i>I/O Stream Objects and Classes</i>	525
	Console Stream Objects	525
	I/O Stream Manipulators	526
	Input Stream Functions	528
	Output Stream Functions	528
	File I/O Functions	529
Appendix H	<i>STL Classes and Objects</i>	531
	The STL String Class	531
	The <bitset> Template	533
	The <list> Template	534
	The <vector> Template	536
	The <stack> Template	538
Appendix I	<i>Glossary of Terms</i>	541
	 <i>Index</i>	 559

This page intentionally left blank

Preface

It's safe to say that C++ is the most important programming language in the world today.

This language is widely used to create commercial applications, ranging from operating systems to word processors. There was a time when big applications had to be written in machine code because there was little room in a computer for anything else. But that time has long passed. Gone are the days in which Bill Gates had to squeeze all of BASICA into 64K!

C++, the successor to the original C language, remains true to the goal of producing efficient programs while maximizing programmer productivity. It typically produces executable files second in compactness only to machine code, but it enables you to get far more done. More often than not, C++ is the language of choice for professionals.

But it sometimes gets a reputation for not being the easiest to learn. That's the reason for this book.

We'll Have Fun, Fun, Fun...

Anything worth learning is worth a certain amount of effort. But that doesn't mean it can't be fun, which brings us to this book.

I've been programming in C since the 1980s and in C++ since the 1990s, and have used them to create business- and systems-level applications. The pitfalls are familiar to me—things like uninitialized pointers and using one equal sign (=) instead of two (==) in an “if” condition. I can steer you past the errors that caused me hours of debugging and sweat, years ago.

But I also love logic problems and games. Learning a programming language doesn't have to be dull. In this book, we'll explore the Tower of Hanoi and the Monty Hall paradox, among other puzzles.

Learning to program is a lot more fun and easy when you can visualize concepts. This book makes heavy use of diagrams and illustrations.

Why C and C++?

There's nothing wrong with other programming languages. I was one of the first people in the world to write a line of code in Visual Basic (while a project lead at Microsoft), and I admire Python as a high-level scripting tool.

But with a little care, you'll find C++ almost as easy to learn. Its syntax is slightly more elaborate than Visual Basic's or Python's, but C++ has long been seen as a clean, flexible, elegant language, which was why its predecessor, C, caught on with so many professionals.

From the beginning, C was designed to provide shortcuts for certain lines of code you'll write over and over; for example, you can use `++n` to add 1 to a variable rather than `n = n + 1`. The more you program in C or C++, the more you'll appreciate its shortcuts, its brevity, and its flexibility.

C++: How to "Think Objects"

A systems programmer named Dennis Ritchie created C as a tool to write operating systems. (He won the Turing Award in 1983.) He needed a language that was concise and flexible, and could manipulate low-level things like physical addresses when needed. The result, C, quickly became popular for other uses as well.

Later, Bjarne Stroustrup created C++, originally as a kind of "C with classes." It added the ability to do object orientation, a subject I'll devote considerable space to, starting in Chapter 10. Object orientation is a way of building a program around intelligent data types. A major goal of this edition is to showcase object orientation as a superior, more modular way to program, and how to "think objects."

Ultimately, C++ became far more than just "C with classes." Over the years, support was added for many new features, notably the Standard Template Library (STL). The STL is not difficult to learn and this book shows you how to use it to simplify a lot of programming work. As time goes on, this library is becoming more central to the work of C++ programmers.

Purpose of the Third Edition

The purpose of the third edition is simple: double down on the strengths of past editions and correct limitations.

In particular, this edition aims at being more fun and easier to use than ever. Most of the features of the previous edition remain, but the focus is more on

the practical (and entertaining) use of C++ and object orientation, and not as much on esoteric features that see little use. For example, I assume you won't want to write your own **string** class, because all up-to-date C++ compilers have provided this feature for a long time now.

In this edition, I also put more stress on “correct” programming practices that have become standard, or nearly so, in the C++ community.

This edition of the book starts out by focusing on successful installation and usage of the Microsoft C++ compiler, Community Edition. If you have another C++ compiler you're happy with, fine. You can use that because the great majority of examples are written in generic C++. The first chapter, however, guides you through the process of using the Microsoft compiler with Visual Studio, if you've never used them before.

Other features of this edition include:

- ▶ **Coverage of new features in C++11 and C++14:** This edition brings you up to date on many of the newest features introduced since C++11, as well as introducing some brand-new features in C++14. It's assumed you have a C++ compiler at least as up to date as the Microsoft Community Edition, so I've purged this edition of the book of some out-of-date programming practices.
- ▶ **Even more puzzles, games, exercises, and figures:** These features, all a successful part of the second edition, show up even more frequently in this edition.
- ▶ **More focus on the “whys” and “how tos” of object orientation:** The class and object features of C++ have always held great promise. A major goal in revising this edition was to put greater emphasis on the practical value of classes and objects, and how to “think objects.”
- ▶ **More on the STL:** The Standard Template Library, far from being difficult to learn, can make your life much easier and make you more productive as a programmer. This edition explores more of the STL.
- ▶ **Useful reference:** This edition maintains the quick-reference appendixes in the back of the book and even expands on them.

Where Do I Begin?

This edition assumes you know little or nothing about programming. If you can turn on a computer and use a menu system, keyboard, and mouse, you can begin with Chapter 1. I'll lead you through the process of installing and using Microsoft C++ Community version.

You should note that this version of C++ runs on recent versions of Microsoft Windows. If you use another system, such as a Macintosh, you'll need to download different tools. But the rules of generic C++ still apply, so you should be able to use most of the book without change.

Icons and More Icons

Building on the helpful icons in the first two editions, this edition provides even more—as signposts on the pages to help you find what you need. Be sure to look for these symbols because they call out sections to which you'll want to pay special attention.



These sections take apart program examples and explain, line by line, how and why the examples work. You don't have to wade through long programming examples—I do that for you! (Or, rather, we go through the examples together.)



After each full programming example, I provide at least one exercise, and usually several, that build on the example in some way. These encourage you to alter and extend the programming code you've just seen. This is the best way to learn. The answers to the exercises can be found on my Web site (brianoverland.com).



These sections develop an example by showing how it can be improved, made shorter, or made more efficient.



As with "Optimizing," these sections take the example in new directions, helping you learn by showing how the example can be varied or modified to do other things.



This icon indicates a place where a keyword of the language is introduced and its usage clearly defined. These places in the text summarize how a given keyword can be used.



The purpose of this icon is similar to "Keyword," but instead it calls attention to a piece of C++ syntax that does not involve a keyword.



"Pseudocode" is a program, or a piece of a program, in English-language form. By reading a pseudocode summary, you understand what a program needs to do. It then remains only to translate English-language statements into C++ statements.

This book also uses “Interludes,” which are side topics that—while highly illuminating and entertaining—aren’t always crucial to the flow of the discussion. They can be read later.

Note ► Finally, some important ideas are sometimes called out with notes; these notes draw your attention to special issues and occasional “gotchas.” For example, one of the most common types of notes deals with version issues, pointing out that some features require a recent compiler:

C++14 ► This note is used to indicate sections that apply only to versions of C++ compliant with the more recent C++ specifications.

Anything Not Covered?

Nothing good in life is free—except maybe love, sunsets, breathing air, and puppies. (Well actually, puppies aren’t free. Not long ago I looked at some Great Dane puppies costing around \$3,000 each. But they were cute.)

To focus more on topics important to the beginner-to-intermediate programmer, this edition has slightly reduced coverage of some of the more esoteric subjects. For example, operator overloading (a feature you might never get around to actually programming into your classes) is still present but moved to the last chapter.

Most other topics—even relatively advanced topics such as bit manipulation—are at least touched upon. But the focus is on fundamentals.

C++ is perhaps the largest programming language on earth, much as English has the largest vocabulary of natural languages. It’s a mistake for an introductory text to try to cover absolutely everything in a language of this size. But once you want to learn more about advanced topics in C++, there are plenty of resources.

Two of the books I’d recommend are Bjarne Stroustrup’s *The C++ Programming Language, Fourth Edition* (Addison-Wesley, 2013), which is by the original author of the C++ language. This is a huge, sophisticated, and exhaustive text, and I recommend it after you’ve learned to be comfortable writing C++ code. As for an easy-to-use reference, I recommend my own *C++ for the Impatient* (Addison-Wesley, 2013), which covers nearly the whole language and almost every part of the Standard Template Library.

Graphical-user-interface (GUI) programming is specific to this or that platform and is deserving of its own—or rather many—books. This book introduces you to the core C++ language, plus its libraries and templates, which are platform independent.

A Final Note: Have Fun!

There's nothing to fear about C++. There are a few potholes here and there, but I'm going to steer you around them. Occasionally, C++ can be a little harder on you if you're not careful or don't know what you're doing, but you'll be better off in the long run by being made to think about these issues.

C++ doesn't have to be intimidating. I hope you use the practical examples and find the puzzles and games entertaining. This is a book about learning and about taking a road to new knowledge, but more than that, it's about enjoying the ride.

Acknowledgments

This edition is largely the result of a conversation between editor Kim Boedigheimer and myself while we had tea in a shop next to Seattle's Pike Place Market. So I think of this book as being as much hers as mine. She brought in an editorial and production team that made life easy for me, including Kesel Wilson, Deborah Thompson, Chris Zahn, Susan Brown Zahn, and John Fuller.

I'm especially indebted to Leor Zolman (yes, that's "Leor"), who provided the single finest technical review I've ever seen. Also providing useful input were John R. Bennett, a software developer emeritus from Microsoft, and online author David Jack ("the logic junkie"), who suggested some useful diagrams.

This page intentionally left blank

About the Author



Brian Overland published his first article in a professional math journal at age 14.

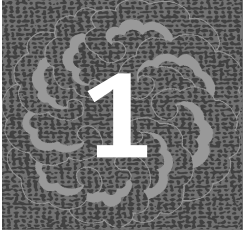
After graduating from Yale, he began working on large commercial projects in C and Basic, including an irrigation-control system used all over the world. He also tutored students in math, computer programming, and writing, as well as lecturing to classes at Microsoft and at the community-college level. On the side, he found an outlet for his life-long love of writing by publishing film and drama reviews in local newspapers. His qualifications as an author of technical books are nearly unique

because they involve so much real programming and teaching experience, as well as writing.

In his 10 years at Microsoft, he was a tester, author, programmer, and manager. As a technical writer, he became an expert on advanced utilities, such as the linker and assembler, and was the “go-to” guy for writing about new technology. His biggest achievement was probably organizing the entire documentation set for Visual Basic 1.0 and having a leading role in teaching the “object-based” way of programming that was so new at the time. He was also a member of the Visual C++ 1.0 team.

Since then, he has been involved with the formation of new start-up companies (sometimes as CEO). He is currently working on a novel.

This page intentionally left blank



Start Using C++

Nothing succeeds like success. This chapter focuses on successfully installing and using the C++ compiler—the tool that translates C++ statements into an executable program (or *application*).

I'm going to assume at first that you're using Microsoft Visual Studio, Community Edition. This includes an excellent C++ compiler—it's powerful, fast, and has nearly all of the up-to-date features. However, the Microsoft compiler raises some special issues, and one of the purposes of this chapter is to acquaint you with those issues so you can successfully use C++.

If you're not using this compiler, skip ahead to the section, "If You're Not Using Microsoft."

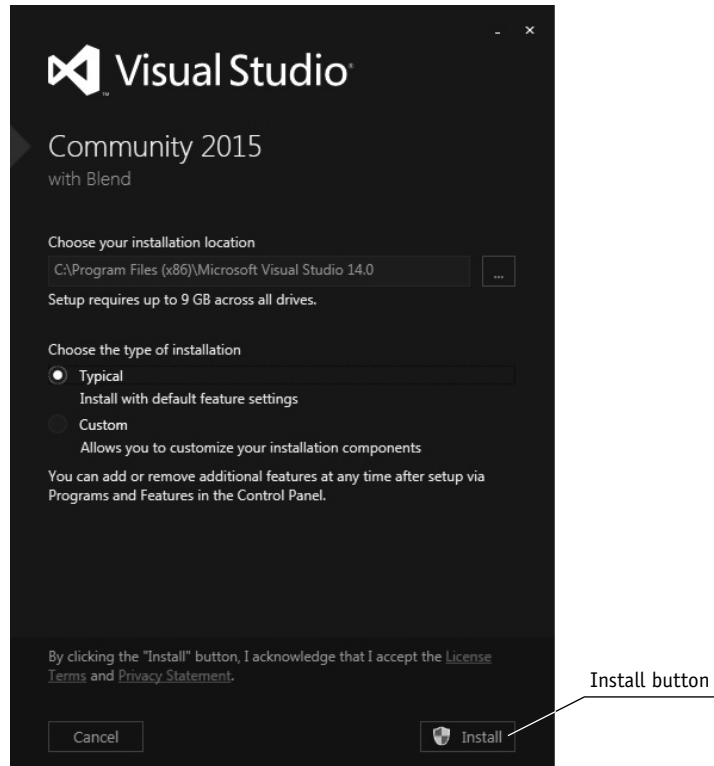
I'll get into the more abstract aspects of C++ later, but first let's get that compiler installed.

