



Stephen Prata

Sixth Edition

# C++ Primer Plus

**Developer's Library**



# C++ Primer Plus

---

Sixth Edition

# Developer's Library

## ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

*Developer's Library* books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

*PHP & MySQL Web Development*

Luke Welling & Laura Thomson  
ISBN-13: 978-0-672-32916-6

*MySQL*

Paul DuBois  
ISBN-13: 978-0-672-32938-8

*Linux Kernel Development*

Robert Love  
ISBN-13: 978-0-672-32946-3

*Python Essential Reference*

David Beazley  
ISBN-13: 978-0-672-32862-6

*PostgreSQL*

Korry Douglas  
ISBN-13: 978-0-672-32756-8

*C++ Primer Plus*

Stephen Prata  
ISBN-13: 978-0-321-77640-2

Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**.

**Developer's  
Library**

[informit.com/devlibrary](http://informit.com/devlibrary)

# C++ Primer Plus

---

Sixth Edition

Stephen Prata

◆ Addison-Wesley

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

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.

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

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

For sales outside the United States, please contact:

**International Sales**  
**international@pearson.com**

Visit us on the Web: [informit.com/aw](http://informit.com/aw).

*Library of Congress Cataloging-in-Publication data is on file.*

Copyright © 2012 Pearson Education, Inc.

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

ISBN-13: 978-0-321-77640-2

ISBN-10: 0-321-77640-2

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

First printing: October 2011

**Acquisitions Editor**  
Mark Taber

**Development Editor**  
Michael Thurston

**Managing Editor**  
Kristy Hart

**Project Editors**  
Samantha Sinkhorn  
Jovana Shirley

**Copy Editor**  
Bart Reed

**Indexer**  
Lisa Stumpf

**Proofreader**  
Language Logistics, LLC

**Technical Reviewer**  
David Horvath

**Publishing Coordinator**  
Vanessa Evans

**Cover Designer**  
Gary Adair

**Compositor**  
Nonie Ratcliff



*To my parents, with love.*



# Contents at a Glance

Introduction	1
1 Getting Started with C++	9
2 Setting Out to C++	27
3 Dealing with Data	65
4 Compound Types	115
5 Loops and Relational Expressions	195
6 Branching Statements and Logical Operators	253
7 Functions: C++'s Programming Modules	305
8 Adventures in Functions	379
9 Memory Models and Namespaces	447
10 Objects and Classes	505
11 Working with Classes	563
12 Classes and Dynamic Memory Allocation	627
13 Class Inheritance	707
14 Reusing Code in C++	785
15 Friends, Exceptions, and More	877
16 The <code>string</code> Class and the Standard Template Library	951
17 Input, Output, and Files	1061
18 Visiting with the New C++ Standard	1153

## **Appendixes**

- A Number Bases 1215**
- B C++ Reserved Words 1221**
- C The ASCII Character Set 1225**
- D Operator Precedence 1231**
- E Other Operators 1235**
- F The `string` Template Class 1249**
- G The Standard Template Library Methods and Functions 1271**
- H Selected Readings and Internet Resources 1323**
- I Converting to ISO Standard C++ 1327**
- J Answers to Chapter Reviews 1335**
  
- Index 1367**



# Table of Contents

## Introduction 1

### 1 Getting Started with C++ 9

- Learning C++: What Lies Before You 10
- The Origins of C++: A Little History 10
- Portability and Standards 15
- The Mechanics of Creating a Program 18
- Summary 25

### 2 Setting Out to C++ 27

- C++ Initiation 27
- C++ Statements 41
- More C++ Statements 45
- Functions 48
- Summary 61
- Chapter Review 62
- Programming Exercises 62

### 3 Dealing with Data 65

- Simple Variables 66
- The `const` Qualifier 90
- Floating-Point Numbers 92
- C++ Arithmetic Operators 97
- Summary 109
- Chapter Review 110
- Programming Exercises 111

### 4 Compound Types 115

- Introducing Arrays 116
- Strings 120
- Introducing the `string` Class 131
- Introducing Structures 140
- Unions 149
- Enumerations 150
- Pointers and the Free Store 153
- Pointers, Arrays, and Pointer Arithmetic 167
- Combinations of Types 184
- Array Alternatives 186
- Summary 190
- Chapter Review 191
- Programming Exercises 192

**5 Loops and Relational Expressions 195**

- Introducing `for` Loops 196
- The `while` Loop 224
- The `do while` Loop 231
- The Range-Based `for` Loop (C++11) 233
- Loops and Text Input 234
- Nested Loops and Two-Dimensional Arrays 244
- Summary 249
- Chapter Review 250
- Programming Exercises 251

**6 Branching Statements and Logical Operators 253**

- The `if` Statement 254
- Logical Expressions 260
- The `cctype` Library of Character Functions 270
- The `?:` Operator 273
- The `switch` Statement 274
- The `break` and `continue` Statements 280
- Number-Reading Loops 283
- Simple File Input/Output 287
- Summary 298
- Chapter Review 298
- Programming Exercises 301

**7 Functions: C++'s Programming Modules 305**

- Function Review 306
- Function Arguments and Passing by Value 313
- Functions and Arrays 320
- Functions and Two-Dimensional Arrays 337
- Functions and C-Style Strings 339
- Functions and Structures 343
- Functions and `string` Class Objects 353
- Functions and `array` Objects 355
- Recursion 357
- Pointers to Functions 361
- Summary 371
- Chapter Review 372
- Programming Exercises 374

**8 Adventures in Functions 379**

- C++ Inline Functions 379
- Reference Variables 383
- Default Arguments 409
- Function Overloading 412
- Function Templates 419

Summary	442
Chapter Review	443
Programming Exercises	444
<b>9 Memory Models and Namespaces</b>	<b>447</b>
Separate Compilation	447
Storage Duration, Scope, and Linkage	453
Namespaces	482
Summary	497
Chapter Review	498
Programming Exercises	501
<b>10 Objects and Classes</b>	<b>505</b>
Procedural and Object-Oriented Programming	506
Abstraction and Classes	507
Class Constructors and Destructors	524
Knowing Your Objects: The <code>this</code> Pointer	539
An Array of Objects	546
Class Scope	549
Abstract Data Types	552
Summary	557
Chapter Review	558
Programming Exercises	559
<b>11 Working with Classes</b>	<b>563</b>
Operator Overloading	564
Time on Our Hands: Developing an Operator	
Overloading Example	565
Introducing Friends	578
Overloaded Operators: Member Versus Nonmember Functions	587
More Overloading: A Vector Class	588
Automatic Conversions and Type Casts for Classes	606
Summary	621
Chapter Review	623
Programming Exercises	623
<b>12 Classes and Dynamic Memory Allocation</b>	<b>627</b>
Dynamic Memory and Classes	628
The New, Improved <code>String</code> Class	647
Things to Remember When Using <code>new</code> in Constructors	659
Observations About Returning Objects	662
Using Pointers to Objects	665
Reviewing Techniques	676
A Queue Simulation	678

Summary 699  
Chapter Review 700  
Programming Exercises 702

### **13 Class Inheritance 707**

Beginning with a Simple Base Class 708  
Inheritance: An *Is-a* Relationship 720  
Polymorphic Public Inheritance 722  
Static and Dynamic Binding 737  
Access Control: `protected` 745  
Abstract Base Classes 746  
Inheritance and Dynamic Memory Allocation 757  
Class Design Review 766  
Summary 778  
Chapter Review 779  
Programming Exercises 780

### **14 Reusing Code in C++ 785**

Classes with Object Members 786  
Private Inheritance 797  
Multiple Inheritance 808  
Class Templates 830  
Summary 866  
Chapter Review 869  
Programming Exercises 871

### **15 Friends, Exceptions, and More 877**

Friends 877  
Nested Classes 889  
Exceptions 896  
Runtime Type Identification 933  
Type Cast Operators 943  
Summary 947  
Chapter Review 947  
Programming Exercises 949

### **16 The `string` Class and the Standard Template Library 951**

The `string` Class 952  
Smart Pointer Template Classes 968  
The Standard Template Library 978  
Generic Programming 992  
Function Objects (a.k.a. Functors) 1026  
Algorithms 1035  
Other Libraries 1045

Summary	1054
Chapter Review	1056
Programming Exercises	1057
<b>17 Input, Output, and Files</b>	<b>1061</b>
An Overview of C++ Input and Output	1062
Output with <code>cout</code>	1069
Input with <code>cin</code>	1093
File Input and Output	1114
Incore Formatting	1142
Summary	1145
Chapter Review	1146
Programming Exercises	1148
<b>18 Visiting with the New C++ Standard</b>	<b>1153</b>
C++11 Features Revisited	1153
Move Semantics and the Rvalue Reference	1164
New Class Features	1178
Lambda Functions	1184
Wrappers	1191
Variadic Templates	1197
More C++11 Features	1202
Language Change	1205
What Now?	1207
Summary	1208
Chapter Review	1209
Programming Exercises	1212
<b>Appendixes</b>	
<b>A Number Bases</b>	<b>1215</b>
<b>B C++ Reserved Words</b>	<b>1221</b>
<b>C The ASCII Character Set</b>	<b>1225</b>
<b>D Operator Precedence</b>	<b>1231</b>
<b>E Other Operators</b>	<b>1235</b>
<b>F The <code>string</code> Template Class</b>	<b>1249</b>
<b>G The Standard Template Library Methods and Functions</b>	<b>1271</b>
<b>H Selected Readings and Internet Resources</b>	<b>1323</b>
<b>I Converting to ISO Standard C++</b>	<b>1327</b>
<b>J Answers to Chapter Reviews</b>	<b>1335</b>
<b>Index</b>	<b>1367</b>

# Acknowledgments

## Acknowledgments for the Sixth Edition

I'd like to thank Mark Taber and Samantha Sinkhorn of Pearson for guiding and managing this project and David Horvath for providing technical review and editing.

## Acknowledgments for the Fifth Edition

I'd like to thank Loretta Yates and Songlin Qiu of Sams Publishing for guiding and managing this project. Thanks to my colleague Fred Schmitt for several useful suggestions. Once again, I'd like to thank Ron Liechty of Metrowerks for his helpfulness.

## Acknowledgments for the Fourth Edition

Several editors from Pearson and from Sams helped originate and maintain this project; thanks to Linda Sharp, Karen Wachs, and Laurie McGuire. Thanks, too, to Michael Maddox, Bill Craun, Chris Maunder, and Phillippe Bruno for providing technical review and editing. And thanks again to Michael Maddox and Bill Craun for supplying the material for the Real World Notes. Finally, I'd like to thank Ron Liechty of Metrowerks and Greg Comeau of Comeau Computing for their aid with C++ compilers.

## Acknowledgments for the Third Edition

I'd like to thank the editors from Macmillan and The Waite Group for the roles they played in putting this book together: Tracy Dunkelberger, Susan Walton, and Andrea Rosenberg. Thanks, too, to Russ Jacobs for his content and technical editing. From Metrowerks, I'd like to thank Dave Mark, Alex Harper, and especially Ron Liechty, for their help and cooperation.

## Acknowledgments for the Second Edition

I'd like to thank Mitchell Waite and Scott Calamar for supporting a second edition and Joel Fugazzotto and Joanne Miller for guiding the project to completion. Thanks to Michael Marcotty of Metrowerks for dealing with my questions about their beta version CodeWarrior compiler. I'd also like to thank the following instructors for taking the time to give us feedback on the first edition: Jeff Buckwalter, Earl Brynner, Mike Holland, Andy Yao, Larry Sanders, Shahin Momtazi, and Don Stephens. Finally, I wish to thank Heidi Brumbaugh for her helpful content editing of new and revised material.

## Acknowledgments for the First Edition

Many people have contributed to this book. In particular, I wish to thank Mitch Waite for his work in developing, shaping, and reshaping this book, and for reviewing the manuscript. I appreciate Harry Henderson's work in reviewing the last few chapters and in

testing programs with the Zortech C++ compiler. Thanks to David Gerrold for reviewing the entire manuscript and for championing the needs of less-experienced readers. Also thanks to Hank Shiffman for testing programs using Sun C++ and to Kent Williams for testing programs with AT&T cfront and with G++. Thanks to Nan Borreson of Borland International for her responsive and cheerful assistance with Turbo C++ and Borland C++. Thank you, Ruth Myers and Christine Bush, for handling the relentless paper flow involved with this kind of project. Finally, thanks to Scott Calamar for keeping everything on track.

## About the Author

**Stephen Prata** taught astronomy, physics, and computer science at the College of Marin in Kentfield, California. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. He has authored or coauthored more than a dozen books on programming topics including *New C Primer Plus*, which received the Computer Press Association's 1990 Best How-to Computer Book Award, and *C++ Primer Plus*, nominated for the Computer Press Association's Best How-to Computer Book Award in 1991.

## We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number.

Email:        feedback@developers-library.info  
Mail:         Reader Feedback  
               Addison-Wesley Developer's Library  
               800 East 96th Street  
               Indianapolis, IN 46240 USA

## Reader Services

Visit our website and register this book at [www.informit.com/register](http://www.informit.com/register) for convenient access to any updates, downloads, or errata that might be available for this book.

# Introduction

Learning C++ is an adventure of discovery, particularly because the language accommodates several programming paradigms, including object-oriented programming, generic programming, and the traditional procedural programming. The fifth edition of this book described the language as set forth in the ISO C++ standards, informally known as C++99 and C++03, or, sometimes as C++99/03. (The 2003 version was largely a technical correction to the 1999 standard and didn't add any new features.) Since then, C++ continues to evolve. As this book is written, the international C++ Standards Committee has just approved a new version of the standard. This standard had the informal name of C++0x while in development, and now it will be known as C++11. Most contemporary compilers support C++99/03 quite well, and most of the examples in this book comply with that standard. But many features of the new standard already have appeared in some implementations, and this edition of *C++ Primer Plus* explores these new features.

*C++ Primer Plus* discusses the basic C language and presents C++ features, making this book self-contained. It presents C++ fundamentals and illustrates them with short, to-the-point programs that are easy to copy and experiment with. You learn about input/output (I/O), how to make programs perform repetitive tasks and make choices, the many ways to handle data, and how to use functions. You learn about the many features C++ has added to C, including the following:

- Classes and objects
- Inheritance
- Polymorphism, virtual functions, and runtime type identification (RTTI)
- Function overloading
- Reference variables
- Generic, or type-independent, programming, as provided by templates and the Standard Template Library (STL)
- The exception mechanism for handling error conditions
- Namespaces for managing names of functions, classes, and variables



## The Primer Approach

*C++ Primer Plus* brings several virtues to the task of presenting all this material. It builds on the primer tradition begun by *C Primer Plus* nearly two decades ago and embraces its successful philosophy:

- A primer should be an easy-to-use, friendly guide.
- A primer doesn't assume that you are already familiar with all relevant programming concepts.
- A primer emphasizes hands-on learning with brief, easily typed examples that develop your understanding, a concept or two at a time.
- A primer clarifies concepts with illustrations.
- A primer provides questions and exercises to let you test your understanding, making the book suitable for self-learning or for the classroom.