Install Microsoft Visual Studio

Even if you have an older version of Microsoft Visual Studio, you should consider updating to the current Community Edition, because it has nearly all the up-to-date features presented in this book. If you're already running Enterprise Edition, congratulations, but make sure it's up to date.

Here are the steps for installing Microsoft Visual Studio Community Edition:

- 1 Regardless of whether you're downloading from the Internet (you can use a search engine to look up "Visual Studio download") or, using the CD accompanying this book's Barnes & Noble Special Edition, get a copy of the file **vc_community** on your computer. If you're downloading, this will be found in your Download folder after using the site.
- 2 Double click the file **vc_community**. This launches the installation program. The following screen appears:



Used with permission from Microsoft.

- 3 Click the Install button in the lower-right corner. Installation should begin right away.
- 4 If you're downloading from the Internet, be prepared for a long wait! If you're using the CD, installation will be many, many times faster.

If all goes well, Microsoft Visual Studio, which includes the Microsoft C++ compiler, should be installed on your computer, and you're ready to start programming. First, however, you need to create a project.

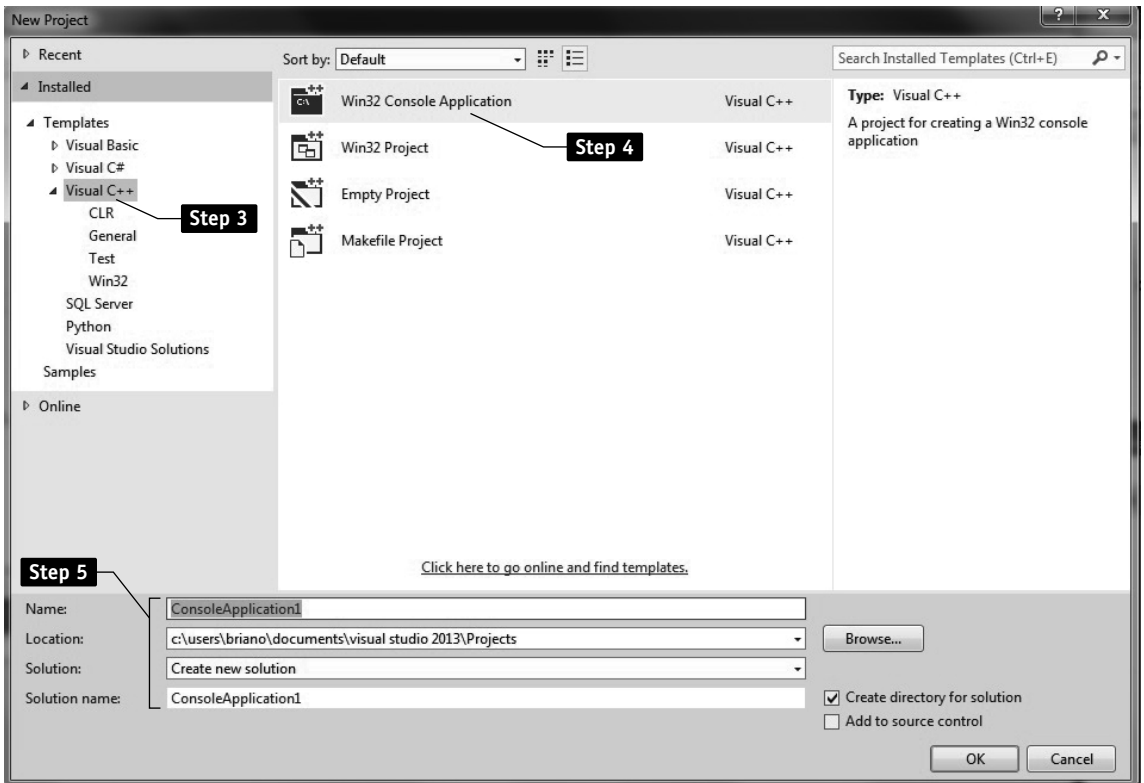
Create a Project with Microsoft

There are some files and settings you need for even the simplest program, but Visual Studio puts all the items you need into something called a *project*.

With Visual Studio, Microsoft makes things easy by providing everything you need when you create a project. Note that you will need to create a new project for each program you work on.

So let's create a project.

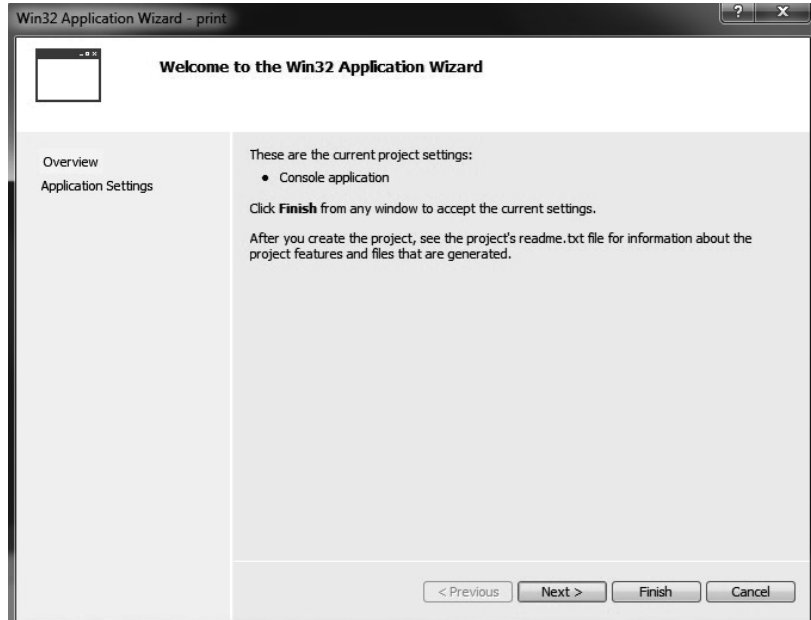
- 1 Launch Visual Studio. After you've installed it, you should find that Visual Studio is available on the Start menu (assuming you're running Windows). Visual Studio should then appear onscreen.
- 2 From the File menu (the first menu on the menu bar), choose the New Project command. The New Project window then appears.



Used with permission from Microsoft.

- 3 In the left pane, select Visual C++.
- 4 In the central windowpane, select Win32 Console Application.
- 5 There are four text boxes at the bottom of the window. You need only fill out one. In the Name box, type the name of the program: in this case, "print1." The Solution name box will automatically display the same text.
- 6 Click OK in the bottom right corner or just press ENTER.

The Application Wizard appears, asking if you're ready to go ahead. (Of course you are.) Click the Finish button at the bottom of the window.



Used with permission from Microsoft

After you complete these steps, a new project is opened for you. The major area on the screen is a text window into which you can enter a program. Visual Studio provides a skeleton (or boilerplate) for a new program containing the following:

```
// print1.cpp: Defines the entry point...
//

#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
    return 0;
}
```

You're probably asking, what is all this stuff? The first thing to be aware of is that any line that begins with double slashes (*//*) is a *comment* and is ignored by the compiler.

Comments exist for the benefit of the programmer, presumably to help a human read and understand the program better, but the C++ compiler completely ignores comments. For now, we're going to ignore them as well.

So the part you care about is just:

```
#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
    return 0;
}
```

Writing a Program in Microsoft Visual Studio

Now—again, assuming you're using Microsoft Visual Studio—you're ready to write your first program. The previous section showed the skeleton (or boilerplate) that's already provided. Your task is to insert some new statements.

In the following example, I've added the new lines and placed them in bold—so you know exactly what to type:

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
    cout << "Never fear, C++ is here!";
    return 0;
}
```

For now, just leave **#include "stdafx.h"** and **t_main** alone, but add new statements where I've indicated. These lines are Microsoft specific, and I'll have more to say about them in the section "Compatibility Issue #1: stdafx.h." First, however, let's just run the program.

Running a Program in Visual Studio

Now you need to translate and run the program. In Visual Studio, all you do is press Ctrl+F5 or else choose the Start Without Debugging command from the Debug menu.

Visual Studio will say that the program is out of date and ask if you want to rebuild it. Say yes by clicking the Yes button.

Note ► You can also build and run the program by pressing F5, but the output of the program will “flash” and not stay on the screen. So use Ctrl+F5 instead.

If you received error messages, you probably have mistyped something. One of the intimidating aspects of C++, until you get used to it, is that even a single mistyped character can result in a series of “cascading errors.” So, don’t get upset, just check your spelling. In particular, check the following:

- The two C++ statements (and most lines of code you type in *will* be C++ statements), end with a semicolon (;), so be careful not to forget those semis.
- But make sure the **#include** directives do *not* end with semicolons(;).
- Case sensitivity absolutely matters in C++ (although spacing, for the most part, does not). Make sure you did not type any capital letters except for text enclosed in quotation marks.

After you’re sure you’ve typed everything correctly, you can rebuild the program by pressing Ctrl+F5 again.

Compatibility Issue #1: stdafx.h

If you’re like me, you’d prefer not to deal with compatibility issues but get right to programming. However, there are a couple of things you need to keep in mind to make sure you succeed with Microsoft Visual Studio.

In order to support something called “precompiled headers,” Microsoft Visual Studio inserts the following line at the beginning of your programs. There’s nothing wrong with this, unless you paste sample code over it and then wonder why nothing works.

```
#include "stdafx.h"
```

The problem is that other compilers will not work with this line of code, but programs built with Microsoft Visual Studio require it, unless you make the changes described in this section.

You can adopt one of several strategies to make sure your programs compile inside Microsoft Visual Studio.

- The easiest thing to do is to make sure this line of code is always the first line in any program created with Visual Studio. So, if you copy generic C++ code listings into a Visual Studio project, make sure you do not erase the directive **#include "stdafx.h"**.
- If you want to compile generic C++ code (nothing Microsoft-specific), then, when creating a project, do not click the Finish button when the Application Wizard window appears. Instead, click Next. Then, in the Application Settings window, click the “Precompiled Headers” button to de-select it.
- After a project is created, you can still change settings by doing the following: First, from the Project menu, choose the Properties command (Alt + F7). Then, in the left pane, select Precompiled Headers. (You may first have to expand “Configuration Properties” and then expand “C/C++” by clicking on these words.) Finally, in the right pane, choose “Not Using Precompiled Headers” from the top drop-down list box.

With the last two options, Microsoft-specific lines such as **#include "stdafx.h"** still appear! However, after the Precompiled Headers option box is de-selected, the Microsoft-specific lines can be replaced with generic C++ code.

Also note that Visual Studio uses the following skeleton for the main function:

```
int _tmain(int arg, _TCHAR* argv[])
{

}
```

instead of:

```
int main()
{

}
```

Both of these work fine with Visual Studio, but if you keep the version that features the word **_tmain**, remember that it requires **#include stdafx.h** as well.

The items inside the parentheses, just after **_tmain**, support access to command-line arguments. But since this book does not address command-line arguments, you won't need them for the examples in this book. Just leave them as they are.

Compatibility Issue #2: Pausing the Screen

As stated earlier, if you build and run the program by pressing Ctrl+F5, your results should be satisfactory, but if you press F5, you'll get the problem of the program output flashing on the screen and disappearing.

If you're using Microsoft Visual Studio, the easiest solution is to just press Ctrl+F5 (Start Without Debugging) every time you build and run the program. However, not all compilers offer this option.

Another way to deal with the problem of output flashing on the screen and disappearing is to add the following line of code, just above "return 0;":

```
system("PAUSE");
```

When this statement is executed, it has roughly the same effect as pressing Ctrl+F5. It causes the program to pause and print "Press any key to continue."

The problem with this statement is that it is system specific. It does what you want in Windows, but it might not work on another platform. Only put this statement in if you're reasonably sure you want your program to run just on Windows-based systems.

If you're working on another platform, you'll need to look for another solution. Check your compiler documentation for more information.

Now, if you're using Microsoft Visual Studio, skip ahead to Exercise 1.1.

If You're Not Using Microsoft

If you're not using Microsoft Visual Studio as your compiler, most of the steps described in the previous sections won't apply. If any documentation comes with your compiler, make sure you read it in case, like Microsoft Visual Studio, it has idiosyncrasies of its own.

With compilers other than Visual Studio, do not put in the line **#include "stdafx.h"** and make sure you use the simpler program skeleton:

```
int main() {  
  
}
```

Beginning with the next section, this book is going to adhere fairly closely to generic C++, which has nothing that is platform or vendor specific. But in this chapter, I'll keep reminding you of what you need to do for Visual Studio.

Example 1.1. *Print a Message*

Here is the program introduced earlier, written in generic C++ (except for the comment, which indicates what you have to do to run it in Visual Studio).

print1.cpp

```
// If you're using Microsoft V.S. leave in this line:  
// #include "stdafx.h"  
  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    cout << "Never fear, C++ is here! ";  
    return 0;  
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

Also remember that if and only if you are working with Microsoft Visual Studio, then, at the beginning of the program, you must leave in the following line:

```
#include "stdafx.h"
```

After entering the program, build and run it (from within Microsoft Visual Studio, press Ctrl+F5). Here's what the program prints when correctly entered and run:

```
Never fear, C++ is here!
```

However, this output may be run together with the message “Press any key to continue.” In the upcoming sections, we're going to correct that.



How It Works

Believe it or not, this simple program has only one real statement. You can think of the rest as “boilerplate” for now—stuff you have to include but can safely ignore. (If you're interested in the details, the upcoming “Interlude” discusses the **#include** directive.)

Except for the one line in italics, the lines below are “boilerplate”: these are items that always have to be present, even if the program doesn’t do anything. For now, don’t worry about why these lines are necessary; their usage will become clearer as you progress with this book. In between the braces ({}), you insert the actual lines of the program—which in this case consist of just one important statement.

```
#include <iostream>
using namespace std;

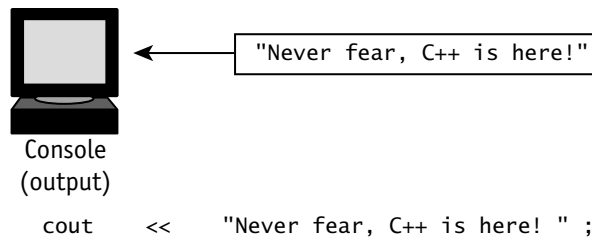
int main()
{
    Enter_your_statements_here!
    return 0;
}
```

This program has one only real statement. Don’t forget the semicolon (;) at the end!

```
cout << "Never fear, C++ is here!";
```

What is **cout**? This is an *object*—that’s a concept I’ll discuss a lot more in the second half of the book. In the meantime, all you have to know is that **cout** stands for “console output.” In other words, it represents the computer screen. When you send something to the screen, it gets printed, just as you’d expect.

In C++, you print output by using **cout** and a leftward stream operator (<<) that shows the flow of data from a value (in this case, the text string “Never fear, C++ is here!”) to the console. You can visualize it this way:



Don’t forget the semicolon (;). Every C++ statement must end with a semicolon, with few exceptions.

For technical reasons, **cout** must always appear on the left side of the line of code whenever it’s used. Data in this case flows to the left. Use the leftward “arrows,” which are actually a pair of less-than signs (<<).

The following table shows other simple uses of **cout**:

STATEMENT	ACTION
<code>cout << "Do you C++?";</code>	Prints the words "Do you C++?"
<code>cout << "I think,";</code>	Prints the words "I think,"
<code>cout << "Therefore I program.";</code>	Prints the words "Therefore I program."



EXERCISES

Exercise 1.1.1. Write a program that prints the message "Get with the program!" If you want, you can work on the same source file used for the featured example and alter it as needed. (Hint: Alter only the text inside the quotation marks; otherwise, reuse all the same programming code.)

Exercise 1.1.2. Write a program that prints your own name.

Exercise 1.1.3. Write a program that prints "Do you C++?"

Interlude

What about the `#include` and `using`?

I said that the fifth line of the program is the first "real" statement of the program. I glossed over the first line:

```
#include <iostream>
```

This is an example of a C++ *preprocessor directive*, a general instruction to the C++ compiler. A directive of the form

```
#include <filename>
```

loads declarations and definitions that support part of the C++ standard library. Without this directive, you couldn't use **cout**.

If you've used older versions of C++ and C, you may wonder why no specific file (such as an `.h` file) is named. The filename `iostream` is a *virtual include file*, which has information in a precompiled form.

If you're new to C++, just remember you have to use **#include** to turn on support for specific parts of the C++ standard library. Later, when we start using math functions such as **sqrt** (square root), you'll need to switch on support for the math library:

```
#include <cmath>
```

Interlude▼ *continued*

Is this extra work? A little, yes. Include files originated because of a distinction between the C language and the standard runtime library. (Professional C/C++ programmers sometimes avoid the standard library and use their own.) Library functions and objects—although they are indispensable to beginners—are treated just like user-defined functions, which means (as you’ll learn in Chapter 4) that they have to be declared. That’s what include files do.

You also need to put in a **using** statement. This enables you to refer directly to objects such as **std::cout**. Without this statement, you’d have to print messages this way:

```
std::cout << "Never fear, C++ is here!";
```

We’re going to be using **cout** (and its cousin, **cin**) quite a lot, so for now it’s easier just to put a **using** statement at the beginning of every program.

Advancing to the Next Print Line

With C++, text sent to the screen does not automatically advance to the next physical line. You have to print a *newline* character to do that. (Exception: If you never print a newline, the text may automatically “wrap” when the current physical line fills up, but this produces an ugly result.)

The easiest way to print a newline is to use the predefined constant **endl**. For example:

```
cout << "Never fear, C++ is here!" << endl;
```

Note ▶ The **endl** name is short for “end line”; it is therefore spelled “end ELL,” not “end ONE.” Also note that **endl** is actually **std::endl**, but the **using** statement saves you from having to type **std::**.

Another way to print a newline is to insert the characters `\n`. This is an escape sequence, which C++ interprets as having a special meaning rather than interpreting it literally. The following statement has the same effect as the previous example:

```
cout << "Never fear, C++ is here!\n";
```

Example 1.2.

Print Multiple Lines

The program in this section prints messages across several lines. If you're following along and entering the programs, remember once again to use uppercase and lowercase letters exactly as shown—although you can change the capitalization of the text inside quotation marks and the program will still run.

If you're working with Visual Studio, the only lines you should add are the ones shown here in bold. Leave **#include stdafx.h** and **_tmain** alone. If you're working with another compiler, the code should look as follows, minus the comments (*//*).

print2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    cout << "I am Blaxxon," << endl;
    cout << "the godlike computer." << endl;
    cout << "Fear me!" << endl;
    return 0;
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

The resulting program, if you're working with Visual Studio, should be as follows. The lines in bold are what you need to add to the code Visual Studio provides for you.

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
    cout << "I am Blaxxon," << endl;
    cout << "the godlike computer." << endl;
}
```

```

    cout << "Fear me!" << endl;
    return 0;
}

```

After entering the program, compile and run it. Here's what the program prints when correctly entered and run:

```

I am Blaxxon,
the godlike computer.
Fear me!

```



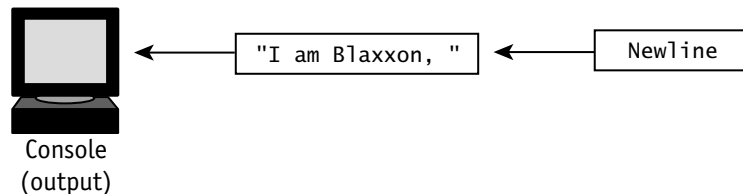
How It Works

This example is similar to the first one I introduced. The main difference is this example uses newline characters. If these characters were omitted, the program would print

```
I am Blaxxon, the godlike computer.Fear me!
```

which is not what we wanted.

Conceptually, here's how the statements in the program work:



```
cout << "I am Blaxxon, " << endl;
```

You can print any number of separate items this way, though again, they won't advance to the next physical line without a newline character (**endl**). You could send several items to the console with one statement

```
cout << "This is a " << "nice " << "C++ program.";
```

which prints the following when run:

```
This is a nice C++ program.
```

Or, you can embed a newline, like this

```
cout << "This is a" << endl << "C++ program.";
```

which prints the following:

```
This is a
C++ program.
```

The example, like the previous one, returns a value. “Returning a value” is the process of sending back a signal—in this case to the operating system or development environment.

You return a value by using the **return** statement:

```
return 0;
```

The return value of **main** is a code sent to the operating system, in which 0 indicates success. The examples in this book return 0, but they could return an error code sometimes (–1 for example) if you found that to be useful. However, I would ignore that for now.



EXERCISES

Exercise 1.2.1. Remove the newlines from the example in this section, but put in extra spaces so that none of the words are crammed together. (Hint: Remember that C++ doesn’t automatically insert a space between output strings.) The resulting output should look like this:

```
I am Blaxxon, the godlike computer. Fear me!
```

Exercise 1.2.2. Alter the example so that it prints a blank line between each two lines of output—in other words, make the results double-spaced rather than single-spaced. (Hint: Print *two* newline characters after each text string.)

Exercise 1.2.3. Alter the example so that it prints two blank lines between each of the lines of output.



What Is a String?

From the beginning, I’ve made use of text inside of quotes, as in this statement:

```
cout << "Never fear, C++ is here!";
```

Everything outside of the quotes is part of C++ syntax. What’s inside the quotes is data.

▼ *continued on next page*

Interlude

▼ *continued*

In actuality, all the data stored on a computer is numeric, but depending on how data is used, it can be interpreted as a string of printable characters. That's the case here.

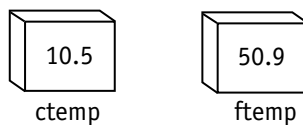
You may have heard of “ASCII code.” That's what kind of data “Never fear, C++ is here!” is in this example. The characters “N”, “e”, “\”, “e”, “r”, and so on, are stored in individual bytes, each of which is a numeric code corresponding to a printable character.

I'll talk a lot more about this kind of data in Chapter 8. The important thing to keep in mind is that text enclosed in quotes is considered raw data, as opposed to a command. This kind of data is considered a string of text or, more commonly, just a *string*.

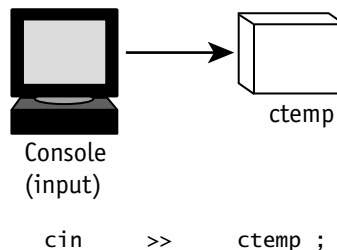
Storing Data: C++ Variables

If all you could do was print messages, C++ wouldn't be useful. The fundamental purpose of nearly any computer program is usually to get data from somewhere—such as end-user input—and then do something interesting with it.

Such operations require *variables*. These are locations into which you can place data. You can think of variables as magic boxes that hold values. As the program proceeds, it can read, write, or alter these values as needed. The upcoming example uses variables named `ctemp` and `ftemp` to hold Celsius and Fahrenheit values, respectively.



How are values put into variables? One way is through console input. In C++, you can input values by using the `cin` object, representing (appropriately enough) console input. With `cin`, you use a stream operator showing data flowing to the right (`>>`):



Here's what happens in response to this statement. (The actual process is a little more complicated, but don't worry about that for now.)

- 1 The program suspends running and waits for the user to enter a number.
- 2 The user types a number and presses ENTER.
- 3 The number is accepted and placed in the variable `ctemp` (in this case).
- 4 The program resumes running.

So, if you think about it, a lot happens in response to this statement:

```
cin >> ctemp;
```

But before you can use a variable in C++, you must declare it. This is an absolute rule and it makes C++ different from Basic, which is sloppy in this regard and doesn't require declaration (but generations of Basic programmers have banged their heads against their terminals as they discovered errors cropping up as a result of Basic's laxness about variables).

This is important enough to justify restating, so I'll make it a cardinal rule:



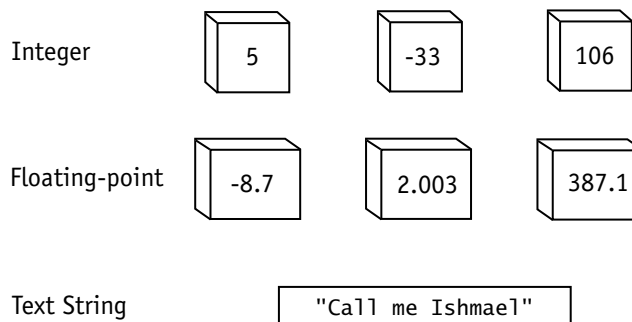
In C++, you must declare a variable before using it.

To declare a variable, you first have to know what *data type* to use. This is a critical concept in C++ as in most other languages.

Introduction to Data Types

A variable is something you can think of as a magic box into which you can place information—or, rather, *data*. But what kind of data?

All data on a computer is ultimately numeric, but it is organized into one of three basic formats: integer, floating-point, and text string.

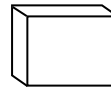


There are several differences between floating-point and integer format. But the main rule is simple:



If you need to store numbers with fractional portions, use a floating-point variable; otherwise, use integer types.

The principal floating-point data type in C++ is **double**. This may seem like a strange name, but it stands for “double-precision floating point.” There is also a single-precision type (**float**), but its use is relatively infrequent. When you need the ability to retain fractional portions, you’ll get better results—and fewer error messages—if you stick to **double**.



aFloat

A **double** declaration has the following syntax. Note that this statement is terminated with a semicolon (;), just as most kinds of statements are.