Following these principles, the book helps you understand this rich language and how to use it. For example

- It provides conceptual guidance about when to use particular features, such as using public inheritance to model what are known as *is-a* relationships.
- It illustrates common C++ programming idioms and techniques.
- It provides a variety of sidebars, including tips, cautions, things to remember, compatibility notes, and real-world notes.

The author and editors of this book do our best to keep the presentation to-the-point, simple, and fun. Our goal is that by the end of the book, you'll be able to write solid, effective programs and enjoy yourself doing so.

## Sample Code Used in This Book

This book provides an abundance of sample code, most of it in the form of complete programs. Like the previous editions, this book practices generic C++ so that it is not tied to any particular kind of computer, operating system, or compiler. Thus, the examples were tested on a Windows 7 system, a Macintosh OS X system, and a Linux system. Those programs using C++11 features require compilers supporting those features, but the remaining programs should work with any C++99/03-compliant system.

The sample code for the complete programs described in this book is available on this book's website. See the registration link given on the back cover for more information.

## How This Book Is Organized

This book is divided into 18 chapters and 10 appendixes, summarized here:

- **Chapter 1: Getting Started with C++**—Chapter 1 relates how Bjarne Stroustrup created the C++ programming language by adding object-oriented programming

support to the C language. You'll learn the distinctions between procedural languages, such as C, and object-oriented languages, such as C++. You'll read about the joint ANSI/ISO work to develop a C++ standard. This chapter discusses the mechanics of creating a C++ program, outlining the approach for several current C++ compilers. Finally, it describes the conventions used in this book.

- **Chapter 2: Setting Out to C++**—Chapter 2 guides you through the process of creating simple C++ programs. You'll learn about the role of the `main()` function and about some of the kinds of statements that C++ programs use. You'll use the predefined `cout` and `cin` objects for program output and input, and you'll learn about creating and using variables. Finally, you'll be introduced to functions, C++'s programming modules.
- **Chapter 3: Dealing with Data**—C++ provides built-in types for storing two kinds of data: integers (numbers with no fractional parts) and floating-point numbers (numbers with fractional parts). To meet the diverse requirements of programmers, C++ offers several types in each category. Chapter 3 discusses those types, including creating variables and writing constants of various types. You'll also learn how C++ handles implicit and explicit conversions from one type to another.
- **Chapter 4: Compound Types**—C++ lets you construct more elaborate types from the basic built-in types. The most advanced form is the class, discussed in Chapters 9 through 13. Chapter 4 discusses other forms, including arrays, which hold several values of a single type; structures, which hold several values of unlike types; and pointers, which identify locations in memory. You'll also learn how to create and store text strings and to handle text I/O by using C-style character arrays and the C++ `string` class. Finally, you'll learn some of the ways C++ handles memory allocation, including using the `new` and `delete` operators for managing memory explicitly.
- **Chapter 5: Loops and Relational Expressions**—Programs often must perform repetitive actions, and C++ provides three looping structures for that purpose: the `for` loop, the `while` loop, and the `do while` loop. Such loops must know when they should terminate, and the C++ relational operators enable you to create tests to guide such loops. In Chapter 5 you learn how to create loops that read and process input character-by-character. Finally, you'll learn how to create two-dimensional arrays and how to use nested loops to process them.
- **Chapter 6: Branching Statements and Logical Operators**—Programs can behave intelligently if they can tailor their behavior to circumstances. In Chapter 6 you'll learn how to control program flow by using the `if`, `if else`, and `switch` statements and the conditional operator. You'll learn how to use logical operators to help express decision-making tests. Also, you'll meet the `cctype` library of functions for evaluating character relations, such as testing whether a character is a digit or a nonprinting character. Finally, you'll get an introductory view of file I/O.

- **Chapter 7: Functions: C++'s Programming Modules**—Functions are the basic building blocks of C++ programming. Chapter 7 concentrates on features that C++ functions share with C functions. In particular, you'll review the general format of a function definition and examine how function prototypes increase the reliability of programs. Also, you'll investigate how to write functions to process arrays, character strings, and structures. Next, you'll learn about recursion, which is when a function calls itself, and see how it can be used to implement a divide-and-conquer strategy. Finally, you'll meet pointers to functions, which enable you to use a function argument to tell one function to use a second function.
- **Chapter 8: Adventures in Functions**—Chapter 8 explores the new features C++ adds to functions. You'll learn about inline functions, which can speed program execution at the cost of additional program size. You'll work with reference variables, which provide an alternative way to pass information to functions. Default arguments let a function automatically supply values for function arguments that you omit from a function call. Function overloading lets you create functions having the same name but taking different argument lists. All these features have frequent use in class design. Also you'll learn about function templates, which allow you to specify the design of a family of related functions.
- **Chapter 9: Memory Models and Namespaces**—Chapter 9 discusses putting together multifile programs. It examines the choices in allocating memory, looking at different methods of managing memory and at scope, linkage, and namespaces, which determine what parts of a program know about a variable.
- **Chapter 10: Objects and Classes**—A class is a user-defined type, and an object (such as a variable) is an instance of a class. Chapter 10 introduces you to object-oriented programming and to class design. A class declaration describes the information stored in a class object and also the operations (class methods) allowed for class objects. Some parts of an object are visible to the outside world (the public portion), and some are hidden (the private portion). Special class methods (constructors and destructors) come into play when objects are created and destroyed. You will learn about all this and other class details in this chapter, and you'll see how classes can be used to implement ADTs, such as a stack.
- **Chapter 11: Working with Classes**—In Chapter 11 you'll further your understanding of classes. First, you'll learn about operator overloading, which lets you define how operators such as + will work with class objects. You'll learn about friend functions, which can access class data that's inaccessible to the world at large. You'll see how certain constructors and overloaded operator member functions can be used to manage conversion to and from class types.
- **Chapter 12: Classes and Dynamic Memory Allocation**—Often it's useful to have a class member point to dynamically allocated memory. If you use `new` in a class constructor to allocate dynamic memory, you incur the responsibilities of providing an appropriate destructor, of defining an explicit copy constructor, and of

defining an explicit assignment operator. Chapter 12 shows you how and discusses the behavior of the member functions generated implicitly if you fail to provide explicit definitions. You'll also expand your experience with classes by using pointers to objects and studying a queue simulation problem.

- **Chapter 13: Class Inheritance**—One of the most powerful features of object-oriented programming is inheritance, by which a derived class inherits the features of a base class, enabling you to reuse the base class code. Chapter 13 discusses public inheritance, which models *is-a* relationships, meaning that a derived object is a special case of a base object. For example, a physicist is a special case of a scientist. Some inheritance relationships are polymorphic, meaning you can write code using a mixture of related classes for which the same method name may invoke behavior that depends on the object type. Implementing this kind of behavior necessitates using a new kind of member function called a virtual function. Sometimes using abstract base classes is the best approach to inheritance relationships. This chapter discusses these matters, pointing out when public inheritance is appropriate and when it is not.
- **Chapter 14: Reusing Code in C++**—Public inheritance is just one way to reuse code. Chapter 14 looks at several other ways. Containment is when one class contains members that are objects of another class. It can be used to model *has-a* relationships, in which one class has components of another class. For example, an automobile has a motor. You also can use private and protected inheritance to model such relationships. This chapter shows you how and points out the differences among the different approaches. Also, you'll learn about class templates, which let you define a class in terms of some unspecified generic type, and then use the template to create specific classes in terms of specific types. For example, a stack template enables you to create a stack of integers or a stack of strings. Finally, you'll learn about multiple public inheritance, whereby a class can derive from more than one class.
- **Chapter 15: Friends, Exceptions, and More**—Chapter 15 extends the discussion of friends to include friend classes and friend member functions. Then it presents several new developments in C++, beginning with exceptions, which provide a mechanism for dealing with unusual program occurrences, such as inappropriate function argument values and running out of memory. Then you'll learn about RTTI, a mechanism for identifying object types. Finally, you'll learn about the safer alternatives to unrestricted typecasting.
- **Chapter 16: The `string` Class and the Standard Template Library**—Chapter 16 discusses some useful class libraries recently added to the language. The `string` class is a convenient and powerful alternative to traditional C-style strings. The `auto_ptr` class helps manage dynamically allocated memory. The STL provides several generic containers, including template representations of arrays, queues, lists, sets, and maps. It also provides an efficient library of generic algorithms that can be used with STL

containers and also with ordinary arrays. The `valarray` template class provides support for numeric arrays.

- **Chapter 17: Input, Output, and Files**—Chapter 17 reviews C++ I/O and discusses how to format output. You'll learn how to use class methods to determine the state of an input or output stream and to see, for example, whether there has been a type mismatch on input or whether the end-of-file has been detected. C++ uses inheritance to derive classes for managing file input and output. You'll learn how to open files for input and output, how to append data to a file, how to use binary files, and how to get random access to a file. Finally, you'll learn how to apply standard I/O methods to read from and write to strings.
- **Chapter 18: Visiting with the New C++ Standard**—Chapter 18 begins by reviewing several C++11 features introduced in earlier chapters, including new types, uniform initialization syntax, automatic type deduction, new smart pointers, and scoped enumerations. The chapter then discusses the new `rvalue` reference type and how it's used to implement a new feature called *move semantics*. Next, the chapter covers new class features, lambda expressions, and variadic templates. Finally, the chapter outlines many new features not covered in earlier chapters of the book.
- **Appendix A: Number Bases**—Appendix A discusses octal, hexadecimal, and binary numbers.
- **Appendix B: C++ Reserved Words**—Appendix B lists C++ keywords.
- **Appendix C: The ASCII Character Set**—Appendix C lists the ASCII character set, along with decimal, octal, hexadecimal, and binary representations.
- **Appendix D: Operator Precedence**—Appendix D lists the C++ operators in order of decreasing precedence.
- **Appendix E: Other Operators**—Appendix E summarizes the C++ operators, such as the bitwise operators, not covered in the main body of the text.
- **Appendix F: The `string` Template Class**—Appendix F summarizes `string` class methods and functions.
- **Appendix G: The Standard Template Library Methods and Functions**—Appendix G summarizes the STL container methods and the general STL algorithm functions.
- **Appendix H: Selected Readings and Internet Resources**—Appendix H lists some books that can further your understanding of C++.
- **Appendix I: Converting to ISO Standard C++**—Appendix I provides guidelines for moving from C and older C++ implementations to ANSI/ISO C++.
- **Appendix J: Answers to Chapter Review**—Appendix J contains the answers to the review questions posed at the end of each chapter.

## Note to Instructors

One of the goals of this edition of *C++ Primer Plus* is to provide a book that can be used as either a teach-yourself book or as a textbook. Here are some of the features that support using *C++ Primer Plus*, Sixth Edition, as a textbook:

- This book describes generic C++, so it isn't dependent on a particular implementation.
- The contents track the ISO/ANSI C++ standards committee's work and include discussions of templates, the STL, the `string` class, exceptions, RTTI, and namespaces.
- It doesn't assume prior knowledge of C, so it can be used without a C prerequisite. (Some programming background is desirable, however.)
- Topics are arranged so that the early chapters can be covered rapidly as review chapters for courses that do have a C prerequisite.
- Chapters include review questions and programming exercises. Appendix J provides the answers to the review questions.
- The book introduces several topics that are appropriate for computer science courses, including abstract data types (ADTs), stacks, queues, simple lists, simulations, generic programming, and using recursion to implement a divide-and-conquer strategy.
- Most chapters are short enough to cover in a week or less.
- The book discusses *when* to use certain features as well as *how* to use them. For example, it links public inheritance to *is-a* relationships and composition and private inheritance to *has-a* relationships, and it discusses when to use virtual functions and when not to.

## Conventions Used in This Book

This book uses several typographic conventions to distinguish among various kinds of text:

- Code lines, commands, statements, variables, filenames, and program output appear in a computer typeface:

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "What's up, Doc!\n";
    return 0;
}
```

- Program input that you should type appears in **bold computer typeface**:

Please enter your name:

**Plato**

- Placeholders in syntax descriptions appear in an *italic computer typeface*. You should replace a placeholder with the actual filename, parameter, or whatever element it represents.
- *Italic type* is used for new terms.

### Sidebar

A sidebar provides a deeper discussion or additional background to help illuminate a topic.

### Tip

Tips present short, helpful guides to particular programming situations.

### Caution

A caution alerts you to potential pitfalls.

### Note

The notes provide a catch-all category for comments that don't fall into one of the other categories.

## Systems Used to Develop This Book's Programming Examples

For the record, the C++11 examples in this book were developed using Microsoft Visual C++ 2010 and Cygwin with Gnu g++ 4.5.0, both running under 64-bit Windows 7. The remaining examples were tested with these systems, as well as on an iMac using g++ 4.2.1 under OS X 10.6.8 and on an Ubuntu Linux system using g++ 4.4.1. Most of the pre-C++11 examples were originally developed using Microsoft Visual C++ 2003 and Metrowerks CodeWarrior Development Studio 9 running under Windows XP Professional and checked using the Borland C++ 5.5 command-line compiler and GNU gpp 3.3.3 on the same system, using Comeau 4.3.3 and GNU g++ 3.3.1 under SuSE 9.0 Linux, and using Metrowerks Development Studio 9 on a Macintosh G4 under OS 10.3.

C++ offers a lot to the programmer; learn and enjoy!

# Setting Out to C++

In this chapter you'll learn about the following:

- Creating a C++ program
- The general format for a C++ program
- The `#include` directive
- The `main()` function
- Using the `cout` object for output
- Placing comments in a C++ program
- How and when to use `endl`
- Declaring and using variables
- Using the `cin` object for input
- Defining and using simple functions

When you construct a simple home, you begin with the foundation and the framework. If you don't have a solid structure from the beginning, you'll have trouble later filling in the details, such as windows, door frames, observatory domes, and parquet ballrooms. Similarly, when you learn a computer language, you should begin by learning the basic structure for a program. Only then can you move on to the details, such as loops and objects. This chapter gives you an overview of the essential structure of a C++ program and previews some topics—notably functions and classes—covered in much greater detail in later chapters. (The idea is to introduce at least some of the basic concepts gradually en route to the great awakenings that come later.)

## C++ Initiation

Let's begin with a simple C++ program that displays a message. Listing 2.1 uses the C++ `cout` (pronounced “see-out”) facility to produce character output. The source code includes several comments to the reader; these lines begin with `//`, and the compiler ignores them. C++ is *case sensitive*; that is, it discriminates between uppercase characters



and lowercase characters. This means you must be careful to use the same case as in the examples. For example, this program uses `cout`, and if you substitute `Count` or `COUT`, the compiler rejects your offering and accuses you of using unknown identifiers. (The compiler is also spelling sensitive, so don't try `kout` or `coot`, either.) The `cpp` filename extension is a common way to indicate a C++ program; you might need to use a different extension, as described in Chapter 1, "Getting Started with C++."

### Listing 2.1 `myfirst.cpp`

---