```
double variable_name;
```

You can also use a **double** declaration to create a series of variables:

```
double variable_name1, variable_name2, ...;
```

For example, this statement creates a **double** variable named aFloat:

```
double aFloat;
```

This statement creates a variable of type **double**.

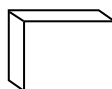
The next statement declares four **double** variables named b, c, d, and amount:

```
double b, c, d, amount;
```

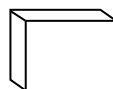
The effect of this statement is equivalent to the following:

```
double b;
double c;
double d;
double amount;
```

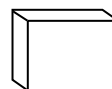
The result of these declarations is to create four variables of type **double**.



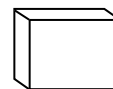
b



c



d



amount



An important rule of good programming style is that variables should usually be initialized, which means giving them a value as soon as you declare them. The declarations just shown should really be:

```
double b = 0.0;
double c = 0.0;
double d = 0.0;
double amount = 0.0;
```

Starting in the next chapter, I'll have a lot more to say about issues such as data types and initialization. But for the next program, I'll keep the code simple. We'll worry about initialization in Chapter 2 onward.

Interlude

Why Double Precision, Not Single?

Double precision is like single precision, except better. Double precision supports a greater range of values, with better accuracy: It uses 8 bytes rather than 4.

C++ converts all data to double precision when doing calculations, which makes sense given that today's PCs include 8-byte co-processors. C++ also stores floating-point constants in double precision unless you specify otherwise (for example, by using the notation 12.5F instead of 12.5).

Double precision has one drawback: it requires more space. This is a factor only when you have large amounts of floating-point values to be stored in a disk file. Then, and only then, should you consider using the single-precision type, **float**.

Example 1.3. *Convert Temperatures*

Every time I go to Canada, I have to convert Celsius temperatures to Fahrenheit in my head. If I had a handheld computer, it would be nice to tell it to do this conversion for me; computers are good at that sort of thing.

Here's the conversion formula. The asterisk (*), when used to combine two values, means "multiply by."

$$\text{Fahrenheit} = (\text{Celsius} * 1.8) + 32$$

Now, a useful program will take *any* value input for Celsius and then convert it. This requires the use of some new features:

- Getting user input
- Storing the value input in a variable

Here is the complete program. Create a new project called “convert.” Then enter the new program, and compile and run (press Ctrl + F5 if you’re using Microsoft).

convert.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double  ctemp, ftemp;

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;
    ftemp = (ctemp * 1.8) + 32;
    cout << "Fahrenheit temp is: " << ftemp;
    return 0;
}
```

Remember, yet again (!), that if and only if you’re working with Microsoft Visual Studio, you must leave the following line in at the beginning of the program:

```
#include "stdafx.h"
```

Programs are easier to follow when you add comments, which in C++ are notated by double slashes (*//*). Comments are ignored by the compiler (they have no effect on program behavior), but they are useful for humans. Here is the more heavily commented version:

convert2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;
```

convert2.cpp, cont.

```
int main()
{
    double ctemp; // Celsius temperature
    double ftemp; // Fahrenheit temperature

    // Get value of ctemp (Celsius temp).

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;

    // Calculate ftemp (Fahrenheit temp) and output.

    ftemp = (ctemp * 1.8) + 32;
    cout << "Fahrenheit temp is: " << ftemp << endl;

    return 0;
}
```

This commented version, although it's easier for humans to read, takes more work to enter. While following the examples in this book, you can always omit the comments or choose to add them later. Remember this cardinal rule for comments:



C++ code beginning with double slashes (//) is a comment and is ignored by the C++ compiler to the end of the line.

Using comments is always optional, although it is a good idea, especially if any humans (including you) are going to ever look at the C++ code.

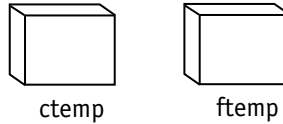


How It Works

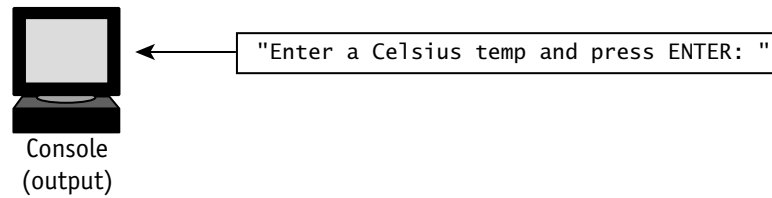
The first statement inside **main** declares variables of type **double**, `ctemp` and `ftemp`, which store Celsius temperature and Fahrenheit temperature, respectively.

```
double ctemp, ftemp;
```

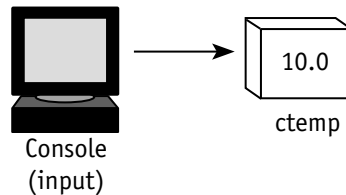
This gives us two locations at which we can store numbers. Because they have type **double**, they can contain fractional portions. Remember that **double** stands for “double-precision floating point.”



The next two statements prompt the user and then store input in the variable `ctemp`. Assume that the user types 10. Then the numeric value 10.0 is put into `ctemp`.



```
cout << "Enter a Celsius temp and press ENTER: " ;
```



```
cin >> ctemp;
```

In general, you can use similar statements in your own programs to print a prompting message and then store the input. The prompt is very helpful because otherwise the user may not know when he or she is supposed to do something.

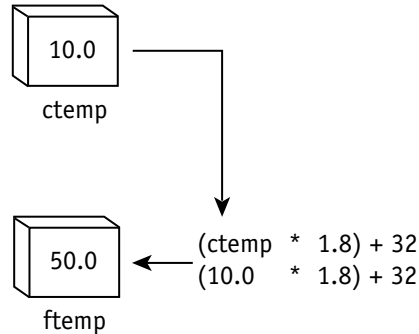
Note ► Although the number entered in this case was 10, it is stored as 10.0. In purely mathematical terms, 10 and 10.0 are equivalent, but in C++ terms, the notation 10.0 indicates that the value is stored in floating-point format rather than integer format. This turns out to have important consequences.

The next statement performs the actual conversion, using the value stored in `ctemp` to calculate the value of `ftemp`:

```
ftemp = (ctemp * 1.8) + 32;
```

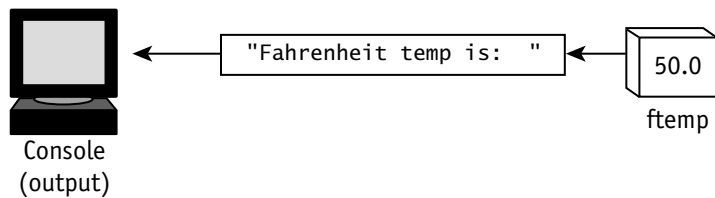
This statement features an *assignment*: the value on the right side of the equal sign (=) is evaluated and then copied to the variable on the left side. This is one of the most common operations in C++.

Again, assuming that the user input 10, this is how data would flow in the program:



```
ftemp = (ctemp * 1.8) + 32 ;
```

Finally, the program prints the result—in this case, 50.



```
cout << "Fahrenheit temp is: " << ftemp ;
```



Optimizing the Program

If you look at the previous example carefully, you might ask yourself, was it really necessary to declare two variables instead of one?

Actually, it wasn't. Welcome to the task of optimization. The following version improves on the first version of the program by getting rid of `ftemp` and combining the conversion and output steps:

convert3.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double ctemp; // Celsius temperature

    // Prompt and input value of ctemp.

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;

    // Convert ctemp and output results.

    cout << "Fahr. temp is: " << (ctemp * 1.8) + 32;
    cout << endl;

    return 0;
}
```

Do you detect a pattern by now? With the simplest programs, the pattern is usually as follows:

- 1 Declare variables.
- 2 Get input from the user (after printing a prompt).
- 3 Perform calculations and output results.

For example, the next program does something different but should look familiar. This program prompts for a number and then prints the square. The statements are similar to those in the previous example but use a different variable (x) and a different calculation.

square.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double x = 0.0;

    // Prompt and input value of x.

    cout << "Input a number and press ENTER: ";
    cin >> x;

    // Calculate and output the square.

    cout << "The square is: " << x * x << endl;
    return 0;
}
```

**EXERCISES**

- Exercise 1.3.1.** Rewrite the example so it performs the reverse conversion: Input a value into *ftemp* (Fahrenheit) and convert to *ctemp* (Celsius). Then print the results. (Hint: The reverse conversion formula is $ctemp = (ftemp - 32) / 1.8$.)
- Exercise 1.3.2.** Write the Fahrenheit-to-Celsius program using only one variable, *ftemp*. This is an optimization of Exercise 1.3.1.
- Exercise 1.3.3.** Write a program that inputs a value into a variable *x* and outputs the cube ($x * x * x$). Make sure the output statement uses the word *cube* rather than *square*.
- Exercise 1.3.4.** Rewrite the example *square.cpp* using the variable name *num* rather than *x*. Make sure you change the name everywhere “*x*” appears.

A Word about Variable Names and Keywords

This chapter has featured the variables `ctemp`, `ftemp`, and `n`. Exercise 1.3.4 suggested that you could replace “`x`” with “`num`,” as long as you do the substitution consistently throughout the program. So “`num`” is a valid name for a variable as well.

There is an endless variety of variable names I could have used instead. I could, for example, give some variables the names `killerRobot` or `GovernorOfCalifornia`.

What variable names are permitted, and what ones are not? You can use any name you want, as long as you follow these rules:

- The first character should be a letter. It cannot be a number. The first character can be an underscore (`_`), but the C++ library uses that naming convention internally, so it’s best to avoid starting a name that way.
- The rest of the name can be a letter, a number, or an underscore (`_`).
- You must avoid words that already have a special, predefined meaning in C++, such as the keywords.

It isn’t necessary to sit down and memorize all the C++ keywords. You need to know only that if you try using a name that conflicts with one of the C++ keywords, the compiler will respond with an error message. In that case, try a different name.



EXERCISE

Exercise 1.3.5. In the following list, which of the words are legal variable names in C++, and which are not? Review the rules just mentioned as needed.

`x1`
`EvilDarkness`
`PennsylvaniaAve1600`
`1600PennsylvaniaAve`
`Bobby_the_Robot`
`Bobby+the+Robot`
`whatThe???`
`amount`

```
count2
count2five
5count
main
main2
```

Chapter 1 *Summary*

Here are the main points of Chapter 1:

- ▶ Creating a program begins with writing C++ source code. This consists of C++ statements, which bear some resemblance to English. (Machine code, by contrast, is completely incomprehensible unless you look up the meaning of each combination of 1s and 0s.) Before the program can be run, it must be translated into machine code, which is all the computer really understands.
- ▶ The process of translating C++ statements into machine code is called *compiling*.
- ▶ After compiling, the program also has to be linked to standard functions stored in the C++ library. This process is called *linking*. After this step is successfully completed, you have an executable program.
- ▶ If you have a development environment, the process of compiling and linking a program (*building*) is automated so you need only press a function key. With Microsoft Visual Studio, press Ctrl+F5 to build programs.
- ▶ If you're working with Microsoft Visual Studio, make sure you leave **#include "stdafx"** at the beginning of every program. If you start a project by going through the New Project command, the environment will always put this in for you. Just make sure you don't delete **#include "stdafx"** when pasting code into the environment.

```
#include "stdafx.h"
```

- ▶ Simple C++ programs have the following general form:

```
#include <iostream>
using namespace std;

int main()
{
```

```
        Enter_your_statements_here!
        return 0;
    }
```

- To print output, use the **cout** object. For example:

```
    cout << "Never fear, C++ is here!";
```

- To print output and advance to the next line, use the **cout** object and send a newline character (**endl**). For example:

```
    cout << "Never fear, C++ is here!" << endl;
```

- Most C++ statements are terminated by a semicolon (;). Directives—lines beginning with a pound sign (#)—are a major exception.
- Double slashes (//) indicate a comment; all text to the end of the line is ignored by the compiler itself. But comments can be read by humans who have to maintain the program.
- Before using a variable, you must declare it. For example:

```
    double x;    // Declare x as a floating-pt variable.
```

- Variables that may store a fractional portion should have type **double**. This stands for “double-precision floating point.” The single-precision type (**float**) should be used only when storing large amounts of floating-point data on disk.
- To get keyboard input into a variable, you can use the **cin** object. For example:

```
    cin >> x;
```

- You can also put data into a variable by using assignment (=). This operation evaluates the expression on the right side of the equal sign (=) and places the value in the variable on the left side. For example:

```
    x = y * 2;    // Multiply y times 2, place result in x.
```

Index

Symbols

- (decrement operator), 53–54
- // (double slashes), 4, 20
- > (arrow operator), 290, 395
- + (addition-operator function), 452–454
- += (add and assign), 477, 482
- = (subtract and assign), 477, 482
- *= (multiply and assign), 477, 482
- /= (divide and assign), 477, 482
- ?: (conditional operator), 481
- , (join operator), 482
- < and > (angle brackets), 333
- = (assignment operator)
 - == vs., 35–36
 - for classes, 463–465
 - in decision making, 35–36
 - introduction to, 32
 - overview of, 482
 - for values, 45
- * (asterisk)
 - as indirection operator, 159–160, 179
 - as “multiply by” operator, 19
- \ (backslash notations), 216
- { } (braces), 102
- ' (digit separator)
 - example of, 425
 - introduction to, 415–417
 - in literal constants, 415
 - summary of, 444
- / (forward slash notations), 216
- ! (not operator), 51, 218
- % (percent sign)
 - as modulus division operator, 36–39, 71
 - as remainder-division operator, 71, 122
- :: (scope operator), 478, 555
- | OR operator, bitwise, 480–481
- || OR operator, logical, 51–53
- ++ (increment operator)
 - introduction to, 48–49
 - logical, 52–54
 - statements vs. expressions and, 49–51
 - zero-out-array function and, 179
- ++i prefix expression, 48–51
- == (equality operator), 32, 35–36
- <= (less-than-or-equal operator), 44–45
- == (test-for-equality operator), 465–466
- & (ampersand)
 - as bitwise operator, 480–481
 - creating reference variables with, 281–282, 287
 - getting addresses with, 159–162
 - in “and operator,” 51–54
 - reference arguments and, 172–173
- &, |, ^, ~, <<, >> (bitwise operators), 480–481
- && (AND operator, logical)
 - introduction to, 51–52
 - testing ages example of, 53–54
 - true values and, 53
- ## (concatenation operator), 507

Numbers

2-D arrays, 152–153

32-bit architecture

as default, 31

limits of storage on, 42

long integers in, 422–424

as standard on PCs, 158

64-bit architecture

adoption of, 158

literals in, example of, 427–431

literals in, generally, 424–425

long long integers in, 422–424

A

abs (absolute value) function, 254

Abstract classes

defined, 541

in polymorphic poker, 402–403

pure virtual functions in, 402, 412

Access

levels of, defined, 541

restricting, 243–246

Access class member operator (->), 290, 395

add function, 262–266, 460–462

Add-and-assign operator (+=), 477, 482

Adding machines

do-while loops in, 67–69

strings for text analysis in, 207–209

Addition-operator (+) function, 452–454

Addresses

appearance of, 157–158

defined, 541

pointers and, 155–158

printing, 161–162

of variables, 161–162

AI (artificial intelligence), 31–32

Algorithms

by Euclid, 119–122, 253

#include directive for, 367, 387

in poker games, 366–367

random_shuffle, 367, 387

selection sort, 167–168

swap, 367, 387

Alphabetical sorting, 298–300

Alt + F7 (properties command), 7

American National Standards Institute (ANSI), 541

The American Statistician, 351

Ampersand (&). *See* & (ampersand)

AND operator, bitwise (&), 480–481

AND operator, logical (&&), 51–53

Angle brackets (< and >), 333

ANSI (American National Standards Institute), 541

Applications, defined, 541

argument_list, 101

Arguments

constructors and, 281–282

defined, 99, 542

passing, 102–103

reference, 172–173, 555

Arithmetic

in Fraction class, 262–267

modular, 353

in pointers, 173–175

Arrays

2-D, 152–153

in C++, generally, 133–135

defined, 542

evaluating poker hands with, 378–379

in *Good Deal, Bad Deal* game, 341–342

initializing, 135

introduction to, 133

matrix and, 152–153

pointers and, 175–180

in poker games, 148–152, 387

in print out elements, 137–139

printing numbers from, 145–147

processing, 175–180

random numbers and, 139–144

sorting, 165–166

strings and, 144–148, 183–184

- summary of, 153–154
 - in text analysis, 183–184
 - zero-based indexing in, 135–139
- Arrow operator (->), 290, 395
- Artificial intelligence (AI), 31–32
- ASCII code
- defined, 542
 - extended, 515
 - overview of, 513–515
 - standard, 514
 - strings and, 181–184, 190
 - in text files, 222–224
- Assignment, defined, 23
- Assignment (=) operator. *See* = (assignment operator)
- Associativity, 542
- Asterisk (*)
- as indirection operator, 179
 - as “multiply by” operator, 19
- atof function, 195
- atoi function
- 64-bit literals and, 425
 - in *Good Deal, Bad Deal* game, 350
 - strings and, 195
- atoll function, 425
- auto keyword, 422
- avg (C) function
- calling, 100, 102–103
 - main function and, 105
 - overview of, 103–106
- ## B
- Backslash (\) notations, 216
- Backward compatibility, defined, 542
- Base 2 (binary) radix. *See* Binary (base 2) radix
- Base classes, defined, 542
- Basic, 96, 240
- Basic Input Output System (BIOS), 419
- begin function, 317–318
- Big Blue, 31
- Binary (base 2) radix
- bitwise operations and, 421
 - introduction to, 416
 - radix, defined, 554
 - summary of, 444
- Binary digits, 157–158
- Binary files, writing to, 226
- Binary literals, 416, 418–420
- Binary operations
- examples of, 227–232
 - exercises in, 233
 - introduction to, 225–227
 - random-access read, 230–233
 - random-access write, 227–230
- Binary Tree app
- alphabetical sorting in, 298–300
 - Bnode class in, 294–296
 - Btree class in, 296–301
 - overview of, 291–294
- BIOS (Basic Input Output System), 419
- Bits, defined, 542
- Bitset, defined, 543
- <bitset> template, 533–534
- Bitwise operations
- in C++14, 421
 - defined, 543
 - operators for, 52, 419–420, 480–481
- Blocks, 32–33, 543
- Bnode class, 294–296
- Bonacci, Leonardo, 432
- Book of Calculation (Liber Abaci)*, 432
- bool (Boolean) data type
- in decision making, generally, 47
 - logical operators and, 53
 - prime-number functions and, 107–108
 - random numbers and, 76
- Boole, George, 51
- Boolean operations
- data type in. *See* bool (Boolean) data type
 - in decision making, generally, 51–53
 - defined, 543
 - example of, 53–54
 - logical operators in, 51–52
 - true values in, 53

- braces (`{}`), 102
- Branch statements, 493–497
- break statements
 - in decision making, 46–47
 - keywords in, 42
 - prime-number functions and, 108–109
 - switch-case statements and, 78–79
 - syntax of, 496
- Btree class, 296–301
- Building programs, defined, 27
- Building strings, 186–189
- Bytes, defined, 543
- C**
- C++ class string
 - building strings with, 205–209
 - declaring/initializing variables of, 203
 - other operations using, 209–210
 - for text analysis, generally, 201–205
- C++ compilers
 - data types in, 17–19
 - defined, 544
 - double vs. single precision in, 19
 - `#include` in, 11–12
 - introduction to, 1, 27
 - keywords in, 26–27
 - non-Microsoft Visual Studio, 8
 - optimizing programs in, 23–25
 - printing messages in, generally, 9–11
 - printing multiple lines in, 13–15
 - printing newline characters in, 12
 - storing data in, 16–17
 - strings in, generally, 15–16
 - summary of, 27–28
 - temperature conversions in, 19–23
 - `using` statements in, 12
 - variable names in, 26–27
 - variables in, generally, 16–17
 - Visual Studio. *See* Visual Studio, Community Edition
- C++ for the Impatient*, 415
- C-strings
 - accessing characters inside, 209–210
 - in Adding Machine #2, 207–209
 - `cstring` for. *See* `cstring`
 - defined, 543
 - introduction to, 181–182
 - as null-terminated strings, 201–202
 - `strcmp` and, 204
 - string literals in, 204–205
- C++11
 - 64-bit literals in, example of, 427–431
 - 64-bit literals in, generally, 424–425
 - `auto` keyword in, 438–439
 - `decltype` keyword in, 438–439
 - defined, 543
 - delegating constructors in, 274–281
 - `for each` in, generally, 318, 433–435
 - `for each` in, setting arrays with, 435–437
 - `enum` classes in, 442–443
 - extended `enum` syntax, controlling storage with, 442–443
 - Fibonacci numbers in, 427–432
 - initializing members in, 271–274
 - localizing numbers in, 431–432
 - `long long` type in, 64-bit literals and, 424–425
 - `long long` type in, accepting input from, 425–426
 - `long long` type in, formatting numbers in, 426–427
 - `long long` type in, generally, 422–424
 - natural integers in, 424
 - new features in, generally, 422, 444–445
 - `nullptr` keyword in, 439–440
 - range-based `for` in, generally, 433–435
 - range-based `for` in, setting arrays with, 435–437
 - raw-string literals in, 443–444
 - strongly typed enumerations in, 440–443
- C++14
 - binary literals in, 416, 418–420
 - bitwise operations in, 421

- defined, 543
 - delegating constructors in, 274–281
 - digit-group separators in, 415–417
 - for `each` in, 318
 - initializing members in, 271–274
 - list templates in, 316
 - literal constants in, 415–417
 - new features in, exercises for, 421–422
 - new features in, generally, 415, 444–445
 - specifications for, 75
 - string-literal suffixes in, 415, 417–418
- Calculators. *See* RPN (Reverse Polish Notation)
- Callback functions, defined, 543–544
- Calling functions
- introduction to, 99–100
 - multiple functions, 105–106
 - overview of, 102–103
- Card class
- drawing cards and, 361–362
 - introduction to, 238
 - overview of, 363–364
 - in primitive video poker, 368–370
- Card games
- arrays in, 148–152
 - poker. *See* Poker
 - polymorphic poker. *See* Polymorphic poker
- Cascading errors, 5
- case labels, 80
- Case sensitivity, 5
- Casts
- defined, 544
 - operators for, 76–77, 479
 - `reinterpret_cast` operator, 225
 - `static_cast` operator, 76–77
 - type. *See* Type casts
- Celsius temperature conversions, 19–25
- Central processing units (CPUs), 545
- `char*`. *See also* C-strings
- constructors and, 285–286
 - converting to `long` long integers, 425
 - converting to numeric values, 211
 - raw string literals and, 443
 - as string type, 144
 - in text analysis, 184
 - type cast, 226–227, 235
- `char`
- building strings and, 186–188
 - C-strings and, 182
 - in expressions, 475
 - in file storage, 226–227, 235
 - single value of type, 373
 - size of, 423
 - string-literal suffixes and, 415
 - strings and, generally, 181–188
- `cin` (console input)
- file-stream objects in, generally, 213
 - introduction to, 16–17
 - strings for text analysis and, 190–192
 - summary of use of, 27–28
- Clarity of instructions, 31
- Class assignment (=) operator. *See also* = (assignment operator), 463–465
- Class-operator functions
- class assignment, 463–465
 - exercises in, 462, 471
 - in Fraction class, completed code for, 467–471
 - in Fraction class, generally, 457–462
 - in Fraction class, print function, 466–467
 - as global functions, 450–452
 - integers and, 463
 - introduction to, 447–450
 - operator overloading and, 472
 - optimizing code for, 461–462
 - in Point class, 454–457
 - printing with, 466–467
 - references, improving efficiency with, 452–454
 - summary of, 472–474
 - test-for-equality, 465–466
 - types, working with other, 463

Class string. *See* C++ class string

Classes

- abstract. *See* Abstract classes
- access restrictions and, 243–246
- add in, 262–266
- base, 542
- constructors and. *See* Constructors
- container, 314
- declaration of. *See* Declaration of classes
- defined, 544
- derived, 546
- exercises in, 248, 258, 262
- Fraction. *See* Fraction class
- greatest common factors in, 253–260
- I/O stream, 525–530
- `#include` in, 261
- inline functions in, 251–253
- introduction to, 237
- `<list>`, 315–316
- list container, 333
- `mult` in, 262–266
- objects and. *See* Objects
- operator functions and. *See* Class-operator functions
- pair container, 314
- Point. *See* Point class
- private, 243–246
- public, 241–242
- stack. *See* Stack classes
- in Standard Template Library, 531–539
- static storage, 556
- storage, 556
- structures and, 242–243, 267
- subclasses, 556–557
- summary of, 267–268
- support for functions in, 255–258
- testing, 246–248, 258–261
- virtual base, 393–396

`class_name`, 270

`clear` function, 372, 388

Code, defined, 544

Comments

- in decision making, 37
- double-precision floating data type and, 20–21
- introduction to, 4–5

Comparative languages, 96–97

Compatibility issues

- in C++11. *See* C++11
- in C++14. *See* C++14
- in Visual Studio, 5–8

Compilers. *See* C++ compilers

Compound statements, 32–33, 544

Concatenation operator (`##`), 507

Condition expressions

- example of, 91, 95–96
- introduction to, 86–88

Conditional operator (`?:`), 481

Conditions

- expressions for. *See* Condition expressions
- introduction to, 40
- true vs. false, 46–47

Console input (`cin`). *See* `cin` (console input)

Console output (`cout`). *See* `cout` (console output)

Console stream objects, 525–526

`const` keyword, 454–457, 462

Constants

- in 64-bit architecture, 424–431
- defined, 545
- digit-group separators in, 415
- literal, 415, 424–431
- predefined, 512

Constructors

- arguments and, 281–282