```
// myfirst.cpp -- displays a message

#include <iostream>                // a PREPROCESSOR directive
int main()                        // function header
{                                  // start of function body
    using namespace std;         // make definitions visible
    cout << "Come up and C++ me some time."; // message
    cout << endl;                // start a new line
    cout << "You won't regret it!" << endl; // more output
    return 0;                    // terminate main()
}                                  // end of function body
```

---

### Program Adjustments

You might find that you must alter the examples in this book to run on your system. The most common reason is a matter of the programming environment. Some windowing environments run the program in a separate window and then automatically close the window when the program finishes. As discussed in Chapter 1, you can make the window stay open until you strike a key by adding the following line of code before the return statement:

```
cin.get();
```

For some programs you must add two of these lines to keep the window open until you press a key. You'll learn more about `cin.get()` in Chapter 4, "Compound Types."

If you have a very old system, it may not support features introduced by the C++98 standard.

Some programs require a compiler with some level of support for the C++11 standard. They will be clearly identified and, if possible, alternative non-C++11 code will be suggested.

After you use your editor of choice to copy this program (or else use the source code files available online from this book's web page—check the registration link on the back cover for more information), you can use your C++ compiler to create the executable code, as Chapter 1 outlines. Here is the output from running the compiled program in Listing 2.1:

```
Come up and C++ me some time.
You won't regret it!
```

## C Input and Output

If you're used to programming in C, seeing `cout` instead of the `printf()` function might come as a minor shock. C++ can, in fact, use `printf()`, `scanf()`, and all the other standard C input and output functions, provided that you include the usual C `stdio.h` file. But this is a C++ book, so it uses C++'s input facilities, which improve in many ways upon the C versions.

You construct C++ programs from building blocks called *functions*. Typically, you organize a program into major tasks and then design separate functions to handle those tasks. The example shown in Listing 2.1 is simple enough to consist of a single function named `main()`. The `myfirst.cpp` example has the following elements:

- Comments, indicated by the `//` prefix
- A preprocessor `#include` directive
- A function header: `int main()`
- A `using namespace` directive
- A function body, delimited by `{` and `}`
- Statements that uses the C++ `cout` facility to display a message
- A return statement to terminate the `main()` function

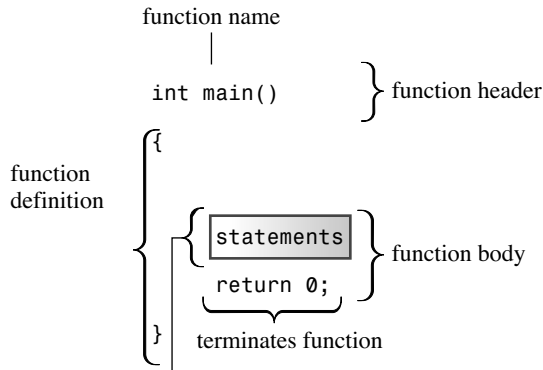
Let's look at these various elements in greater detail. The `main()` function is a good place to start because some of the features that precede `main()`, such as the preprocessor directive, are simpler to understand after you see what `main()` does.

## Features of the `main()` Function

Stripped of the trimmings, the sample program shown in Listing 2.1 has the following fundamental structure:

```
int main()
{
    statements
    return 0;
}
```

These lines state that there is a function called `main()`, and they describe how the function behaves. Together they constitute a *function definition*. This definition has two parts: the first line, `int main()`, which is called the *function header*, and the portion enclosed in braces (`{` and `}`), which is the *function body*. (A quick search on the Web reveals braces also go by other names, including “curly brackets,” “flower brackets,” “fancy brackets,” and “chicken lips.” However, the ISO Standard uses the term “braces.”) Figure 2.1 shows the `main()` function. The function header is a capsule summary of the function's interface with the rest of the program, and the function body represents instructions to the computer about what the function should do. In C++ each complete instruction is called a *statement*. You must terminate each statement with a semicolon, so don't omit the semicolons when you type the examples.



Statements are C++ expressions terminated by a semicolon.

Figure 2.1 The `main()` function.

The final statement in `main()`, called a *return statement*, terminates the function. You'll learn more about the return statement as you read through this chapter.

### Statements and Semicolons

A statement represents an action to be taken. To understand your source code, a compiler needs to know when one statement ends and another begins. Some languages use a statement separator. FORTRAN, for example, uses the end of the line to separate one statement from the next. Pascal uses a semicolon to separate one statement from the next. In Pascal you can omit the semicolon in certain cases, such as after a statement just before an `END`, when you aren't actually separating two statements. (Pragmatists and minimalists will disagree about whether *can* implies *should*.) But C++, like C, uses a semicolon as a *terminator* rather than as a separator. The difference is that a semicolon acting as a terminator is *part* of the statement rather than a marker *between* statements. The practical upshot is that in C++ you should never omit the semicolon.

### The Function Header as an Interface

Right now the main point to remember is that C++ syntax requires you to begin the definition of the `main()` function with this header: `int main()`. This chapter discusses the function header syntax in more detail later, in the section "Functions," but for those who can't put their curiosity on hold, here's a preview.

In general, a C++ function is activated, or *called*, by another function, and the function header describes the interface between a function and the function that calls it. The part preceding the function name is called the *function return type*; it describes information flow from a function back to the function that calls it. The part within the parentheses following the function name is called the *argument list* or *parameter list*; it describes information flow from the calling function to the called function. This general description is a bit confusing when you apply it to `main()` because you normally don't call `main()` from other parts of your program. Typically, however, `main()` is called by startup code that the compiler adds to your program to mediate between the program and the operating system

(Unix, Windows 7, Linux, or whatever). In effect, the function header describes the interface between `main()` and the operating system.

Consider the interface description for `main()`, beginning with the `int` part. A C++ function called by another function can return a value to the activating (calling) function. That value is called a *return value*. In this case, `main()` can return an integer value, as indicated by the keyword `int`. Next, note the empty parentheses. In general, a C++ function can pass information to another function when it calls that function. The portion of the function header enclosed in parentheses describes that information. In this case, the empty parentheses mean that the `main()` function takes no information, or in the usual terminology, `main()` takes no arguments. (To say that `main()` takes no arguments doesn't mean that `main()` is an unreasonable, authoritarian function. Instead, *argument* is the term computer buffs use to refer to information passed from one function to another.)

In short, the following function header states that the `main()` function returns an integer value to the function that calls it and that `main()` takes no information from the function that calls it:

```
int main()
```

Many existing programs use the classic C function header instead:

```
main()    // original C style
```

Under classic C, omitting the return type is the same as saying that the function is type `int`. However, C++ has phased out that usage.

You can also use this variant:

```
int main(void)    // very explicit style
```

Using the keyword `void` in the parentheses is an explicit way of saying that the function takes no arguments. Under C++ (but not C), leaving the parentheses empty is the same as using `void` in the parentheses. (In C, leaving the parentheses empty means you are remaining silent about whether there are arguments.)

Some programmers use this header and omit the return statement:

```
void main()
```

This is logically consistent because a `void` return type means the function doesn't return a value. However, although this variant works on some systems, it's not part of the C++ Standard. Thus, on other systems it fails. So you should avoid this form and use the C++ Standard form; it doesn't require that much more effort to do it right.

Finally, the ISO C++ Standard makes a concession to those who complain about the tiresome necessity of having to place a return statement at the end of `main()`. If the compiler reaches the end of `main()` without encountering a return statement, the effect will be the same as if you ended `main()` with this statement:

```
return 0;
```

This implicit return is provided only for `main()` and not for any other function.

### Why `main()` by Any Other Name Is Not the Same

There's an extremely compelling reason to name the function in the `myfirst.cpp` program `main()`: You must do so. Ordinarily, a C++ program requires a function called `main()`. (And not, by the way, `Main()` or `MAIN()` or `mane()`. Remember, case and spelling count.) Because the `myfirst.cpp` program has only one function, that function must bear the responsibility of being `main()`. When you run a C++ program, execution always begins at the beginning of the `main()` function. Therefore, if you don't have `main()`, you don't have a complete program, and the compiler points out that you haven't defined a `main()` function.

There are exceptions. For example, in Windows programming you can write a dynamic link library (DLL) module. This is code that other Windows programs can use. Because a DLL module is not a standalone program, it doesn't need a `main()`. Programs for specialized environments, such as for a controller chip in a robot, might not need a `main()`. Some programming environments provide a skeleton program calling some nonstandard function, such as `_tmain()`; in that case there is a hidden `main()` that calls `_tmain()`. But your ordinary standalone program does need a `main()`; this book discusses that sort of program.

### C++ Comments

The double slash (`//`) introduces a C++ comment. A *comment* is a remark from the programmer to the reader that usually identifies a section of a program or explains some aspect of the code. The compiler ignores comments. After all, it knows C++ at least as well as you do, and, in any case, it's incapable of understanding comments. As far as the compiler is concerned, Listing 2.1 looks as if it were written without comments, like this:

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "Come up and C++ me some time.";
    cout << endl;
    cout << "You won't regret it!" << endl;
    return 0;
}
```

C++ comments run from the `//` to the end of the line. A comment can be on its own line, or it can be on the same line as code. Incidentally, note the first line in Listing 2.1:

```
// myfirst.cpp -- displays a message
```

In this book all programs begin with a comment that gives the filename for the source code and a brief program summary. As mentioned in Chapter 1, the filename extension for source code depends on your C++ system. Other systems might use `myfirst.C` or `myfirst.cxx` for names.

## Tip

You should use comments to document your programs. The more complex the program, the more valuable comments are. Not only do they help others to understand what you have done, but also they help you understand what you've done, especially if you haven't looked at the program for a while.

## C-Style Comments

C++ also recognizes C comments, which are enclosed between `/*` and `*/` symbols:

```
#include <iostream> /* a C-style comment */
```

Because the C-style comment is terminated by `*/` rather than by the end of a line, you can spread it over more than one line. You can use either or both styles in your programs. However, try sticking to the C++ style. Because it doesn't involve remembering to correctly pair an end symbol with a begin symbol, it's less likely to cause problems. Indeed, C99 has added the `//` comment to the C language.

## The C++ Preprocessor and the `iostream` File

Here's the short version of what you need to know. If your program is to use the usual C++ input or output facilities, you provide these two lines:

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

There are some alternatives to using the second line, but let's keep things simple for now. (If your compiler doesn't like these lines, it's not C++98 compatible, and it will have many other problems with the examples in this book.) That's all you really must know to make your programs work, but now let's take a more in-depth look.

C++, like C, uses a *preprocessor*. This is a program that processes a source file before the main compilation takes place. (Some C++ implementations, as you might recall from Chapter 1, use a translator program to convert a C++ program to C. Although the translator is also a form of preprocessor, we're not discussing that preprocessor; instead, we're discussing the one that handles directives whose names begin with `#`.) You don't have to do anything special to invoke this preprocessor. It automatically operates when you compile the program.

Listing 2.1 uses the `#include` directive:

```
#include <iostream>    // a PREPROCESSOR directive
```

This directive causes the preprocessor to add the contents of the `iostream` file to your program. This is a typical preprocessor action: adding or replacing text in the source code before it's compiled.

This raises the question of why you should add the contents of the `iostream` file to the program. The answer concerns communication between the program and the outside world. The `io` in `iostream` refers to *input*, which is information brought into the program, and to *output*, which is information sent out from the program. C++'s input/output scheme involves several definitions found in the `iostream` file. Your first program needs

these definitions to use the `cout` facility to display a message. The `#include` directive causes the contents of the `iostream` file to be sent along with the contents of your file to the compiler. In essence, the contents of the `iostream` file replace the `#include <iostream>` line in the program. Your original file is not altered, but a composite file formed from your file and `iostream` goes on to the next stage of compilation.

### Note

Programs that use `cin` and `cout` for input and output must include the `iostream` file.

## Header Filenames

Files such as `iostream` are called *include files* (because they are included in other files) or *header files* (because they are included at the beginning of a file). C++ compilers come with many header files, each supporting a particular family of facilities. The C tradition has been to use the `.h` extension with header files as a simple way to identify the type of file by its name. For example, the C `math.h` header file supports various C math functions. Initially, C++ did the same. For instance, the header file supporting input and output was named `iostream.h`. But C++ usage has changed. Now the `.h` extension is reserved for the old C header files (which C++ programs can still use), whereas C++ header files have no extension. There are also C header files that have been converted to C++ header files. These files have been renamed by dropping the `.h` extension (making it a C++-style name) and prefixing the filename with a `c` (indicating that it comes from C). For example, the C++ version of `math.h` is the `cmath` header file. Sometimes the C and C++ versions of C header files are identical, whereas in other cases the new version might have a few changes. For purely C++ header files such as `iostream`, dropping the `.h` is more than a cosmetic change, for the `.h`-free header files also incorporate namespaces, the next topic in this chapter. Table 2.1 summarizes the naming conventions for header files.

Table 2.1 Header File Naming Conventions

Kind of Header	Convention	Example	Comments
C++ old style	Ends in <code>.h</code>	<code>iostream.h</code>	Usable by C++ programs
C old style	Ends in <code>.h</code>	<code>math.h</code>	Usable by C and C++ programs
C++ new style	No extension	<code>iostream</code>	Usable by C++ programs, uses namespace <code>std</code>
Converted C	<code>c</code> prefix, no extension	<code>cmath</code>	Usable by C++ programs, might use non-C features, such as namespace <code>std</code>

In view of the C tradition of using different filename extensions to indicate different file types, it appears reasonable to have some special extension, such as `.hpp` or `.hxx`, to indicate C++ header files. The ANSI/ISO committee felt so, too. The problem was agreeing on which extension to use, so eventually they agreed on nothing.

## Namespaces

If you use `iostream` instead of `iostream.h`, you should use the following namespace directive to make the definitions in `iostream` available to your program:

```
using namespace std;
```

This is called a *using directive*. The simplest thing to do is to accept this for now and worry about it later (for example, in Chapter 9, “Memory Models and Namespaces”). But so you won’t be left completely in the dark, here’s an overview of what’s happening.