- copy. *See* Copy constructors
- default. *See* Default constructors
- defined, 545
- delegating, 274–281
- in Fraction class, 278–281, 285–286
- initializing members in, 271–274

- introduction to, 269–270
 - multiple, 270–271
 - overloading, 270–271
 - in Point class, 275–278, 283–284
 - reference variables and, 281–282, 284
 - returning objects from functions with, 381–387
 - strings and, 285–286
 - summary of, 286–287
 - warnings about, 272–274
- Container classes, 314
- Containers, 434–435
- Content *vs.* address of variables, 161–162
- Continually sorted lists, 321–322
- `continue` statements, 90, 496
- Control structures
- defined, 545
 - `do-while` loops. *See* `do-while` loops
 - introduction to, 65
 - summary of, 83
 - `switch-case` statements, 77–82
 - syntax of, 493–497
- Converting characters to uppercase, 195–197
- Copy constructors
- class assignment function and, 464
 - defined, 545
 - introduction to, 281–284
 - reference variables and, 287
- `cout` (console output)
- file-stream objects in, generally, 213
 - introduction to, 10–14
 - `ostream` and, 466–467
 - in polymorphic poker, 404–405
 - summary of use of, 28–29
- CPUs (central processing units), defined, 545
- Creating projects with Microsoft, 2–5
- Cstack class, 304–305
- `c_str` method, 212
- `cstring`. *See also* C-strings
- building strings with, 186–187
 - converting characters to uppercase, 195
 - defined, 517–518
 - getting numbers with, 192–194
 - `#include` directive and, 285
 - in RPN calculators, 329
 - support for, 203
- Ctrl+F5 (Start Without Debugging), 5–6, 8
- Cube *vs.* square, 25
- ## D
- Data-conversion functions, defined, 518
- Data declarations, 241–242
- Data flow in functions, 165
- Data members, 241–246, 545
- Data types
- in C++, generally, 17–19
 - in decision making, 29–31
 - escape sequences, 486–487
 - intrinsic, 484
 - of numeric literals, 485–486
 - overview of, 483
 - precision of, 484
 - signed integers, 487–489
 - string literals, 486–487
 - two's-complement format for, 487–489
- Dealing cards
- arrays in, 148–152
 - `deal_a_card` function for, 392–393
 - Deck class in. *See* Deck class
- Decision making
- artificial intelligence in, 31–32
 - `bool` data type in, 47
 - Boolean logic in, example of, 53–54
 - Boolean logic in, generally, 51–53
 - control structures in. *See* Control structures
 - structures
 - data types and, 29–31
 - `if` and `if-else` statements in, 32–35
 - increment operators in, 48–49
 - infinite loops in, 42, 47
 - introduction to, 29
 - loops in, example of, 42–45

Decision making (*continued*)

- loops in, generally, 39–42
- loops in, optimizing program for, 45–46
- math library in, example of, 55–57
- math library in, generally, 55
- math library in, optimizing program for, 57–58
- odd-or-even programs in, 36–39
- prime number test in, generally, 55–57
- prime number test in, optimizing program for, 57–58
- printing 1 to N in, 43–46
- in programs, generally, 31–34
- statements *vs.* expressions in, 49–51
- in Subtraction Game, 58–61
- summary of, 62–63
- testing ages in, 53
- true *vs.* false in, 46–47, 53
- = *vs.* == operators in, 35–36

Deck class

- drawing cards and, 361–363
- introduction to, 237
- multiple decks and, 389–391
- overview of, 364–367
- in primitive video poker example, 368–370
- switching decks at runtime and, 391–396

Declaration

- of class strings, 203
- of classes. *See* Declaration of classes
- defined, 545
- of functions, 101, 104
- of pointers, 158–160

Declaration of classes

- Fraction class, 249, 267
- introduction to, 241–242
- syntax in, 502–503

decltype keyword, 422

Deconstructors, defined, 546

Decrement operator (--), 53–54

Default constructors

- class assignment function and, 463
- defined, 545–546
- introduction to, 272–274

Default statements, 79–80

#define directives

- defined, 505–506
- localizing numbers with, 431–432
- in polymorphic poker, 392

defined function, 507

Definition of functions, 102–104, 546

Delegation

- of constructors, 274–281
- of tasks, 337–339

delete keyword, 290–291, 312

Deprecation, defined, 546

Dereferencing, defined, 546

Derived classes, defined, 546

Dice games, 127

Digit separator ('). *See* ' (digit separator)

Directives

- #define. *See* #define directives
- defined, 546
- #elif, 507–508
- #endif, 508
- #error, 508
- #if, 508–509
- #ifdef, 509–510
- #ifndef, 510
- #include. *See* #include directives
- #line, 511
- preprocessor. *See* Preprocessor directives
- #undef, 511

Directories, 215

Disk files

- file-stream objects in, generally, 213–214
- referring to, 215–216
- storage and, 213

Displaying text files, 219–222

Divisors, 108

do-while loops
adding machine example of, 67–69
guess-the-number game example of, 72–77
introduction to, 65–67
random numbers and, 69–77

do-while statements, 494

DoorManager class
inserted in game code, 347–350
introduction to, 343–346

double (double-precision floating data type)
declaring arrays with, 133–134
defined, 18–19
example of, 20–21
in file storage, 225
inefficiency of, 29
overview of, 29–30
single precision *vs.*, 19
summary of use of, 28

Double slashes (//), 4, 20

Double value, 195

Double_it function, 162–164

Draw-poker payout, 383–386

Drawing cards, 361–363, 373–378

Dynamic memory allocation, 289

Dynamic object creation, 289–290

E

Electronic storage. *See* File storage

Elements, 121

Elements of arrays, 133–134

#elif directive, defined, 507–508

Encapsulation, 214, 546

end function, 317–318

End users, 547

#endif directive, defined, 508

endl object, 13–14

enum declarations, 503–504

Equality (==) operator, 32, 35–36

#error directive, defined, 508

Escape sequences
data types and, 486–487
introduction to, 188
strings for text analysis and, 189–190

Euclid’s algorithm, 119–122, 253

Eval class, 379–386

Exceptions, defined, 547

Expressions, 324, 491–492

Extensibility, 404–405

F

Factors, 108, 116

Fahrenheit temperature conversions, 19–25

false
bool data type and, 47
Boolean logical operators and, 51–53
in decision making, 46–47, 53
is_prime for, 95–96
predefined, 57
random numbers and, 75–76
setting **do_more** to, 73–76
testing for equality and, 35
true *vs.*, 254
while statements and, 40, 44

Fibonacci numbers, 427–432

File I/O functions, 529–530

File storage
binary operations, examples of, 227–232
binary operations, exercises in, 230, 233
binary operations, generally, 225–227
disk files, referring to, 215–216
displaying text files, exercises, 222
displaying text files, generally, 219–221
file-stream objects in. *See* File-stream objects
introduction to, 213
random-access read, 230–233
random-access write, 227–230
summary of, 233–235
text *vs.* binary files in, 222–224
writing text to files in, 216–218

- File-stream objects
 - disk files, referring to, 215–216
 - exercises in, 219
 - introduction to, 213–215
 - writing text to files and, 216–218
- float (single-precision data type), 28
- FloatFraction, 400
- Floating point data
 - defined, 17–19, 547
 - double-precision. *See* `double` (double-precision floating data type)
 - example of, 20–21
 - single-precision, 28
- Floating-point division operator, 480
- Flush hands, 360
- Folders, 215
- for each, 422
- for loops
 - introduction to, 86–88
 - printing elements in arrays with, 137–138
 - for statements in, 88–90, 92–96
- for statements
 - in comparative languages, 96–97
 - exercises in, 92
 - introduction to, 85
 - in for loops, declaring variables, 92–96
 - in for loops, examples of, 88–91
 - in for loops, generally, 86–88
 - prime number tests with, 93–96
 - printing 1 to N with, 90–91
 - summary of, 97
 - while loops *vs.*, 90
- FORTRAN, 95, 136
- Forward Polish Notation, 325
- Forward slash (/) notations, 216
- fout, 214–215
- Fraction class
 - add in, 262–266
 - arithmetic functions in, 262–267
 - class operator functions and, 457–462
 - completed code for, 467–471
 - constructors in, 278–281
 - dynamic object creation in, generally, 289
 - exercises in, 258, 262, 267, 462
 - greatest common factors in, 253–260
 - `#include` in, 261
 - inline functions in, 251–253
 - introduction to, 248–251
 - lowest common denominators in, 254–260
 - `mult` in, 262–266
 - objects in, 285–286
 - optimizing code for, 461–462
 - print function and, 466–467
 - private access to, 249, 267
 - public members in, 249, 267
 - support for functions in, 255–258
 - testing in, 258–261
- Friend functions, 451–452
- `fstream`, 214
- Full house hands, 360
- Functions. *See also specific functions*
 - `avg()`. *See* `avg()` function
 - calling. *See* Calling functions
 - class-operator. *See* Class-operator functions
 - concept of, 99–101
 - declaring, 101, 104, 500–501
 - defining, 102, 104, 546
 - Euclid's algorithm and, 119–122
 - Fraction class supporting, 255–258
 - global, 109–112, 450–452
 - greatest common factors and, 119–122
 - inline, 251–253, 458
 - introduction to, 99
 - library. *See* Library functions
 - local *vs.* global variables in, 109–112
 - prime factorization in, 113–118
 - prime-number, 106–109
 - prototyping, 101
 - pure virtual. *See* Pure virtual functions

random-number generator, 127–129
recursive, 112–113, 122–126
returning objects from, 362–365,
381–387
in Subtraction Game, 129–130
summary of, 131–132
Tower of Hanoi puzzle and, 122–126
using, generally, 101

G

Games

dice, 127
Good Deal, Bad Deal. *See Good Deal, Bad Deal* game
Guess-the-number, 72–77
poker. *See Poker*
polymorphic poker. *See Polymorphic poker*
Subtraction, 58–61, 129–130
video. *See Video games*

Garbage, 135

GCFs (greatest common factors). *See* Greatest common factors (GCFs)

`get_divisors` function, 114–117

`get_good_prize`, `get_bad_prize`,
339–342

`getline` method, 188–192

`get_number` function, 190–195

Global functions, 450–452

Global variables, 547

Golden Ratio, 432

Good Deal, Bad Deal game

delegating tasks in, 337–339
DoorManager class in, generally,
343–346
DoorManager class, inserting in game
code, 347–350
example of complete code for, 347–350
exercises in, 342, 346–347, 350
introduction to, 335–337
optimizing code for, 341–342
paradox in, 351–353

PrizeManager class in, generally,
339–342

PrizeManager class in, optimizing,
353–356

PrizeManager class, inserted in game
code, 347–350
summary of, 356–357

`Goto` statements, 497

Graphical-user-interfaces (GUIs), 240

Greatest common factors (GCFs)
defined, 547

in Fraction class, 253–260
`main` function and, 119–122

Guess-the-number game, 72–77

GUIs (Graphical-user-interfaces), 240

H

Hall, Monty. *See also Good Deal, Bad Deal*
game, 335

Hard-coded array sizes, 341–342

Header files, defined, 547

Hexadecimal notation, 157

I

I/O (input/output) stream

console stream objects, 525–526

file I/O functions, 529–530

input stream functions, 528

manipulators, 526–527

objects and classes, generally, 525

output stream functions, 528

in polymorphic poker, 403–410

`i++` postfix expression, 48–51

IDEs (integrated development environments),
548

`#if` directive, 508–509

`if-else` statements

arrays *vs.*, 147

example of, 36–37

explanation of, 37–38

introduction to, 32–35

syntax of, 493

- if statements, 32–35, 38
- `#ifdef` directive, defined, 509–510
- `#ifndef` directive, defined, 510
- `ifstream`
 - displaying text files and, 219
 - introduction to, 214
 - summary of, 234
- Implementation, defined, 548
- `#include` directives
 - `<algorithm>`, 367, 387
 - `<cmath>`, 55
 - defined, 510–511
 - in Fraction class, 261
 - `<fstream>`, 213–214, 233
 - `<list>`, 315, 333
 - semicolons and, 6
 - `<stack>`, 327, 330, 334
 - "`stdafx.h`", 5–7, 12, 28–29
 - `<string>` class, 202–203
- Increment expressions, 41–42, 86–91, 95–97
- Increment operator (`++`). *See* `++` (increment operator)
- Independent, self-contained objects, 410–411
- Index numbers
 - defined, 548
 - of elements in arrays, 136
 - in one-based indexing, 552
 - in zero-based indexing, 135–139, 558
- Indirection, defined, 548
- Indirection operator (`*`), 159–160
- Individual characters *vs.* strings, 197–198
- Infinite loops
 - in decision making, 42, 47
 - defined, 548
- Infix notation, 324
- Inheritance
 - defined, 548
 - polymorphism and, 392–396
- Initialization
 - of arrays, 135
 - constructors for, 269
 - expressions for, 86–88, 91, 95–97
 - of Fraction objects from strings, 285–286
 - of members, 271–272
 - of objects, 283
- Inline functions
 - defined, 548
 - in Fraction class, 251–253
- Input, getting with `cin`, 16, 22, 526
- Input, getting from a file, 220–221, 528
- Input/output (I/O) stream. *See* I/O (input/output) stream
- Input stream functions, 528
- `insert` function, 319–322
- Installation of Visual Studio. *See also* Visual Studio, Community Edition, 1–2
- Instances/instantiation, defined, 548
- `int` (integer) data type
 - class operator functions and, 463