Namespace support is a C++ feature designed to simplify the writing of large programs and of programs that combine pre-existing code from several vendors and to help organize programs. One potential problem is that you might use two prepackaged products that both have, say, a function called `wanda()`. If you then use the `wanda()` function, the compiler won’t know which version you mean. The namespace facility lets a vendor package its wares in a unit called a *namespace* so that you can use the name of a namespace to indicate which vendor’s product you want. So Microflop Industries could place its definitions in a namespace called `Microflop`. Then `Microflop::wanda()` would become the full name for its `wanda()` function. Similarly, `Piscine::wanda()` could denote Piscine Corporation’s version of `wanda()`. Thus, your program could now use the namespaces to discriminate between various versions:

```
Microflop::wanda("go dancing?"); // use Microflop namespace version
Piscine::wanda("a fish named Desire"); // use Piscine namespace version
```

In this spirit, the classes, functions, and variables that are a standard component of C++ compilers are now placed in a namespace called `std`. This takes place in the `h-free` header files. This means, for example, that the `cout` variable used for output and defined in `iostream` is really called `std::cout` and that `endl` is really `std::endl`. Thus, you can omit the `using` directive and, instead, code in the following style:

```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;
```

However, many users don’t feel like converting pre-namespace code, which uses `iostream.h` and `cout`, to namespace code, which uses `iostream` and `std::cout`, unless they can do so without a lot of hassle. This is where the `using` directive comes in. The following line means you can use names defined in the `std` namespace without using the `std::` prefix:

```
using namespace std;
```

This `using` directive makes all the names in the `std` namespace available. Modern practice regards this as a bit lazy and potentially a problem in large projects. The preferred approaches are to use the `std::` qualifier or to use something called a *using declaration* to make just particular names available:

```
using std::cout; // make cout available
using std::endl; // make endl available
using std::cin; // make cin available
```



If you use these directives instead of the following, you can use `cin` and `cout` without attaching `std::` to them:

```
using namespace std; // lazy approach, all names available
```

But if you need to use other names from `iostream`, you have to add them to the `using` list individually. This book initially uses the lazy approach for a couple reasons. First, for simple programs, it's not really a big issue which namespace management technique you use. Second, I'd rather emphasize the more basic aspects about learning C++. Later, the book uses the other namespace techniques.

## C++ Output with `cout`

Now let's look at how to display a message. The `myfirst.cpp` program uses the following C++ statement:

```
cout << "Come up and C++ me some time.";
```

The part enclosed within the double quotation marks is the message to print. In C++, any series of characters enclosed in double quotation marks is called a *character string*, presumably because it consists of several characters strung together into a larger unit. The `<<` notation indicates that the statement is sending the string to `cout`; the symbols point the way the information flows. And what is `cout`? It's a predefined object that knows how to display a variety of things, including strings, numbers, and individual characters. (An *object*, as you might remember from Chapter 1, is a particular instance of a class, and a *class* defines how data is stored and used.)

Well, using objects so soon is a bit awkward because you won't learn about objects for several more chapters. Actually, this reveals one of the strengths of objects. You don't have to know the innards of an object in order to use it. All you must know is its interface—that is, how to use it. The `cout` object has a simple interface. If `string` represents a string, you can do the following to display it:

```
cout << string;
```

This is all you must know to display a string, but now take a look at how the C++ conceptual view represents the process. In this view, the output is a stream—that is, a series of characters flowing from the program. The `cout` object, whose properties are defined in the `iostream` file, represents that stream. The object properties for `cout` include an insertion operator (`<<`) that inserts the information on its right into the stream. Consider the following statement (note the terminating semicolon):

```
cout << "Come up and C++ me some time.";
```

It inserts the string "Come up and C++ me some time." into the output stream. Thus, rather than say that your program displays a message, you can say that it inserts a string into the output stream. Somehow, that sounds more impressive (see Figure 2.2).

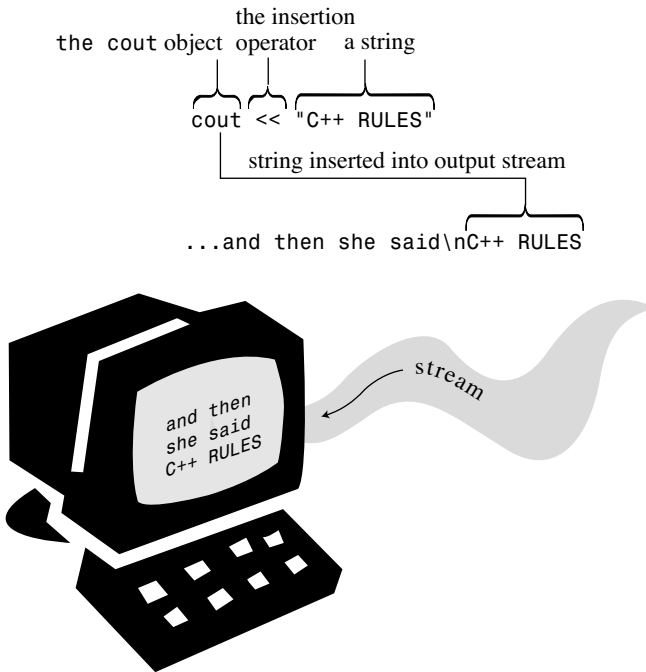


Figure 2.2 Using `cout` to display a string.

## A First Look at Operator Overloading

If you're coming to C++ from C, you probably noticed that the insertion operator (`<<`) looks just like the bitwise left-shift operator (`<<`). This is an example of *operator overloading*, by which the same operator symbol can have different meanings. The compiler uses the context to figure out which meaning is intended. C itself has some operator overloading. For example, the `&` symbol represents both the address operator and the bitwise `AND` operator. The `*` symbol represents both multiplication and dereferencing a pointer. The important point here is not the exact function of these operators but that the same symbol can have more than one meaning, with the compiler determining the proper meaning from the context. (You do much the same when you determine the meaning of "sound" in "sound card" versus "sound financial basis.") C++ extends the operator overloading concept by letting you redefine operator meanings for the user-defined types called classes.

## The Manipulator `endl`

Now let's examine an odd-looking notation that appears in the second output statement in Listing 2.1:

```
cout << endl;
```

`endl` is a special C++ notation that represents the important concept of beginning a new line. Inserting `endl` into the output stream causes the screen cursor to move to the beginning of the next line. Special notations like `endl` that have particular meanings to

`cout` are dubbed *manipulators*. Like `cout`, `endl` is defined in the `iostream` header file and is part of the `std` namespace.

Note that the `cout` facility does not move automatically to the next line when it prints a string, so the first `cout` statement in Listing 2.1 leaves the cursor positioned just after the period at the end of the output string. The output for each `cout` statement begins where the last output ended, so omitting `endl` would result in this output for Listing 2.1:

```
Come up and C++ me some time.You won't regret it!
```

Note that the `Y` immediately follows the period. Let's look at another example. Suppose you try this code:

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukulele";
cout << endl;
```

It produces the following output:

```
The Good, theBad, and the Ukulele
```

Again, note that the beginning of one string comes immediately after the end of the preceding string. If you want a space where two strings join, you must include it in one of the strings. (Remember that to try out these output examples, you have to place them in a complete program, with a `main()` function header and opening and closing braces.)

## The Newline Character

C++ has another, more ancient, way to indicate a new line in output—the C notation `\n`:

```
cout << "What's next?\n";    // \n means start a new line
```

The `\n` combination is considered to be a single character called the *newline* character.

If you are displaying a string, you need less typing to include the newline as part of the string than to tag an `endl` onto the end:

```
cout << "Pluto is a dwarf planet.\n";    // show text, go to next line
cout << "Pluto is a dwarf planet." << endl; // show text, go to next line
```

On the other hand, if you want to generate a newline by itself, both approaches take the same amount of typing, but most people find the keystrokes for `endl` to be more comfortable:

```
cout << "\n";    // start a new line
cout << endl;    // start a new line
```

Typically, this book uses an embedded newline character (`\n`) when displaying quoted strings and the `endl` manipulator otherwise. One difference is that `endl` guarantees the output will be *flushed* (in, this case, immediately displayed onscreen) before the program moves on. You don't get that guarantee with `\n`, which means that it is possible on some

systems in some circumstances a prompt might not be displayed until after you enter the information being prompted for.

The newline character is one example of special keystroke combinations termed “escape sequences”; they are further discussed in Chapter 3, “Dealing with Data.”

## C++ Source Code Formatting

Some languages, such as FORTRAN, are line-oriented, with one statement to a line. For these languages, the carriage return (generated by pressing the Enter key or the Return key) serves to separate statements. In C++, however, the semicolon marks the end of each statement. This leaves C++ free to treat the carriage return in the same way as a space or a tab. That is, in C++ you normally can use a space where you would use a carriage return and vice versa. This means you can spread a single statement over several lines or place several statements on one line. For example, you could reformat `myfirst.cpp` as follows:

```
#include <iostream>
    int
main
() {   using
      namespace
        std; cout
          <<
"Come up and C++ me some time."
;   cout <<
endl; cout <<
"You won't regret it!" <<
endl;return 0; }
```

This is visually ugly but valid code. You do have to observe some rules. In particular, in C and C++ you can't put a space, tab, or carriage return in the middle of an element such as a name, nor can you place a carriage return in the middle of a string. Here are examples of what you can't do:

```
int ma in()    // INVALID -- space in name
re
turn 0; // INVALID -- carriage return in word
cout << "Behold the Beans
of Beauty!"; // INVALID -- carriage return in string
```

(However, the *raw* string, added by C++11 and discussed briefly in Chapter 4, does allow including a carriage return in a string.)

## Tokens and White Space in Source Code

The indivisible elements in a line of code are called *tokens* (see Figure 2.3). Generally, you must separate one token from the next with a space, tab, or carriage return, which collectively are termed *white space*. Some single characters, such as parentheses and commas, are

tokens that need not be set off by white space. Here are some examples that illustrate when white space can be used and when it can be omitted:

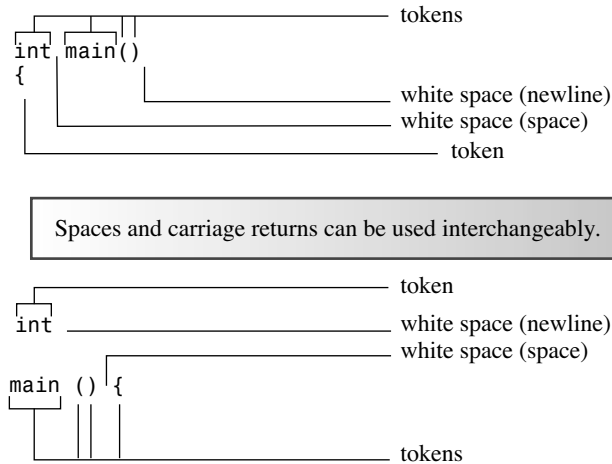


Figure 2.3 Tokens and white space.

```
return0;           // INVALID, must be return 0;
return(0);        // VALID, white space omitted
return (0);       // VALID, white space used
intmain();        // INVALID, white space omitted
int main()        // VALID, white space omitted in ( )
int main ( )     // ALSO VALID, white space used in ( )
```

### C++ Source Code Style

Although C++ gives you much formatting freedom, your programs will be easier to read if you follow a sensible style. Having valid but ugly code should leave you unsatisfied.

Most programmers use styles similar to that of Listing 2.1, which observes these rules:

- One statement per line
- An opening brace and a closing brace for a function, each of which is on its own line
- Statements in a function indented from the braces
- No whitespace around the parentheses associated with a function name

The first three rules have the simple intent of keeping the code clean and readable. The fourth helps to differentiate functions from some built-in C++ structures, such as loops, that also use parentheses. This book alerts you to other guidelines as they come up.

## C++ Statements

A C++ program is a collection of functions, and each function is a collection of statements. C++ has several kinds of statements, so let's look at some of the possibilities. Listing 2.2 provides two new kinds of statements. First, a *declaration statement* creates a variable. Second, an *assignment statement* provides a value for that variable. Also the program shows a new capability for `cout`.

Listing 2.2 `carrots.cpp`

---

```
// carrots.cpp -- food processing program
// uses and displays a variable

#include <iostream>

int main()
{
    using namespace std;

    int carrots;           // declare an integer variable

    carrots = 25;         // assign a value to the variable
    cout << "I have ";
    cout << carrots;      // display the value of the variable
    cout << " carrots.";
    cout << endl;
    carrots = carrots - 1; // modify the variable
    cout << "Crunch, crunch. Now I have " << carrots << " carrots." << endl;
    return 0;
}
```

---

A blank line separates the declaration from the rest of the program. This practice is the usual C convention, but it's somewhat less common in C++. Here is the program output for Listing 2.2:

```
I have 25 carrots.
Crunch, crunch. Now I have 24 carrots.
```

The next few pages examine this program.

### Declaration Statements and Variables

Computers are precise, orderly machines. To store an item of information in a computer, you must identify both the storage location and how much memory storage space the information requires. One relatively painless way to do this in C++ is to use a *declaration statement* to indicate the type of storage and to provide a label for the location. For example, the program in Listing 2.2 has this declaration statement (note the semicolon):

```
int carrots;
```

This statement provides two kinds of information: the type of memory storage needed and a label to attach to that storage. In particular, the statement declares that the program requires enough storage to hold an integer, for which C++ uses the label `int`. The compiler takes care of the details of allocating and labeling memory for that task. C++ can handle several kinds, or types, of data, and the `int` is the most basic data type. It corresponds to an integer, a number with no fractional part. The C++ `int` type can be positive or negative, but the size range depends on the implementation. Chapter 3 provides the details on `int` and the other basic types.

Naming the storage is the second task achieved. In this case, the declaration statement declares that henceforth the program will use the name `carrots` to identify the value stored at that location. `carrots` is called a *variable* because you can change its value. In C++ you must declare all variables. If you were to omit the declaration in `carrots.cpp`, the compiler would report an error when the program attempts to use `carrots` further on. (In fact, you might want to try omitting the declaration just to see how your compiler responds. Then if you see that response in the future, you'll know to check for omitted declarations.)

### Why Must Variables Be Declared?

Some languages, notably BASIC, create a new variable whenever you use a new name, without the aid of explicit declarations. That might seem friendlier to the user, and it is—in the short term. The problem is that if you misspell the name of a variable, you inadvertently can create a new variable without realizing it. That is, in BASIC, you can do something like the following:

```
CastleDark = 34
...
CastleDank = CastleDark + MoreGhosts
...
PRINT CastleDark
```

Because `CastleDank` is misspelled (the *r* was typed as an *n*), the changes you make to it leave `CastleDark` unchanged. This kind of error can be hard to trace because it breaks no rules in BASIC. However, in C++, `CastleDark` would be declared while the misspelled `CastleDank` would not be declared. Therefore, the equivalent C++ code breaks the rule about the need to declare a variable for you to use it, so the compiler catches the error and stomps the potential bug.

In general, then, a declaration indicates the type of data to be stored and the name the program will use for the data that's stored there. In this particular case, the program creates a variable called `carrots` in which it can store an integer (see Figure 2.4).

The declaration statement in the program is called a *defining declaration* statement, or *definition*, for short. This means that its presence causes the compiler to allocate memory space for the variable. In more complex situations, you can also have *reference declarations*. These tell the computer to use a variable that has already been defined elsewhere. In general, a declaration need not be a definition, but in this example it is.

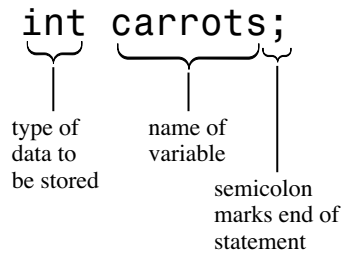


Figure 2.4 A variable declaration.

If you're familiar with C or Pascal, you're already familiar with variable declarations. You also might have a modest surprise in store for you. In C and Pascal, all variable declarations normally come at the very beginning of a function or procedure. But C++ has no such restriction. Indeed, the usual C++ style is to declare a variable just before it is first used. That way, you don't have to rummage back through a program to see what the type is. You'll see an example of this later in this chapter. This style does have the disadvantage of not gathering all your variable names in one place; thus, you can't tell at a glance what variables a function uses. (Incidentally, C99 now makes the rules for C declarations much the same as for C++.)

### Tip

The C++ style for declaring variables is to declare a variable as close to its first use as possible.

## Assignment Statements

An assignment statement assigns a value to a storage location. For example, the following statement assigns the integer 25 to the location represented by the variable `carrots`:

```
carrots = 25;
```

The `=` symbol is called the *assignment operator*. One unusual feature of C++ (and C) is that you can use the assignment operator serially. For example, the following is valid code:

```
int steinway;  
int baldwin;  
int yamaha;  
yamaha = baldwin = steinway = 88;
```

The assignment works from right to left. First, 88 is assigned to `steinway`; then the value of `steinway`, which is now 88, is assigned to `baldwin`; then `baldwin`'s value of 88 is assigned to `yamaha`. (C++ follows C's penchant for allowing weird-appearing code.)

The second assignment statement in Listing 2.2 demonstrates that you can change the value of a variable:

```
carrots = carrots - 1; // modify the variable
```



The expression to the right of the assignment operator (`carrots - 1`) is an example of an arithmetic expression. The computer will subtract 1 from 25, the value of `carrots`, obtaining 24. The assignment operator then stores this new value in the `carrots` location.

## A New Trick for `cout`

Up until now, the examples in this chapter have given `cout` strings to print. Listing 2.2 also gives `cout` a variable whose value is an integer:

```
cout << carrots;
```

The program doesn't print the word `carrots`; instead, it prints the integer value stored in `carrots`, which is 25. Actually, this is two tricks in one. First, `cout` replaces `carrots` with its current numeric value of 25. Second, it translates the value to the proper output characters.

As you can see, `cout` works with both strings and integers. This might not seem particularly remarkable to you, but keep in mind that the integer 25 is something quite different from the string "25". The string holds the characters with which you write the number (that is, a 2 character and a 5 character). The program internally stores the numeric codes for the 2 character and the 5 character. To print the string, `cout` simply prints each character in the string. But the integer 25 is stored as a numeric value. Rather than store each digit separately, the computer stores 25 as a binary number. (Appendix A, "Number Bases," discusses this representation.) The main point here is that `cout` must translate a number in integer form into character form before it can print it. Furthermore, `cout` is smart enough to recognize that `carrots` is an integer that requires conversion.

Perhaps the contrast with old C will indicate how clever `cout` is. To print the string "25" and the integer 25 in C, you could use C's multipurpose output function `printf()`:

```
printf("Printing a string: %s\n", "25");  
printf("Printing an integer: %d\n", 25);
```

Without going into the intricacies of `printf()`, note that you must use special codes (`%s` and `%d`) to indicate whether you are going to print a string or an integer. And if you tell `printf()` to print a string but give it an integer by mistake, `printf()` is too unsophisticated to notice your mistake. It just goes ahead and displays garbage.

The intelligent way in which `cout` behaves stems from C++'s object-oriented features. In essence, the C++ insertion operator (`<<`) adjusts its behavior to fit the type of data that follows it. This is an example of operator overloading. In later chapters, when you take up function overloading and operator overloading, you'll learn how to implement such smart designs yourself.

### `cout` and `printf()`

If you are used to C and `printf()`, you might think `cout` looks odd. You might even prefer to cling to your hard-won mastery of `printf()`. But `cout` actually is no stranger in appearance than `printf()`, with all its conversion specifications. More importantly, `cout` has significant advantages. Its capability to recognize types reflects a more intelligent and foolproof

design. Also, it is *extensible*. That is, you can redefine the `<<` operator so that `cout` can recognize and display new data types you develop. And if you relish the fine control `printf()` provides, you can accomplish the same effects with more advanced uses of `cout` (see Chapter 17, “Input, Output, and Files”).

## More C++ Statements

Let’s look at a couple more examples of statements. The program in Listing 2.3 expands on the preceding example by allowing you to enter a value while the program is running. To do so, it uses `cin` (pronounced “see-in”), the input counterpart to `cout`. Also the program shows yet another way to use that master of versatility, the `cout` object.

Listing 2.3 `getinfo.cpp`

---

```
// getinfo.cpp -- input and output
#include <iostream>

int main()
{
    using namespace std;

    int carrots;

    cout << "How many carrots do you have?" << endl;
    cin >> carrots;           // C++ input
    cout << "Here are two more. ";
    carrots = carrots + 2;
// the next line concatenates output
    cout << "Now you have " << carrots << " carrots." << endl;
    return 0;
}
```

---

### Program Adjustments

If you found that you had to add a `cin.get()` statement in the earlier listings, you will need to add two `cin.get()` statements to this listing to keep the program output visible onscreen. The first one will read the newline generated when you press the Enter or Return key after typing a number, and the second will cause the program to pause until you hit Return or Enter again.

Here is an example of output from the program in Listing 2.3:

```
How many carrots do you have?
12
Here are two more. Now you have 14 carrots.
```

The program has two new features: using `cin` to read keyboard input and combining four output statements into one. Let’s take a look.

## Using `cin`

As the output from Listing 2.3 demonstrates, the value typed from the keyboard (12) is eventually assigned to the variable `carrots`. The following statement performs that wonder:

```
cin >> carrots;
```

Looking at this statement, you can practically see information flowing from `cin` into `carrots`. Naturally, there is a slightly more formal description of this process. Just as C++ considers output to be a stream of characters flowing out of the program, it considers input to be a stream of characters flowing into the program. The `iostream` file defines `cin` as an object that represents this stream. For output, the `<<` operator inserts characters into the output stream. For input, `cin` uses the `>>` operator to extract characters from the input stream. Typically, you provide a variable to the right of the operator to receive the extracted information. (The symbols `<<` and `>>` were chosen to visually suggest the direction in which information flows.)

Like `cout`, `cin` is a smart object. It converts input, which is just a series of characters typed from the keyboard, into a form acceptable to the variable receiving the information. In this case, the program declares `carrots` to be an integer variable, so the input is converted to the numeric form the computer uses to store integers.

## Concatenating with `cout`

The second new feature of `getinfo.cpp` is combining four output statements into one. The `iostream` file defines the `<<` operator so that you can combine (that is, concatenate) output as follows:

```
cout << "Now you have " << carrots << " carrots." << endl;
```

This allows you to combine string output and integer output in a single statement. The resulting output is the same as what the following code produces:

```
cout << "Now you have ";
cout << carrots;
cout << " carrots";
cout << endl;
```

While you're still in the mood for `cout` advice, you can also rewrite the concatenated version this way, spreading the single statement over four lines:

```
cout << "Now you have "
     << carrots
     << " carrots."
     << endl;
```

That's because C++'s free format rules treat newlines and spaces between tokens interchangeably. This last technique is convenient when the line width cramps your style.

Another point to note is that

```
Now you have 14 carrots.
```

appears on the same line as

Here are two more.

That's because, as noted before, the output of one `cout` statement immediately follows the output of the preceding `cout` statement. This is true even if there are other statements in between.

## **cin and cout: A Touch of Class**

You've seen enough of `cin` and `cout` to justify your exposure to a little object lore. In particular, in this section you'll learn more about the notion of classes. As Chapter 1 outlined briefly, classes are one of the core concepts for object-oriented programming (OOP) in C++.

A *class* is a data type the user defines. To define a class, you describe what sort of information it can represent and what sort of actions you can perform with that data. A class bears the same relationship to an object that a type does to a variable. That is, a class definition describes a data form and how it can be used, whereas an object is an entity created according to the data form specification. Or, in noncomputer terms, if a class is analogous to a category such as famous actors, then an object is analogous to a particular example of that category, such as Kermit the Frog. To extend the analogy, a class representation of actors would include definitions of possible actions relating to the class, such as Reading for a Part, Expressing Sorrow, Projecting Menace, Accepting an Award, and the like. If you've been exposed to different OOP terminology, it might help to know that the C++ class corresponds to what some languages term an *object type*, and the C++ object corresponds to an object instance or instance variable.

Now let's get a little more specific. Recall the following declaration of a variable:

```
int carrots;
```

This creates a particular variable (`carrots`) that has the properties of the `int` type. That is, `carrots` can store an integer and can be used in particular ways—for addition and subtraction, for example. Now consider `cout`. It is an object created to have the properties of the `ostream` class. The `ostream` class definition (another inhabitant of the `iostream` file) describes the sort of data an `ostream` object represents and the operations you can perform with and to it, such as inserting a number or string into an output stream. Similarly, `cin` is an object created with the properties of the `istream` class, also defined in `iostream`.

### **Note**

The class describes all the properties of a data type, including actions that can be performed with it, and an object is an entity created according to that description.

You have learned that classes are user-defined types, but as a user, you certainly didn't design the `ostream` and `istream` classes. Just as functions can come in function libraries, classes can come in class libraries. That's the case for the `ostream` and `istream` classes. Technically, they are not built in to the C++ language; instead, they are examples of classes

that the language standard specifies. The class definitions are laid out in the `iostream` file and are not built into the compiler. You can even modify these class definitions if you like, although that's not a good idea. (More precisely, it is a truly dreadful idea.) The `iostream` family of classes and the related `fstream` (or file I/O) family are the only sets of class definitions that came with all early implementations of C++. However, the ANSI/ISO C++ committee added a few more class libraries to the Standard. Also most implementations provide additional class definitions as part of the package. Indeed, much of the current appeal of C++ is the existence of extensive and useful class libraries that support Unix, Macintosh, and Windows programming.

The class description specifies all the operations that can be performed on objects of that class. To perform such an allowed action on a particular object, you send a message to the object. For example, if you want the `cout` object to display a string, you send it a message that says, in effect, "Object! Display this!" C++ provides a couple ways to send messages. One way, using a class method, is essentially a function call like the ones you'll see soon. The other way, which is the one used with `cin` and `cout`, is to redefine an operator. Thus, the following statement uses the redefined `<<` operator to send the "display message" to `cout`:

```
cout << "I am not a crook."
```

In this case, the message comes with an argument, which is the string to be displayed. (See Figure 2.5 for a similar example.)

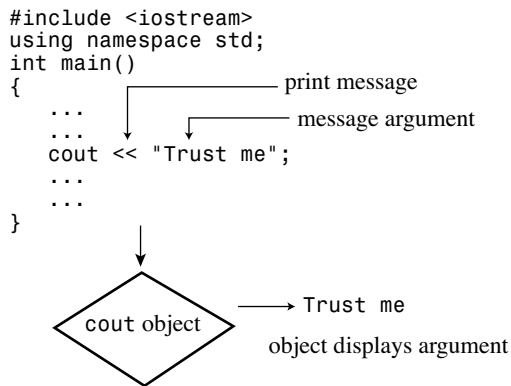


Figure 2.5 Sending a message to an object.

## Functions

Because functions are the modules from which C++ programs are built and because they are essential to C++ OOP definitions, you should become thoroughly familiar with them. Some aspects of functions are advanced topics, so the main discussion of functions comes later, in Chapter 7, "Functions: C++'s Programming Modules," and Chapter 8,

“Adventures in Functions.” However, if we deal now with some basic characteristics of functions, you’ll be more at ease and more practiced with functions later. The rest of this chapter introduces you to these function basics.

C++ functions come in two varieties: those with return values and those without them. You can find examples of each kind in the standard C++ library of functions, and you can create your own functions of each type. Let’s look at a library function that has a return value and then examine how you can write your own simple functions.

## Using a Function That Has a Return Value

A function that has a return value produces a value that you can assign to a variable or use in some other expression. For example, the standard C/C++ library includes a function called `sqrt()` that returns the square root of a number. Suppose you want to calculate the square root of 6.25 and assign it to the variable `x`. You can use the following statement in your program:

```
x = sqrt(6.25); // returns the value 2.5 and assigns it to x
```

The expression `sqrt(6.25)` invokes, or *calls*, the `sqrt()` function. The expression `sqrt(6.25)` is termed a *function call*, the invoked function is termed the *called function*, and the function containing the function call is termed the *calling function* (see Figure 2.6).

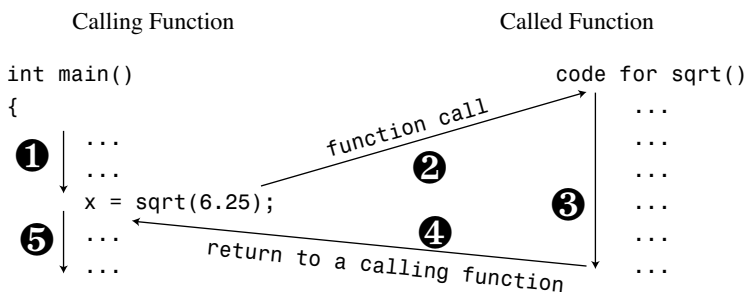


Figure 2.6 Calling a function.

The value in the parentheses (6.25, in this example) is information that is sent to the function; it is said to be *passed* to the function. A value that is sent to a function this way is called an *argument* or *parameter* (see Figure 2.7). The `sqrt()` function calculates the answer to be 2.5 and sends that value back to the calling function; the value sent back is termed the *return value* of the function. Think of the return value as what is substituted for the function call in the statement after the function finishes its job. Thus, this example assigns the return value to the variable `x`. In short, an argument is information sent to the function, and the return value is a value sent back from the function.



When you use `sqrt()` in a program, you must also provide the prototype. You can do this in either of two ways:

- You can type the function prototype into your source code file yourself.
- You can include the `cmath` (`math.h` on older systems) header file, which has the prototype in it.

The second way is better because the header file is even more likely than you to get the prototype right. Every function in the C++ library has a prototype in one or more header files. Just check the function description in your manual or with online help, if you have it, and the description tells you which header file to use. For example, the description of the `sqrt()` function should tell you to use the `cmath` header file. (Again, you might have to use the older `math.h` header file, which works for both C and C++ programs.)

Don't confuse the function prototype with the function definition. The prototype, as you've seen, only describes the function interface. That is, it describes the information sent to the function and the information sent back. The definition, however, includes the code for the function's workings—for example, the code for calculating the square root of a number. C and C++ divide these two features—prototype and definition—for library functions. The library files contain the compiled code for the functions, whereas the header files contain the prototypes.

You should place a function prototype ahead of where you first use the function. The usual practice is to place prototypes just before the definition of the `main()` function. Listing 2.4 demonstrates the use of the library function `sqrt()`; it provides a prototype by including the `cmath` file.

#### Listing 2.4 `sqrt.cpp`

---