 - declaring arrays with, 134–135
 - in file storage, 226
 - introduction to, 17–18, 29
 - `long long`, 31, 42
 - operators and, 480
 - pointers to variables of, 158
 - size limitations on variables in, 42
 - summary of, 62
 - variables in, generally, 30
- Integers. *See also* `int` (integer) data type
 - defined, 548
 - value of, 175
 - variables of, 37
- Integrated development environments (IDEs), 548
- Intelligent data structures, 238
- Interfaces
 - defined, 549
 - in polymorphic poker, 402–403
- International formats, 431–432
- Intrinsic data types, 484
- `ios::in/ios::out`, 229–234

- iostream (virtual include files), 211
- IPrintable class, 405–409
- is_prime, 95, 106–108
- Iteration/iterative computing, defined, 549
- Iterators
 - in Binary Tree app, 298
 - defined, 549
 - listing, 321–322, 334
 - in STL, creating/using, 316–318
 - in STL, pointers *vs.*, 319
- iVec (vector of integers), 371–372
- J**
- Join (.) operator, 482
- K**
- Kasparov, Gary, 31
- Keywords
 - auto, 422
 - case, 78–80
 - class, 241
 - const, 454–457
 - decltype, 422
 - defined, 549
 - delete, 290–291, 312
 - do, 65–66
 - else, 33–34
 - for, 65–83
 - if, 32–35, 38
 - new, 289–291, 312
 - nullptr, 331, 422
 - return, 102–108
 - struct, 272–273
 - this, 464
 - in Visual Studio, 26–27
 - while, 40
- L**
- Labeled statements, 80
- Last-in-first-out (LIFO)
 - in calling functions, 113
 - defined, 549
 - in RPN calculators, 325
 - in Standard Template Library, 334
- Late binding
 - defined, 549, 557
 - in polymorphic poker, 412
- Law of Large Numbers, 140
- LCD (lowest common denominator), 254–260, 262–266
- LCM (lowest common multiple), 262–266, 549
- Left value (Lvalue), 550
- Less-than-or-equal operator (<=), 44–45
- Let's Make a Deal*, 335
- Liber Abaci (Book of Calculation)*, 432
- Library functions
 - <cstring>, 517–518
 - data-conversion, 518
 - math, 520
 - overview of, 517
 - randomization, 521
 - single-character, 519
 - strftime, formats for, 523–524
 - string, 517–518
 - time, 521–523
- LIFO (last-in-first-out) mechanisms. *See* Last-in-first-out (LIFO)
- #line directive, defined, 511
- Linking, 27
- <list> classes, 315–316
- List container classes, 333
- <list> template
 - in C++11/C++14, for each, 318
 - continually sorted lists in, 321–322
 - iterators in, creating/using, 316–318
 - <list> classes in, creating/using, 315–316
 - ordered list example in, 319–321
 - pointers *vs.* iterators in, 319
 - in Standard Template Library, 534–536
 - in STL, generally, 313–314
 - writing templates in C++ and, 314–315

Literal constants, 415

Literals

- binary, 416, 418–420
- defined, 549–550
- numeric, 485–486
- operator, 447–461
- public, 241
- raw string, 422
- return, 47, 102
- string, 415–418, 486–487, 556
- switch, 78–80
- virtual, 394
- while, 40

Local variables

- defined, 550
- global variables *vs.*, 109–112

Localizing numbers, 431–432

Location of data. *See* Pointers

Logical (Boolean) operators. *See also* Boolean operations, 51–53

Logical negation operator (!), 51, 218

Logical operations, defined, 550

long integer data type (`time_t`), 76

long long int data type

- in 64-bit architecture, 424–425, 427–431

accepting from C++11, 425–426

in C++14, 417

formatting in C++11, 426–427

infinite loops and, 42

introduction to, 31

Loop counters, defined, 550

Loops

- break keyword and, 58, 61
- for counting numbers, 85–86
- defined, 550
- exercises in, 46
- explanation of example of, 44
- for. *See* for loops
- infinite, 42
- introduction to, 39–42

optimizing program for, 45–46

in prime number test, 57

printing 1 to N example of, 42–45

Lowest common denominator (LCD), 254–260, 262–266

Lowest common multiple (LCM), 262–266, 549

Lukasiewicz, Jan, 324

Lvalue (left value), 550

M

Machine code

compiling C++ statements into, 27

defined, 550–551

linking to C++ functions, 27

main function

avg function and, 105

building strings and, 188

defined, 551

functions in, generally, 100–101

`get_divisors` function and, 115

greatest common factors and, 119–120

introduction to, 74–75

local *vs.* global variables in, 109–112

Point class and, 247

`rand_0toN1` function and, 140–143

Main memory, defined, 551

Manipulators, I/O stream, 526–527

Math

algorithms for. *See* Algorithms

arithmetic functions in. *See* Arithmetic

functions for, defined, 520

Math library

of decision making, 55–57

introduction to, 11

optimizing program for, 57–58

matrix, 152–153

Member functions, 268, 551

Members, defined, 551

Memory, 291, 551

Messages, printing, 9–11

Methods

- `c_str`, 212
- defined, 551
- `getline`, 188–192

Microsoft

- code developed at, 424
- Foundation Classes by, 240
- Visual Studio by. *See* Visual Studio, Community Edition

MOD 3 operation, 344–345

Modular arithmetic, 353

Modules, defined, 551

Modulus division (%) operator, 36–39, 71

Monty Hall game. *See* *Good Deal*, *Bad Deal* game

Monty Hall Paradox, 351–353

`mult` (multiplication) function, 19, 262–266, 460–462

Multiple constructors, 270–271

Multiple decks, 389–391

Multiple lines, printing, 13–15

N

Nested loops, 171

Nesting, defined, 551

`new` keyword, 289–291, 312

New-style type casts, 479

Newline characters

- defined, 551
- printing, 12–15

NIM (Subtraction Game), 58–61, 129–130

No-ops, 37

Non-Microsoft Visual Studio, 8

`normalize` function

- in Fraction class constructors, 278–280
- in Fraction class, generally, 255–261
- in Fraction class, math functions, 264
- virtual function calls and, 399–401

Not (!) operator, 51, 218

Null pointers

- in Binary Tree app, 292–296
- defined, 551–552
- introduction to, 71

`nullptr` keyword

- in C++11/C++14, 422
- introduction to, 142
- in RPN calculators, 331

Number-printing program, 80–82

Numeric literals, 485–486

O

Object code, defined, 552

Object-oriented programming (OOP)

- alphabetical sorting in, 298–300
- binary tree apps in, 291–294
- `Bnode` class in, 294–296
- `Btree` class in, 296–301
- `Cstack` class in, 304–305
- defined, 552
- `delete` keyword in, 290–291
- dynamic object creation in, 289–290
- examples of, 289
- exercises in, 300–301
- general steps in, 238–240
- Monty Hall game in. *See* *Good Deal*, *Bad Deal* game
- `new` keyword in, 289–291
- objects in, generally, 237–238, 240
- poker in. *See* Poker
- polymorphic poker in, 403–411
- pros vs. cons of, 240
- recursion vs. iteration in, 301–302
- stack classes in, 304
- summary of, 311–312
- Tower of Hanoi animation in, 302–311

Object-oriented programming systems (OOPS), 552

Objects. *See also* Classes

- defined, 238–240, 552
- I/O stream, 525–530
- independent and self-contained, 410–411
- introduction to, 237
- in OOP, generally, 237–238, 240
- in Standard Template Library, 531–539
- strings and, 191

- Odd-or-even programs, 36–39
- Offsets, 136
- `ofstream`, 214–215, 233
- Old-style type casts, 479
- One-based indexing, defined, 552
- OOP (object-oriented programming). *See* Object-oriented programming (OOP)
- OOPS (Object-oriented programming systems), 552
- Operands, defined, 553
- Operator overloading, 447, 472
- Operators. *See also specific operators*
 - assignment, *See* = (assignment operator)
 - bitwise, 52, 419–420, 480–481
 - cast, 76–77, 479
 - classes and. *See* Class-operator functions
 - conditional, 481
 - defined, 553
 - floating-point division, 480
 - increment. *See* ++ (increment operator)
 - integer, 480
 - join, 482
 - new-style type casts, 479
 - old-style type casts, 479
 - overview of, 475
 - by precedence level, 476–478
 - scope, 478
 - `sizeof`. *See* `sizeof` operator
 - type casts, 479
- Optimizing programs, in Visual Studio, 23–25
- OR operator, bitwise (`|`), 480–481
- OR operator, logical (`||`), 51–53
- Ordered lists, 319–321
- `ostream` (output stream)
 - defined, 528
 - `IPrintable` class and, 408
 - print function and, 466–467
- Output, printing to the console, 10–12, 512
- Output stream (`ostream`). *See* `ostream` (output stream)
- Output, writing to a file, 213–218, 529–530
- Overloading
 - constructors and, 270–271
 - defined, 553
 - operator, 447, 472
- P**
- `p = arr` statements, 174
- Pair container classes, 314
- Parade* magazine, 351
- Paradoxes
 - in Monty Hall game, 351–353
 - Russell’s, 295
- Pascal, 240
- Pass by reference, 165–172
- Passing arguments, 102–103
- Passing pointers, 165–173
- Pausing screens, 7–8
- Penrose, Roger, 32
- Percent sign (%). *See* % (percent sign)
- Persistent memory, 553
- Pinochle, 389–396
- Point class
 - class operator functions and, 454–457
 - constructors in, 275–278, 283–284
 - declaration of, 241–242
 - delegating constructors in, 274–275
 - initializing members in, 271–274
 - private access to, 243–248
 - public access in, 241–242
 - testing, 246–248
- Pointer-indirection (*) operator, 179
- Pointers
 - addresses, appearance of, 157–158
 - arithmetic in, 173–175
 - array processing and, 175–180
 - array sorting and, 165–166
 - in Binary Tree app, 292–298, 309
 - concept of, 156–157
 - content vs. address of variables, 161–162
 - data flow in functions and, 165
 - declaring, generally, 158–160
 - defined, 155–156, 553

- double_it function and, 162–164
 - in dynamic object creation, 289–290
 - introduction to, 155
 - iterators vs., 319
 - printing addresses and, 161–162
 - reference arguments and, 172–173
 - reference variables and, 281
 - summary of, 180
 - swap function and, generally, 165–166
 - swap function for sorting arrays and, 166–172
 - using, generally, 158–160
- Poker
- algorithms for, 366–367
 - arrays in, 148–152
 - Card class in, 363–364
 - Deck class in, 364–366
 - draw-poker payout in, 383–386
 - drawing cards in, example of code for, 373–378
 - drawing cards in, generally, 361–363
 - evaluating hands in, 378–382
 - exercises in, 370, 378, 386–387
 - getting numbers from players in, 373
 - introduction to, 359
 - polymorphic. *See* Polymorphic poker
 - primitive version of, 368–370
 - strategy for winning, 359–360
 - summary of, 387–388
 - vector template for, 371–377
- Polish Notation. *See also* RPN (Reverse Polish Notation), 325
- Polymorphism
- abstract classes/interfaces in, 402–403
 - cout in, 404–405
 - #define directives in, 392
 - exercises in, 399, 409–410
 - extensibility in, 404–405
 - I/O in, 403–410
 - independent, self-contained objects in, 410–411
 - introduction to, 389
 - multiple decks in, 389–391
 - OOP in, 403–411
 - polymorphism in, cout vs., 404–405
 - polymorphism in, generally, 392–396
 - polymorphism in, IPrintable class for, 405–409
 - pure virtual functions in, 401–402
 - summary of, 412–413
 - switching decks at runtime in, 391–392
 - virtual dealers in, 396–399
 - virtual penalties in, 399–400
- Polymorphism, defined, 412, 553
- Pop function
- introduction to, 309–310
 - in RPN calculators, 328–331
 - in Standard Template Library, 334
- Precedence levels
- defined, 553
 - operators by, 476–478
- Precision of data types, 484
- Precompiled headers, 6–7
- Predefined constants, 512
- Preprocessor directives
- concatenation operator, 507
 - #define, 505–506
 - defined, 507
 - #elif, 507–508
 - #endif, 508
 - #error, 508
 - #if, 508–509
 - #ifdef, 509–510
 - #ifndef, 510
 - #include, 510–511
 - #line, 511
 - overview of, 505
 - predefined constants, 512
 - #undef, 511
- Prime factorization, 113–118
- Prime number functions, 106–109
- Prime number tests, 55–57, 93–96

printf, 403

Printing

- 1 to N, 43–46, 90–91
- addresses, 161–162
- with class operator functions, 466–467
- elements in, 137–139
- messages, 9–11
- multiple lines, 13–15
- newline characters, 12–15
- number-printing program for, 80–82
- numbers, 145–147
- output with `cout`, 10–12, 525
- square roots, 190–195
- with `for` statements, 90–91

PrizeManager class

- inserted in game code, 347–350
- introduction to, 339–342
- optimizing, 353–356

Procedures. *See* Functions

Processors, 545

Programs. *See also specific programs*

- building, generally, 27
- decision making in, 31–34
- defined, 554
- odd-or-even, 36–39
- optimizing, generally, 23–25
- translation of, 182–183
- writing, generally, 5

Promoting values, 103

Properties command (Alt + F7), 7

Prototypes, defined, 554

Prototyping functions, 101

Pseudo-random numbers, 128

Pure virtual functions

- defined, 554
- in polymorphic poker, 401–402, 412

push function, 309–310

push_back function, 371, 388

push_back member function, 315, 328, 333–334

push_front member function, 315, 328, 333–334

Q

query_door function, 339

R

Radix, defined. *See also* Binary (base 2) radix, 554

rand function, 71–75

rand_0toN1 function, 139–143

Random-access read, 230–233

Random-access write, 227–230

Random numbers

- arrays and, 139–144
- do-while loops and, 69–77
- generator for, 127–129
- guess-the-number game example of, 72–76
- introduction to, 69–71
- optimizing code for, 76–77

Randomization functions, defined, 521