```
// sqrt.cpp -- using the sqrt() function

#include <iostream>
#include <cmath>    // or math.h

int main()
{
    using namespace std;

    double area;
    cout << "Enter the floor area, in square feet, of your home: ";
    cin >> area;
    double side;
    side = sqrt(area);
    cout << "That's the equivalent of a square " << side
         << " feet to the side." << endl;
    cout << "How fascinating!" << endl;
    return 0;
}
```

---



### Using Library Functions

C++ library functions are stored in library files. When the compiler compiles a program, it must search the library files for the functions you've used. Compilers differ on which library files they search automatically. If you try to run Listing 2.4 and get a message that `_sqrt` is an undefined external (sounds like a condition to avoid!), chances are that your compiler doesn't automatically search the math library. (Compilers like to add an underscore prefix to function names—another subtle reminder that they have the last say about your program.) If you get such a message, check your compiler documentation to see how to have the compiler search the correct library. If you get such a complaint on a Unix implementation, for example, it may require that you use the `-lm` option (for *library math*) at the end of the command line:

```
CC sqrt.C -lm
```

Some versions of the Gnu compiler under Linux behave similarly:

```
g++ sqrt.C -lm
```

Merely including the `cmath` header file provides the prototype but does not necessarily cause the compiler to search the correct library file.

Here's a sample run of the program in Listing 2.4:

```
Enter the floor area, in square feet, of your home: 1536
That's the equivalent of a square 39.1918 feet to the side.
How fascinating!
```

Because `sqrt()` works with type `double` values, the example makes the variables that type. Note that you declare a type `double` variable by using the same form, or syntax, as when you declare a type `int` variable:

```
type-name variable-name;
```

Type `double` allows the variables `area` and `side` to hold values with decimal fractions, such as `1536.0` and `39.1918`. An apparent integer, such as `1536`, is stored as a real value with a decimal fraction part of `.0` when stored in a type `double` variable. As you'll see in Chapter 3, type `double` encompasses a much greater range of values than type `int`.

C++ allows you to declare new variables anywhere in a program, so `sqrt.cpp` didn't declare `side` until just before using it. C++ also allows you to assign a value to a variable when you create it, so you could also have done this:

```
double side = sqrt(area);
```

You'll learn more about this process, called *initialization*, in Chapter 3.

Note that `cin` knows how to convert information from the input stream to type `double`, and `cout` knows how to insert type `double` into the output stream. As noted earlier, these objects are smart.

## Function Variations

Some functions require more than one item of information. These functions use multiple arguments separated by commas. For example, the math function `pow()` takes two arguments and returns a value equal to the first argument raised to the power given by the second argument. It has this prototype:

```
double pow(double, double); // prototype of a function with two arguments
```

If, say, you wanted to find  $5^8$  (5 to the eighth power), you would use the function like this:

```
answer = pow(5.0, 8.0); // function call with a list of arguments
```

Other functions take no arguments. For example, one of the C libraries (the one associated with the `cstdlib` or the `stdlib.h` header file) has a `rand()` function that has no arguments and that returns a random integer. Its prototype looks like this:

```
int rand(void); // prototype of a function that takes no arguments
```

The keyword `void` explicitly indicates that the function takes no arguments. If you omit `void` and leave the parentheses empty, C++ interprets this as an implicit declaration that there are no arguments. You could use the function this way:

```
myGuess = rand(); // function call with no arguments
```

Note that unlike some computer languages, in C++ you must use the parentheses in the function call even if there are no arguments.

There also are functions that have no return value. For example, suppose you wrote a function that displayed a number in dollars-and-cents format. You could send to it an argument of, say, 23.5, and it would display \$23.50 onscreen. Because this function sends a value to the screen instead of to the calling program, it doesn't require a return value. You indicate this in the prototype by using the keyword `void` for the return type:

```
void bucks(double); // prototype for function with no return value
```

Because `bucks()` doesn't return a value, you can't use this function as part of an assignment statement or of some other expression. Instead, you have a pure function call statement:

```
bucks(1234.56); // function call, no return value
```

Some languages reserve the term *function* for functions with return values and use the terms *procedure* or *subroutine* for those without return values, but C++, like C, uses the term *function* for both variations.

## User-Defined Functions

The standard C library provides more than 140 predefined functions. If one fits your needs, by all means use it. But often you have to write your own, particularly when you design classes. Anyway, it's fun to design your own functions, so now let's examine that process. You've already used several user-defined functions, and they have all been named `main()`. Every C++ program must have a `main()` function, which the user must define.

Suppose you want to add a second user-defined function. Just as with a library function, you can call a user-defined function by using its name. And, as with a library function, you must provide a function prototype before using the function, which you typically do by placing the prototype above the `main()` definition. But now you, not the library vendor, must provide source code for the new function. The simplest way is to place the code in the same file after the code for `main()`. Listing 2.5 illustrates these elements.

### Listing 2.5 `ourfunc.cpp`

---

```
// ourfunc.cpp -- defining your own function
#include <iostream>
void simon(int);    // function prototype for simon()

int main()
{
    using namespace std;
    simon(3);       // call the simon() function
    cout << "Pick an integer: ";
    int count;
    cin >> count;
    simon(count);  // call it again
    cout << "Done!" << endl;
    return 0;
}

void simon(int n)  // define the simon() function
{
    using namespace std;
    cout << "Simon says touch your toes " << n << " times." << endl;
}
// void functions don't need return statements
```

---

The `main()` function calls the `simon()` function twice, once with an argument of 3 and once with a variable argument `count`. In between, the user enters an integer that's used to set the value of `count`. The example doesn't use a newline character in the `cout` prompting message. This results in the user input appearing on the same line as the prompt. Here is a sample run of the program in Listing 2.5:

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
Done!
```

### Function Form

The definition for the `simon()` function in Listing 2.5 follows the same general form as the definition for `main()`. First, there is a function header. Then, enclosed in braces, comes the function body. You can generalize the form for a function definition as follows:

```

type functionname(argumentlist)
{
    statements
}

```

Note that the source code that defines `simon()` follows the closing brace of `main()`. Like C, and unlike Pascal, C++ does not allow you to embed one function definition inside another. Each function definition stands separately from all others; all functions are created equal (see Figure 2.8).

```

                                #include <iostream>
                                using namespace std;

function prototypes { void simon(int);
                    { double taxes(double);

function #1 { int main()
             { {
               ...
               return 0;
             }
             }

function #2 { void simon(int n)
             { {
               ...
             }
             }

function #3 { double taxes(double t)
             { {
               ...
               return 2 * t;
             }
             }

```

Figure 2.8 Function definitions occur sequentially in a file.

## Function Headers

The `simon()` function in Listing 2.5 has this header:

```
void simon(int n)
```

The initial `void` means that `simon()` has no return value. So calling `simon()` doesn't produce a number that you can assign to a variable in `main()`. Thus, the first function call looks like this:

```
simon(3);           // ok for void functions
```

Because poor `simon()` lacks a return value, you can't use it this way:

```
simple = simon(3);  // not allowed for void functions
```

The `int n` within the parentheses means that you are expected to use `simon()` with a single argument of type `int`. The `n` is a new variable assigned the value passed during a

function call. Thus, the following function call assigns the value 3 to the `n` variable defined in the `simon()` header:

```
simon(3);
```

When the `cout` statement in the function body uses `n`, it uses the value passed in the function call. That's why `simon(3)` displays a 3 in its output. The call to `simon(count)` in the sample run causes the function to display 512 because that was the value entered for `count`. In short, the header for `simon()` tells you that this function takes a single type `int` argument and that it doesn't have a return value.

Let's review `main()`'s function header:

```
int main()
```

The initial `int` means that `main()` returns an integer value. The empty parentheses (which optionally could contain `void`) means that `main()` has no arguments. Functions that have return values should use the keyword `return` to provide the return value and to terminate the function. That's why you've been using the following statement at the end of `main()`:

```
return 0;
```

This is logically consistent: `main()` is supposed to return a type `int` value, and you have it return the integer 0. But, you might wonder, to what are you returning a value? After all, nowhere in any of your programs have you seen anything calling `main()`:

```
squeeze = main(); // absent from our programs
```

The answer is that you can think of your computer's operating system (Unix, say, or Windows) as calling your program. So `main()`'s return value is returned not to another part of the program but to the operating system. Many operating systems can use the program's return value. For example, Unix shell scripts and Windows's command-line interface batch files can be designed to run programs and test their return values, usually called *exit values*. The normal convention is that an exit value of zero means the program ran successfully, whereas a nonzero value means there was a problem. Thus, you can design a C++ program to return a nonzero value if, say, it fails to open a file. You can then design a shell script or batch file to run that program and to take some alternative action if the program signals failure.

### Keywords

Keywords are the vocabulary of a computer language. This chapter has used four C++ keywords: `int`, `void`, `return`, and `double`. Because these keywords are special to C++, you can't use them for other purposes. That is, you can't use `return` as the name for a variable or `double` as the name of a function. But you can use them as part of a name, as in `painter` (with its hidden `int`) or `return_aces`. Appendix B, "C++ Reserved Words," provides a complete list of C++ keywords. Incidentally, `main` is not a keyword because it's not part of the language. Instead, it is the name of a required function. You can use `main` as a variable name. (That can cause a problem in circumstances too esoteric to describe here, and because it is confusing in any case, you'd best not.) Similarly, other function names and

object names are not keywords. However, using the same name, say `cout`, for both an object and a variable in a program confuses the compiler. That is, you can use `cout` as a variable name in a function that doesn't use the `cout` object for output, but you can't use `cout` both ways in the same function.

## Using a User-Defined Function That Has a Return Value

Let's go one step further and write a function that uses the return statement. The `main()` function already illustrates the plan for a function with a return value: Give the return type in the function header and use `return` at the end of the function body. You can use this form to solve a weighty problem for those visiting the United Kingdom. In the United Kingdom, many bathroom scales are calibrated in *stone* instead of in U.S. pounds or international kilograms. The word *stone* is both singular and plural in this context. (The English language does lack the internal consistency of, say, C++.) One stone is 14 pounds, and the program in Listing 2.6 uses a function to make this conversion.

### Listing 2.6 `convert.cpp`

---

```
// convert.cpp -- converts stone to pounds
#include <iostream>
int stonetolb(int);    // function prototype
int main()
{
    using namespace std;
    int stone;
    cout << "Enter the weight in stone: ";
    cin >> stone;
    int pounds = stonetolb(stone);
    cout << stone << " stone = ";
    cout << pounds << " pounds." << endl;
    return 0;
}

int stonetolb(int sts)
{
    return 14 * sts;
}
```

---

Here's a sample run of the program in Listing 2.6:

```
Enter the weight in stone: 15
15 stone = 210 pounds.
```

In `main()`, the program uses `cin` to provide a value for the integer variable `stone`. This value is passed to the `stonetolb()` function as an argument and is assigned to the variable `sts` in that function. `stonetolb()` then uses the `return` keyword to return the value of `14 * sts` to `main()`. This illustrates that you aren't limited to following `return` with a simple number. Here, by using a more complex expression, you avoid the bother of having

to create a new variable to which to assign the value before returning it. The program calculates the value of that expression (210 in this example) and returns the resulting value. If returning the value of an expression bothers you, you can take the longer route:

```
int stonetolb(int sts)
{
    int pounds = 14 * sts;
    return pounds;
}
```

Both versions produce the same result. The second version, because it separates the computation process from the return process, is easier to read and modify.

In general, you can use a function with a return value wherever you would use a simple constant of the same type. For example, `stonetolb()` returns a type `int` value. This means you can use the function in the following ways:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16) << " pounds." << endl;
```

In each case, the program calculates the return value and then uses that number in these statements.

As these examples show, the function prototype describes the function interface—that is, how the function interacts with the rest of the program. The argument list shows what sort of information goes into the function, and the function type shows the type of value returned. Programmers sometimes describe functions as *black boxes* (a term from electronics) specified by the flow of information into and out of them. The function prototype perfectly portrays that point of view (see Figure 2.9).

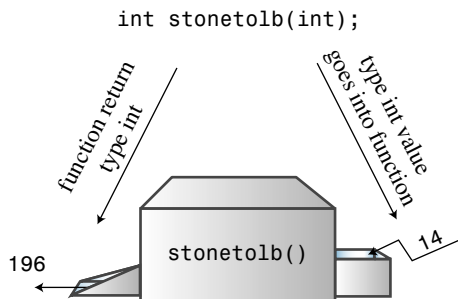


Figure 2.9 The function prototype and the function as a black box.

The `stonetolb()` function is short and simple, yet it embodies a full range of functional features:

- It has a header and a body.
- It accepts an argument.

- It returns a value.
- It requires a prototype.

Consider `stoneto1b()` as a standard form for function design. You'll further explore functions in Chapters 7 and 8. In the meantime, the material in this chapter should give you a good feel for how functions work and how they fit into C++.

## Placing the `using` Directive in Multifunction Programs

Notice that Listing 2.5 places a `using` directive in each of the two functions:

```
using namespace std;
```

This is because each function uses `cout` and thus needs access to the `cout` definition from the `std` namespace.

There's another way to make the `std` namespace available to both functions in Listing 2.5, and that's to place the directive outside and above both functions:

```
// ourfunc1.cpp -- repositioning the using directive
#include <iostream>
using namespace std; // affects all function definitions in this file
void simon(int);

int main()
{
    simon(3);
    cout << "Pick an integer: ";
    int count;
    cin >> count;
    simon(count);
    cout << "Done!" << endl;
    return 0;
}

void simon(int n)
{
    cout << "Simon says touch your toes " << n << " times." << endl;
}
```

The current prevalent philosophy is that it's preferable to be more discriminating and limit access to the `std` namespace to only those functions that need access. For example, in Listing 2.6, only `main()` uses `cout`, so there is no need to make the `std` namespace available to the `stoneto1b()` function. Thus, the `using` directive is placed inside the `main()` function only, limiting `std` namespace access to just that function.



In summary, you have several choices for making `std` namespace elements available to a program. Here are some:

- You can place the following above the function definitions in a file, making all the contents of the `std` namespace available to every function in the file:

```
using namespace std;
```

- You can place the following in a specific function definition, making all the contents of the `std` namespace available to that specific function:

```
using namespace std;
```

- Instead of using

```
using namespace std;
```

you can place `using` declarations like the following in a specific function definition and make a particular element, such as `cout`, available to that function:

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

- You can omit the `using` directives and declarations entirely and use the `std::` prefix whenever you use elements from the `std` namespace:

```
std::cout << "I'm using cout and endl from the std namespace" << std::endl;
```

## Naming Conventions

C++ programmers are blessed (or cursed) with myriad options when naming functions, classes, and variables. Programmers have strong and varied opinions about style, and these often surface as holy wars in public forums. Starting with the same basic idea for a function name, a programmer might select any of the following:

```
MyFunction( )
myfunction( )
myFunction( )
my_function( )
my_funct( )
```

The choice will depend on the development team, the idiosyncrasies of the technologies or libraries used, and the tastes and preferences of the individual programmer. Rest assured that any style consistent with the C++ rules presented in Chapter 3 is correct as far as the C++ language is concerned, and it can be used based on your own judgment.

Language allowances aside, it is worth noting that a personal naming style—one that aids you through consistency and precision—is well worth pursuing. A precise, recognizable personal naming convention is a hallmark of good software engineering, and it will aid you throughout your programming career.

## Summary

A C++ program consists of one or more modules called functions. Programs begin executing at the beginning of the function called `main()` (all lowercase), so you should always have a function by this name. A function, in turn, consists of a header and a body. The function header tells you what kind of return value, if any, the function produces and what sort of information it expects arguments to pass to it. The function body consists of a series of C++ statements enclosed in paired braces `{ }`.

C++ statement types include the following:

- **Declaration statement**—A declaration statement announces the name and the type of a variable used in a function.
- **Assignment statement**—An assignment statement uses the assignment operator (`=`) to assign a value to a variable.
- **Message statement**—A message statement sends a message to an object, initiating some sort of action.
- **Function call**—A function call activates a function. When the called function terminates, the program returns to the statement in the calling function immediately following the function call.
- **Function prototype**—A function prototype declares the return type for a function, along with the number and type of arguments the function expects.
- **Return statement**—A return statement sends a value from a called function back to the calling function.

A class is a user-defined specification for a data type. This specification details how information is to be represented and also the operations that can be performed with the data. An object is an entity created according to a class prescription, just as a simple variable is an entity created according to a data type description.

C++ provides two predefined objects (`cin` and `cout`) for handling input and output. They are examples of the `istream` and `ostream` classes, which are defined in the `iostream` file. These classes view input and output as streams of characters. The insertion operator (`<<`), which is defined for the `ostream` class, lets you insert data into the output stream, and the extraction operator (`>>`), which is defined for the `istream` class, lets you extract information from the input stream. Both `cin` and `cout` are smart objects, capable of automatically converting information from one form to another according to the program context.

C++ can use the extensive set of C library functions. To use a library function, you should include the header file that provides the prototype for the function.

Now that you have an overall view of simple C++ programs, you can go on in the next chapters to fill in details and expand horizons.

## Chapter Review

You can find the answers to the chapter review at the end of each chapter in Appendix J, “Answers to Chapter Review.”

1. What are the modules of C++ programs called?
2. What does the following preprocessor directive do?  