random_shuffle algorithm, 367, 387

Range-based “for,” 422, 554

Range, defined, 554

Raw pointers, 319

Raw string literals, 422

read function, 190, 226, 230–234

Records, 229–230

Recursion

- in Binary Tree app, 294, 297–298, 312
- defined, 554
- functions for, generally, 112–113
- iteration *vs.*, 301–302
- prime factorization in, 114
- in Tower of Hanoi puzzle, 122–126

Reference arguments

- defined, 555
- pointers and, 172–173
- swap behavior with, 282

Reference variables

- constructors and, 281–282, 284
- copy constructors and, 287
- defined, 555

References

- arguments. *See* Reference arguments
- class operator functions and, 452–454
- passing by, 165–172
- variables in. *See* Reference variables

`reinterpret_cast` operator, 225

Relational operators, 44

Remainder division, 37–39, 59–61

Remainder-division operator (%), 71, 122

return statements

- functions and, generally, 102
- `get_divisors` function and, 116
- introduction to, 47
- prime-number functions and, 108
- syntax of, 494–495, 497

Return values

- introduction to, 99–102
- local *vs.* global variables in, 111–112
- passing pointers and, 165
- in poker games, 375–377, 381–387

Returning objects from functions, 362–365, 381–387

`return_type` data, 101

Reverse Polish Notation (RPN). *See* RPN (Reverse Polish Notation)

Rings, moving, 302–303

Royal flush hands, 360

RPN (Reverse Polish Notation)

- design of, generally, 323–325
- example of code for, 329–332
- exercises in design of, 332
- stack classes for, 327–328
- stacks for, 325–327

Running programs, 5–6

Russell, Bertrand, 295

S

s (string-literal suffix), 415, 417–418

Scaling integers, 174–177

Scope, defined, 555

Scope (: :) operator, 478, 555

Searle, John, 183

Seed, 70–71

`Seekp` member function, 211, 234–235

Select door functions, 339–346

Selection sort algorithm, 167–168

Selvin, Steve, 351

Semicolons

- blocks and, 33
- C++ statements and, 10
- class/data declarations and, 242, 268
- data declarations and, 135
- function prototypes and, 135
- `#include` directives and, 6
- statements *vs.* expressions and, 49–50
- summary of use of, 28

`set_sel_door` function, 343

Short-circuit (Boolean) logic. *See also* Boolean operations, 51–54

Shuffling cards, 148–152

Side effects, 50

Signed integers, 487–489

Simula, 240

Single-character functions, 519

Single-precision data type (`float`), 28

`size` function, 371, 388

`sizeof` operator

- defined, 478–479
- in file storage, 227, 235
- in *Good Deal, Bad Deal* game, 341–342
- in poker games, 357

Smalltalk, 240, 410

`sort` functions, 319–322

Sorting arrays, 167–171

Source files, defined, 555

`sqrt` (square root) function, 11, 99

Square *vs.* cube, 25

Squirt function, 55

`srand`, 70–76

Stack classes

- design of, 304
- in object-oriented programming, 304–305
- in RPN calculator design, 327–328
- use of, 304–305

<stack> template, 538–539

Stacks

of calls, 112–113

classes in. *See* Stack classes

defined, 555

in RPN calculator design, 325–327

Standard Template Library (STL). *See* STL
(Standard Template Library)

Start Without Debugging (Ctrl+F5), 5–6, 8

start_new_game function, 339

Statement blocks, 32–33, 544

Statements. *See also specific statements*

branch, 493–497

compound, 32–33, 544

default, 79–80

defined, 555–556

expressions vs., 49–51

labeled, 80

syntax of basic, 492

Static storage classes, defined, 556

static_cast operator, 76–77

std namespace, 314–315

std:: prefix, 12, 328

stdafx.h, 6–7

STL (Standard Template Library)

angle brackets in, 333

<bitset> template in, 533–534

in C++11/C++14, 318

classes and objects in, generally, 531

continually sorted lists in, 321–322

defined, 555

for each in, 318

introduction to, 313

iterators in, 316–318

<list> classes in, 315–316

<list> template in, 313–314,
534–536

ordered list example in, 319–321

pointers vs. iterators in, 319

Polish Notation and, 325

prerequisites for using, 240

RPN calculators, designing generally,
323–325

RPN calculators, example of code for,
329–332

RPN calculators, exercises in design of, 332

RPN calculators, stack classes for,
327–328

RPN calculators, stacks for, 325–327

<stack> template in, 538–539

string class in, 531–533

summary of, 333–334

<vector> template in, 536–538

writing templates in, 314–315

Storage

classes, 556

of files. *See* File storage

in Visual Studio, 16–17

Straight flush hands, 360

Straight hands, 360

strcat (string concatenation), 185–188

strcmp (string compare), 219

strcpy (string copy), 185–188

Streams

file-stream objects. *See* File-stream
objects

I/O. *See* I/O (input/output) stream

introduction to, 213

iostream, 211

stringstream, 426–427

strftime functions, 523–524

String class

introduction to, 144–145

in Standard Template Library, 531–533

String functions

compare (strcmp), 219

concatenation (strcat), 185–188

copy (strcpy), 185–188

defined, 517–518

String-literal suffix (s), 415, 417–418

String literals, 486–487, 556

String-manipulation functions, 184–190

- Strings
 - arrays and, 144–148
 - building, 186–189
 - C-strings, 543
 - class. *See* C++ class string
 - constructors, 285–286
 - defined, 556
 - for text analysis. *See* Strings, for text analysis
 - in Visual Studio, generally, 15–16
- Strings, for text analysis
 - Adding Machine #2, 207–209
 - arrays and, 183–184
 - building, 186–189
 - building strings, 205–209
 - C++ class string in, generally, 201–205, 209–210
 - converting characters to uppercase, 195–197
 - declaring/initializing variables in, 203
 - escape sequences and, 189–190
 - getting numbers with, 190–195
 - `#include <string>` class support, 202–203
 - individual characters *vs.*, 197–198
 - introduction to, 181
 - printing square roots with, 190–195
 - reading input and, generally, 190
 - string-manipulation functions and, 184–190
 - `strtok` in, 198–201
 - summary of, 210–211
 - text storage on computers and, 181–182
 - translation of programs, 182–183
 - variables of class string, 203–205
- `stringstream` class, 426–427
- Strongly typed enumerations, 422
- Stroustrup, Bjarne, 240, 472
- `strtok` function
 - for breaking up input, 198–201
 - in RPN calculators, 323, 329–331
- `struct` keyword, 272–273
- Structures, 242–243, 267
- Subclasses, defined, 556–557
- Subroutines. *See* Functions
- Subtract-and-assign operator (`--`), 477, 482
- Subtraction Game (NIM), 58–61, 129–130
- Subtraction operator (`-`), 476
- swap algorithm, 367, 387
- Swap function
 - pointers and, generally, 165–166
 - for sorting arrays, 166–172
- `switch-case` statements
 - arrays *vs.*, 145, 147
 - introduction to, 77–82
 - number-printing program example of, 80–82
 - in RPN calculators, 331–332
 - syntax of, 495–496
- `switch` statements, 142
- Switching decks at runtime, 391–392
- Symbols, defined, 557
- Syntax
 - of basic expressions, 491–492
 - of basic statements, 492
 - of branch statements, 493–497
 - of break statements, 496
 - of class declarations, 502–503
 - of `continue` statements, 496
 - of control structures, 493–497
 - of `do-while` statements, 494
 - of enum declarations, 503–504
 - of function declarations, 500–501
 - of `goto` statements, 497
 - of `if-else` statements, 493
 - of return statements, 497
 - of for statements, 494–495
 - of `switch-case` statements, 495–496
 - of throw statements, 497
 - of variable declarations, 498–500
 - of `while` statements, 493–494

- T**
- Temperature conversions, 19–25
- Templates. *See also specific templates*
defined, 557
in Standard Template Library. *See* STL
(Standard Template Library)
- Test-for-equality (==) operator, 465–466
- Tests
of ages, 53
for equality, 465–466
prime number, 55–57, 93–96
- Text
analyzing with strings. *See* Strings, for
text analysis
binary files *vs.*, 222–224
displaying, 222
storage of, 181–182
string data, generally, 17–18
strings. *See* Strings
writing to files, 216–218
- `this` keyword, 464
- throw statements, 497
- Time functions, defined, 521–523
- `time_t` (long integer data type), 76
- `t_main`, 5, 7
- Tokens, defined, 557
- Top and pop operations, 328–331, 334
- Top of Stack (`tos`), 302–303, 309
- Tower of Hanoi puzzle
animating, example of, 305–311
animating, exercises in, 311
animating, generally, 302–304
`Cstack` class in, using, 304–305
functions in, 122–126
stack classes in, designing, 304
- Translation of programs, 182–183
- `true`
as absolute value function, 254
in Boolean logic, 51–54, 108–109
Boolean variables and, 75–76, 83
break statements and, 42
in decision making, 46–47, 53
`if` statements and, 32–35
`is_prime` for, 95
nonzero values as, 97
in prime number tests, 55–57
random numbers and, 75–76
reversing, 218
in strings, 204
in Subtraction Game, 59–61
`while` statements and, 40–44
`while(true)`, 69, 107–108, 130
- TV programs. *See* *Good Deal*, *Bad Deal*
game
- Two pair hands, 360
- Two's-complement
defined, 557
format for data types, 487–489
introduction to, 29
- Type casts
defined, 544
in file storage, 225
new-style, 479
old-style, 479
operators, 479
- Types
casts of. *See* Type casts
class operator functions and, 463
of data. *See* Data types
double-precision floating. *See* `double`
(double-precision floating data type)
integer, *See* `int` (integer) data type
Single-precision floating, 28
in STL, generally, 313
- U**
- `#undef` directive, defined, 511
- Unsigned long long integers, 425
- Unsigned short/unsigned long integers, 423
- Uppercase characters, 195–197

using statements
 namespace `std`, 328, 333
 printing messages and, 12

V

Values

absolute value function, 254
assignment operator for, 45
double, 195
false. *See* `false`
left, 550
pointers and, generally, 156
random numbers and, 75–76
return. *See* Return values
true. *See* `true`

Variables

in C++, generally, 16–17
data types in, 17–19
declaring, 498–500
defined, 557
names of, 26–27
pointers as, 156
summary of use of, 28

Vector of integers (`iVec`), 371–372

vector template

in poker, generally, 371–377
in polymorphic poker, 388
in Standard Template Library, 536–538

Vectors, defined, 557

Video games

poker, generally. *See* Poker
polymorphic poker. *See* Polymorphic
 poker

Vinci, Leonardo da, 432

virtual base classes, 393–396

Virtual dealers, 396–399

Virtual dice, 69–70

Virtual functions

address resolution for, 412
defined, 557–558

Virtual include files (`istream`), 211

Virtual keyword, 394

Virtual penalty, 399–400, 412

Visibility, defined, 558

Visual Studio, Community Edition

compatibility issues and, 5–8
creating projects with Microsoft and, 2–5
data types in, 17–19
double vs. single precision in, 19
installation of, 1–2
introduction to, 1
keywords in, 26–27
optimizing programs in, 23–25
pausing screens in, 7–8
printing messages in, generally, 9–11
printing multiple lines in, 13–15
printing newline characters in, 12
running, generally, 5–6
running programs in, 5–6
`stdafx.h` in, 6–7
storing data in, 16–17
strings in, generally, 15–16
summary of, 27–28
temperature conversions in, 19–23
variable names in, 26–27
variables in, 16–17
writing programs in, 5

Visual Studio, Non-Microsoft, 8

Vitruvian Man, 432

`void` pointers, 404–405

vos Savant, Marilyn, 351–353

vtable pointers (`vtpr`), 400

W

while loops

do-while loops vs., 65–67
for loops vs., 90
printing 1 to N with, 43–45
zero-out-array function and, 178

while statements, 40, 493–494

`while(true)`, 69

`write` function

 generally, 226

 random-access, 227–230, 234–235

Writing programs, in Visual Studio, 5

Writing templates, in STL, 314–315

Writing text to files, 216–218

Writing to a binary file, 226

Z

Zero-based indexing, 135–139, 558

`zero-out-array` function, 177–179

This page intentionally left blank



REGISTER YOUR PRODUCT at informit.com/register Access Additional Benefits and SAVE 35% on Your Next Purchase

- Download available product updates.
- Access bonus material when applicable.
- Receive exclusive offers on new editions and related products.
(Just check the box to hear from us when setting up your account.)
- Get a coupon for 35% for your next purchase, valid for 30 days. Your code will be available in your InformIT cart. (You will also find it in the Manage Codes section of your account page.)

Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com you can

- Shop our books, eBooks, software, and video training.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletters (informit.com/newsletters).
- Read free articles and blogs by information technology experts.
- Access thousands of free chapters and video lessons.

Connect with InformIT—Visit informit.com/community

Learn about InformIT community events and programs.



informIT.com

the trusted technology learning source

Addison-Wesley • Cisco Press • IBM Press • Microsoft Press • Pearson IT Certification • Que • Sams • VMware Press