```
#include <iostream>
```
3. What does the following statement do?  

```
using namespace std;
```
4. What statement would you use to print the phrase “Hello, world” and then start a new line?
5. What statement would you use to create an integer variable with the name `cheeses`?
6. What statement would you use to assign the value 32 to the variable `cheeses`?
7. What statement would you use to read a value from keyboard input into the variable `cheeses`?
8. What statement would you use to print “We have X varieties of cheese,” where the current value of the `cheeses` variable replaces X?
9. What do the following function prototypes tell you about the functions?  

```
int froop(double t);  
void rattle(int n);  
int prune(void);
```
10. When do you not have to use the keyword `return` when you define a function?
11. Suppose your `main()` function has the following line:  

```
cout << "Please enter your PIN: ";
```

And suppose the compiler complains that `cout` is an unknown identifier. What is the likely cause of this complaint, and what are three ways to fix the problem?

## Programming Exercises

1. Write a C++ program that displays your name and address (or if you value your privacy, a fictitious name and address).
2. Write a C++ program that asks for a distance in furlongs and converts it to yards. (One furlong is 220 yards.)

3. Write a C++ program that uses three user-defined functions (counting `main()` as one) and produces the following output:

```
Three blind mice
Three blind mice
See how they run
See how they run
```

One function, called two times, should produce the first two lines, and the remaining function, also called twice, should produce the remaining output.

4. Write a program that asks the user to enter his or her age. The program then should display the age in months:

```
Enter your age: 29
```

```
Your age in months is 384.
```

5. Write a program that has `main()` call a user-defined function that takes a Celsius temperature value as an argument and then returns the equivalent Fahrenheit value. The program should request the Celsius value as input from the user and display the result, as shown in the following code:

```
Please enter a Celsius value: 20
20 degrees Celsius is 68 degrees Fahrenheit.
```

For reference, here is the formula for making the conversion:

$$\text{Fahrenheit} = 1.8 \times \text{degrees Celsius} + 32.0$$

6. Write a program that has `main()` call a user-defined function that takes a distance in light years as an argument and then returns the distance in astronomical units. The program should request the light year value as input from the user and display the result, as shown in the following code:

```
Enter the number of light years: 4.2
4.2 light years = 265608 astronomical units.
```

An astronomical unit is the average distance from the earth to the sun (about 150,000,000 km or 93,000,000 miles), and a light year is the distance light travels in a year (about 10 trillion kilometers or 6 trillion miles). (The nearest star after the sun is about 4.2 light years away.) Use type `double` (as in Listing 2.4) and this conversion factor:

$$1 \text{ light year} = 63,240 \text{ astronomical units}$$

7. Write a program that asks the user to enter an hour value and a minute value. The `main()` function should then pass these two values to a type `void` function that displays the two values in the format shown in the following sample run:

```
Enter the number of hours: 9
Enter the number of minutes: 28
Time: 9:28
```

# Index

## Symbols

---

**<=**, 217

**+** (addition operator), overloading, 569-572

**-=** (assignment operator), 212

**\*=** (assignment operator), 212

**+=** (assignment operator), 211

**%=** (assignment operator), 212

**/=** (assignment operator), 212

**=** (assignment operator), 43-44, 644, 767-768, 772-775

compared to equality operator, 218-220

custom definitions, 645-646

enumerator values, setting, 152

overloading, 652-658

sayings1.cpp, 656

string1.cpp, 653-656

string1.h, 652-653

potential problems, 645

strings, 133-134

structures, 145-146

when to use, 644

**&** (bitwise AND operator), 1239-1240

**~** (bitwise negation operator), 1237

**|** (bitwise OR operator), 1237-1238

**^** (bitwise XOR operator), 1238

**{}** (braces), 258

**[]** (brackets), 649-651

**,** (comma operator), 214-217

example, 214-216

precedence, 217

**/\*...\*/** comment notation, 33

**//** comment notation, 27, 32

**+** (concatenation operator), strings, 133-134

**?:** (conditional operator), 273-274

**--** (decrement operator), 207-208

pointers, 210-211

postfixing, 209-210

prefixing, 209-210

**\*** (dereferencing operator), 155-159  
pointers, 171-172

**/** (division operator), 100-101

**==** (equality operator), 217  
compared to assignment operator, 218-220

**++** (increment operator), 197, 207-208

pointers, 210-211

postfixing, 209-210

prefixing, 209-210

**!=** (inequality operator), 217

**&&** (logical AND operator), 262

alternative representations, 270

example, 263-265

precedence, 269-270

ranges, 265-267

**!** (logical NOT operator), 267-269

alternative representations, 270

precedence, 269

**||** (logical OR operator), 260-262

alternative representations, 270

example, 261-262

**-]\*** (member dereferencing operator), 1243-1246

**.\*** (member dereferencing operator), 1243-1246

**%** (modulus operator), 101-102

**\*** (multiplication operator), overloading, 574-578

**\n** newline character, 38-39

**<** operator, 217

**.** (period), 255

**#** (pound sign), 234

**"** (quotation marks), 36

**&** (reference operator), 383-386

**]]=** (right shift and assign operator), 1237

**]])(right shift operator), 1236**

**::** (scope resolution operator), 467, 514, 1332,

**;** (semicolon), 29-30

- (subtraction operator), overloading, 574-578
- ~ (tilde), 529
- \_ (underscore), 1222

## A

---

- ABCs (abstract base classes), 746-749**
  - ABC philosophy, 756
  - AcctABC example, 749-751, 754-755
  - enforcing interface rules with, 757
- abort() function, 897-898, 928-930**
- abstract data types (ADTs), 552**
- abstraction, 507. See also classes**
- access control, 892**
- access control (classes), 511-513**
- accessing**
  - content of template and parameter packs, 1198-1199
  - strings, 1259
- AcctABC class, 749-751, 754-755**
- acctabc.cpp, 751**
- acctabc.h, 750**
- accumulate() function, 1320**
- acquire() function, 516**
- actual arguments, 314**
- adaptable binary functions, 1035**
- adaptable binary predicate, 1035**
- adaptable functors, 1032**
- adaptable generators, 1035**
- adaptable predicate, 1035**
- adaptable unary functions, 1035**
- adapters, 1002**
- adding vectors, 590**
- addition operator (+), overloading, 569-572**
- addpntrs.cpp, 167**
- address.cpp, 154**
- addresses**
  - addresses of functions, obtaining, 362
  - of arrays, 170
  - structure addresses, passing, 351-353
- adjacent\_difference() function, 1321-1322**
- adjacent\_find() function, 1287, 1290**
- ADTs (abstract data types), 552-557**
- algorithms, 1035**
  - copying, 1036
  - groups, 1035-1036
  - in-place, 1036
  - properties, 1036-1037

- aliases**
  - creating, 230
  - declarations, 1157
  - namespaces, 491
- align.cpp, 1246**
- alignof() operator, 1204**
- allocating memory, 968**
  - bad\_alloc exceptions, 921
  - dynamic memory allocation, 757
    - derived class does use new, 758-760
    - derived class doesn't use new, 757-758
    - example, 761-766
    - new operator, 160-162
- allocators, 979**
- alternative tokens, table of, 1222**
- American National Standards Institute (ANSI), C++ standard, 16**
- American Standard Code for Information Interchange. See ASCII character set**
- ampersand (&), 1239-1240**
  - logical AND operator (&&), 262
    - alternative representations, 270
    - example, 263-265
    - precedence, 269-270
    - ranges, 265-267
  - reference operator (&), 383-386
- AND operators**
  - bitwise AND (&), 1239-1240
  - logical AND (&&), 262
    - alternative representations, 270
    - example, 263-265
    - precedence, 269-270
    - ranges, 265-267
- and.cpp, 263-264**
- angle brackets, 1162**
- anonymous unions, 150**
- ANSI (American National Standards Institute), C++ standard, 16**
- ANSI C Standard Input/Output, 1062**
- append() method, 1265**
- append.cpp, 1125**
- appending**
  - data to files, 1125-1127
  - strings, 133-134, 1265-1266
- applications. See also compilers**
  - creating, 18-19

- portability, 15, 17-18
  - ANSI/ISO standard, 16
  - limitations, 16
- source code, 19
  - file extensions, 20
- apply() method, 1046**
- Area() method, 748**
- args, 1198**
- argument lists, 30**
- arguments, 31**
  - arrays, 322-325
  - C-style strings, 339-341
  - command-line, 1119-1120
  - default arguments, 409-412
  - formal/actual, 314
  - multiple, 314-320
    - n\_chars() example, 314-317
    - probability() example, 318-320
  - parameters, 314
  - passing by reference, 386, 389-390
  - passing by value, 313-314
  - reference arguments, 392-394, 408-409
  - two-dimensional arrays, 337-339
  - type conversions, 106
- arguments (functions), 49, 53**
- arith.cpp, 98**
- arithmetic, pointers, 167-172**
- arithmetic operators, 97-99**
  - associativity, 99-100
  - division (/), 100-101
  - functor equivalents, 1031-1032
  - modulus (%), 101-102
  - order of precedence, 99-100
  - overloading
    - addition operator (+), 569-572
    - multiplication operator (\*), 574-578, 600
    - subtraction operator (-), 574-578
    - vector class, 599-600
- array notation, 173**
- array objects, 355**
  - fill function, 357
  - versus arrays, 188-189
  - versus vector objects, 188-189
- array template class, 187**
- ArrayDb class, 791**
- arraynew.cpp, 166**
- arrayone.cpp, 117**

- arrays**
  - addresses, 170
  - array notation, 173
  - arrays of objects, 546-549
    - declaring, 546
    - example, 547-549
    - initializing, 546
  - as function arguments, 322-325
  - const keyword, 327-328
  - declaring, 116-119
  - defined, 77, 116
  - design decisions, 325-326
  - displaying contents of, 327-328
  - dynamic arrays, 172-173
    - creating, 164-165
    - sample program, 165-167
  - examples, 328-331
  - filling, 326-327
  - function idioms, 331
  - functions and, 320-321
  - indexes, 117
  - initializing, 117-120
    - in C++11, 120
  - modifying, 328
  - naming, 172
  - one-dimensional arrays, 244
  - pointer arithmetic, 167-172
  - pointers, 321-322
  - ranges, 332-334
  - static binding, 172
  - strings, 123-124
    - initializing, 121
  - structures, 147-148
  - subscripts, 117
  - templates, non-type arguments, 843-845
  - two-dimensional arrays, 244-249, 337-339
    - declaring, 244-246
    - initializing, 246-249
  - variable arrays, 1329
  - versus vector objects, 188-189
- arraytp.h, 843-844**
- arrfun1.cpp, 321**
- arrfun2.cpp, 323**
- arrfun3.cpp, 328**
- arrfun4.cpp, 332**
- arrstruc.cpp, 147-148**

**ASCII character set, table of, 1225-1229****assert, 1204****assign\_st.cpp, 145****assign() method, 1266, 1278****assignable objects, 1008****assigning**

strings, 1266

values to pointers, 171

values to variables, 43

variable values, 43

**assignment**

string class, 133-134

type conversions, 103-104

**assignment methods, 1260****assignment operator (=), 43-44, 644, 767-768, 772-775**compared to equality operator,  
218-220

custom definitions, 645-646

enumerator values, setting, 152

enumerators, value ranges, 153

overloading, 652-658

sayings1.cpp, 656

string1.cpp, 653-656

string1.h, 652-653

potential problems, 645

strings, 133-134

structures, 145-146

when to use, 644

**assignment operators, combination****assignment operators, 211-212****assignment statements, 43-44****assignments, 1172-1173****associative containers, 1018, 1026**

methods, 1281-1284

multimap, 1023-1025

set, 1019-1022

**associativity, arithmetic operators, 99-100****associativity of operators, 1231**

examples, 1234

table of, 1232-1234

asterisk (\*), dereferencing operator  
(\*)155, 159

pointers171-172

**at() method, 1259****atan() function, 348****atan2() function, 348****ATM queue simulation**

bank.cpp simulation, 695

Customer class, 694

Queue class

class declaration, 691

public interface, 679

**ATM queue simulation, 678**

bank.cpp simulation, 694-698

Customer class, 690-691

Queue class

class declaration, 694

design, 679

implementation, 680-682

methods, 682-690

public interface, 680

**auto, 370**

declarations, 1155

**auto declarations, 109****auto keyword, 472****auto ptr, 1158****auto.cpp, 456-457****automatic memory storage, 182****automatic sizing (strings), 966-967****automatic teller machine simulation. See****ATM queue simulation****automatic type conversion. See type  
conversion****automatic variables, 182, 314, 453-457**

example, 455-457

initializing, 458

stacks, 458-459

**autoptr template, 1333****auto\_ptr class, 969, 973-975**

versus unique\_ptr, 975-977

**average() function, 800**

---

**B**

---

**back insert iterators, 1005-1007****bad() method, 296****bad() stream state method, 1098-1102****badbit stream state, 1097-1102****bad\_alloc exceptions, 921****Balance() function, 731****bank.cpp, 695-697****bank.cpp simulation, 694-698****Base 10 notation, 1215**



**Base 16 notation, 1216**

- binary equivalents, 1217-1218

**Base 2 notation, 1217**

- hexadecimal equivalents, 1217-1218

**Base 8 notation, 1215-1216****base classes**

- ABCs (abstract base classes), 746-749

- ABC philosophy, 756

- AcctABC example, 749-751, 754-755

- enforcing interface rules with, 757

- components, initializing, 798-799

- friends, accessing, 801-804

- methods, accessing, 800-801

- objects, accessing, 801

- relationships with derived classes, 718-720

- TableTennisPlayer example, 708-710

- using declarations, 807-808

- virtual, methods, 826

- virtual base classes, 815-817

- combining with nonvirtual base classes, 828

- constructors, 817-818

- dominance, 828-829

- methods, 818-828

**base-class functions, 777****begin() method, 981-984, 1251-1252, 1275****best matches, 432-434****bidirectional iterators, 998****Big Endian, 1218****bigstep.cpp, 205****binary files, 1127-1133****binary functions, 1027-1030****binary numbers, 1217**

- hexadecimal equivalents, 1217-1218

**binary operators, 601, 1234****binary predicates, 1027, 1030****binary search operations**

- binary\_search() function, 1304, 1310

- equal\_range() function, 1304, 1309

- lower\_bound() function, 1304, 1309

- upper\_bound() function, 1304, 1309

**binary searching**

- binary\_search() function, 1304, 1310

- equal\_range() function, 1304, 1309

- lower\_bound() function, 1304, 1309

- upper\_bound() function, 1304, 1309

**binary.cpp, 1131****binary\_search() function, 1304, 1310****binding**

- dynamic binding, 173, 737, 739-740

- static binding, 172, 737, 740

**bit fields, 148****bit values, represented by constants, 1085****bitmask data type, 1085****bits, 68-69**

- clearing, 1086

- testing, 1241-1242

- toggling, 1241

- turning off, 1241

- turning on, 1241

**bitwise AND operator (&), 1239-1240****bitwise negation operator (~), 1237****bitwise operators, 1235**

- alternative representations, 1240

- logical bitwise operators, 1237-1240

- shift operators, 1235-1237

- overloading, 581-587

- testing bit values, 1241-1242

- toggling, 1241

- turning bits off, 1241

- turning bits on, 1241

**bitwise OR operator (|), 1237-1238****bitwise XOR operator (^), 1238****block scope, 454****block.cpp, 212****blocks, 212-214****body (function), 29**

- for loops, 196-197

**bondini.cpp, 86****books, 1323-1324****bool data type, 90****boolalpha manipulator, 1090****Boost project, 1205-1207****bottom-up programming, 13, 331****bound template friend functions, 861-864****braces {}, 258****bracket notation, 649-651****brackets, angle brackets, 1162****branching statements**

- if, 254

- bug prevention, 260

- example, 255

- syntax, 254

- if else
  - example, 256–257
  - formatting, 257–258
  - if else if else construction, 258–260
  - syntax, 255
- switch, 274
  - enumerators as labels, 278–280
  - example, 275–278
  - syntax, 275
- Brass class**
  - class declaration, 723–727, 730–733
  - virtual destructors, 737
  - virtual method behavior, 734–736
- brass.cpp, 727**
- brass.h, 724**
- BrassPlus class**
  - class declaration, 723–726
  - class implementation, 727, 730–731
  - class objects, 732–733
  - virtual destructors, 737
  - virtual method behavior, 734–736
- break statement, 280–282**
- bucks() function, 53**
- buffers, 1063–1064, 1067**
  - flushing, 1063
- Build All option (compiler), 24**
- buildstr() function, 341**
- buy() function, 516**
- bytes, 69**

---

## C

- C language**
  - ANSI C, 17
  - classic C, 17
  - development history, 11
  - programming philosophy, 11–13
- C++, Macintosh, 25**
- C++ FAQ Lite, 1325**
- C++ FAQs, Second Edition, 1323**
- The C++ Programming Language, Third Edition, 1324***
- The C++ Standard Library: A Tutorial and Reference, 1323–1324***
- C++ Templates: The Complete Guide, 1324***
- C++11**
  - arrays, initializing, 120
  - auto declarations, 109
  - basic assignments, 1260
  - constructors
    - initialization list, 1258
    - Rvalue reference, 1256
  - container requirements, 1010
  - containers, unordered associative containers, 1283
  - exception specifications, 908
  - initializer\_list template, 1051–1053
  - libraries, 1203
  - list initialization, 537
  - noexcept, 1248
  - range-based loops, 233–234
  - scoped enumerations, 551–552
  - STL, 1271
    - containers, 1271–1273
    - structure initialization, 144
    - template aliases, 866
- C-style strings, 120–122**
  - in arrays, 123–124
  - combining with numeric input, 130–131
  - concatenating, 122
  - empty lines, 130
  - failbits, 130
  - null characters, 121
  - passing as arguments, 339–341
  - pointers, 173–178
  - returning from functions, 341–343
  - string input, entering, 124–126
  - string input, reading with get(), 127–130
  - string input, reading with getline(), 126–127
- C-style strings, comparing, 220–223**
- call signatures, 1194**
- calling**
  - class member functions, 523
  - constructors, 526–527
  - functions, 309–311
    - pointers, 363–364
- calling functions, 30, 49**
- calling.cpp, 306**
- callme1() function, 636**
- callme2() function, 636**
- capacity() method, 966, 1251, 1279**
- caret (^), 1238**
- carrots.cpp, 41**

- case sensitivity, 27, 32**
- casting, 1330-1331**
  - downcasting, 738
  - upcasting, 738
    - implicit upcasting, 807
- casting data types, 606-610, 612**
- casting types, 107-109**
- catch keyword, 900**
- catching exceptions, 900, 916-917**
- CC compiler (UNIX), 21-22**
- cctype library, 270-272**
- cctypes.cpp, 271-272**
- cerr object, 1067**
- cfront translator, 21**
- char data type, 80-87, 1064**
  - escape sequences, 84-87
  - signed char, 88-89
  - universal character names, 87-88
  - unsigned char, 88-89
  - wchar\_t, 89
- character strings, 36**
- characters**
  - ASCII character set, table of, 1225-1229
  - fill, 1081-1082
- chartype.cpp, 81**
- CHAR\_BIT constant, 72**
- CHAR\_MAX constant, 72**
- char\_type type, 1250**
- check\_it.cpp, 1096**
- cheers() function, 307-309**
- choices.cpp, 188**
- choosing integer types, 76-77**
- cin, cin.get() function, 235-237, 241-244**
- cin object, 1067, 1093-1095**
  - get() function, 128-130
  - getline() function, 126-127
  - loops, 234-235
  - operator overloading, 1095-1097
  - stream states, 1097-1098
    - effects, 1100-1102
    - exceptions, 1099-1100
    - setting, 1098
- cin statement, 46**
- cin.get() function, 235-237, 241-244, 317**
- cin.get() member function, 1103-1105**
- cin.get(ch) function, 317**
- cinexp.cpp, 1099**
- cinfish.cpp, 283-284**
- cingolf.cpp, 285-286**
- class declaration, 511-513**
- class inheritance, private inheritance, 797**
  - base-class components, initializing, 798-799
  - base-class friends, accessing, 801-804
  - base-class methods, accessing, 800-801
  - base-class objects, accessing, 801
  - compared to containment, 806
  - Student class example, 798, 804-805
- class keyword, 831**
- class member functions, operator overloading, 587-588**
- class scope, 454, 514, 549-551**
- class templates, 830-837**
  - arrays, non-type arguments, 843-845
  - complex, 1045
  - explicit instantiations, 850
  - explicit specializations, 850-851
  - friend classes, 858
    - bound template friend functions, 861-864
    - non-template friend functions, 858-861
    - unbound template friend functions, 864-865
  - implicit instantiations, 850
  - member templates, 854-855
  - parameters, 855-858
  - partial specializations, 851-852
  - pointers, stacks of pointers, 837-843
  - versatility, 845-846
    - default type parameters, 849
    - multiple type parameters, 847
    - recursive use, 846-847
- classes, 47-48, 508, 520, 1159**
  - ABCs (abstract base classes), 746-749
    - ABC philosophy, 756
    - AcctABC example, 749-751, 754-755
    - enforcing interface rules with, 757
  - abstraction, 507
  - access control, 511-513
  - AcctABC, 749-751, 754-755
  - ADTs (abstract data types), 552-557
  - array template class, 187
  - ArrayDb, 791
  - auto\_ptr, 969, 973-975

- bad\_alloc, 921
- base classes
  - components, initializing, 798-799
  - friends, accessing, 801-804
  - methods, accessing, 800-801
  - objects, accessing, 801
  - using declarations, 807-808
  - virtual base classes, 815-829
- Brass
  - class declaration, 723-726
  - class implementation, 727, 730-731
  - class objects, 732-733
  - virtual destructors, 737
  - virtual method behavior, 734-736
- BrassPlus
  - class declaration, 723-726
  - class implementation, 727, 730-731
  - class objects, 732-733
  - virtual destructors, 737
  - virtual method behavior, 734-736
- class scope, 549-551
- client files, 533-536
- compared to structures, 514
- constructors, 524, 538-539, 768
  - calling, 526-527
  - conversion, 769-770
  - copy constructors, 639-644, 767
  - declaring, 525-526
  - default constructors, 527-528, 638-639, 766-767
  - defining, 525-526
  - delegating, 1180-1181
  - inheriting, 1181-1183
  - new operator, 659-661, 677-678
- converting class type, 677
- Customer, 690-691, 694
- data hiding, 511-513, 523
- data types, 507-508
- declarations, 509-511, 522
- defaulted and deleted methods, 1179-1180
- defined, 36, 47, 508
- defining, 47
- definition of, 13
- derived classes, 405
- destructors, 524, 528-529, 538-539, 768
- encapsulation, 512, 523
- exception, 917
- explicit conversion operators, 1159-1160
- friend classes, 578-580, 877, 880-883, 886-888
  - compared to class member functions, 886
  - templates, 858-865
  - Tv class example, 878-883
- header files, 530
- ifstream, 1116-1119
- implementation files, 530
- inheritance
  - assignment operators, 772-775
  - base classes, 708-710, 718-720
  - constructors, 713-715
  - derived classes, 711-712, 716-720
  - exceptions, 922-927
  - has-a relationships, 721
  - is-a relationships, 720-722, 772
  - multiple, 814, 826
  - what's not inherited, 772
- ios, 1065
- iostream, 1065
- ios\_base, 1065
  - constants representing bit values, 1085
- istream, 47, 1065
  - data types recognized, 1093-1095
  - input methods, 1109-1114
  - single-character input, 1102-1106
  - string input, 1106-1108
- member functions
  - const member functions, 537
  - const objects, 662-665
  - constructors, 524-528, 538-539, 638-639, 659-661, 677-678
  - copy constructors, 639-644
  - definitions, 509, 514-516, 523
  - destructors, 528-529, 538-539
  - friend member functions, 883-886, 888-889
  - implicit member functions, 637-638
  - inline functions, 517-518
  - invoking, 523
  - non-const objects, 663
  - object membership, 518
  - objects, returning, 662-665

- private, 513
- properties, 777-778
- public, 513
- qualified names, 514
- this pointer, 539-546
- unqualified names, 514
- member in-class initialization, 1160
- nested classes, 682, 889-891
  - access control, 892
  - scope, 891-892
  - templates, 892-896
- objects, 786-788
  - contained objects, 791-795
  - subobjects, 797
- ofstream, 1115-1119
- ostream, 47, 1065
- ostreamstream, 1142, 1144-1145
- pointers, member dereferencing operators, 1242-1246
- private inheritance
  - base-class components, initializing, 798-799
  - base-class friends, accessing, 801-802, 804
  - base-class methods, accessing, 800-801
  - base-class objects, accessing, 801
  - Student class example, 798
- protected classes, 745-746, 775
- Queue
  - class declaration, 691-694
  - design, 679
  - implementation, 680-682
  - methods, 682-690
  - public interface, 679-680
- Sales
  - sales.cpp, 924
  - sales.h, 922
  - use\_sales.cpp, 925-927
- sample program, 518-520
- special member functions, 1178-1179
- Stack, 831-836
  - pointers, 837-843
- static class members, 628-637
- stdexcept exception classes, 918-920
- Stock, 511
- streambuf, 1065
- string, 131-133, 353-354, 647, 952, 960, 965-966, 1333
  - appending, 133-134
  - assignment, 133-134
  - assignment operator, overloading, 652-658
  - automatic sizing, 966-967
  - bracket notation, 649-651
  - comparing, 960
  - comparison members, 648-649
  - complex operations, 135-136
  - concatenation, 133-134
  - constructors, 952-956
  - default constructor, 647-648
  - finding size of, 960
  - Hangman sample program, 962-965
  - input, 957-960
  - reading line by line, 136-140
  - searching, 960-961
  - static class member functions, 651-652
  - STL interface, 1038-1039
  - string comparisons, 223-224
  - structures, 144-145
- StringBad, 628
  - constructors, 632-633
  - destructor, 633
  - strngbad.cpp, 630-631
  - strngbad.h, 628-629
  - vegnews.cpp sample program, 633-637
- Student
  - contained objects interfaces, 792-795
  - contained objects, initializing, 791
  - design, 787-788
  - methods, 793-795
  - private inheritance, 798-805
  - sample program, 795-797
  - studentc.h, 789-790
- TableTennisPlayer, 708
  - tabtenn0.cpp, 709
  - tabtenn0.h, 708
  - uset0.cpp, 710
- this pointer, 539-546
- Tv, 878-879, 883
  - tv.cpp, 880-882
  - tv.h, 879-880
  - tvfm.h, 885-886
  - use\_tv.cpp, 882

- type casts, 606-612
- type conversions, 606-612
  - applying automatically, 616-618
  - conversion functions, 612-616
  - friends, 618-621
  - implicit conversion, 609
- type info, 939-944
- valarray, 786-787, 1045-1046, 1049-1051
- vector, 120, 588-590, 600, 979-991, 1045-1046, 1049-1051
  - adding elements to, 982-983
  - adding vectors, 590
  - declaring, 591-592
  - displacement vectors, 589
  - implementation comments, 602
  - member functions, 592, 597
  - multiple representations, 599
  - overloaded arithmetic operators, 599-600
  - overloading overloaded operators, 601
  - past-the-end iterators, 981-982
  - Random Walk sample program, 602, 605-606
  - removing ranges of, 982
  - shuffling elements in, 987
  - sorting, 987
  - state members, 597-599
  - vect1.cpp example, 980-981
  - vect2.cpp sample program, 984-986
  - vect3.cpp sample program, 988-991
- vector template class, 186-187
- virtual methods
  - final, 1183-1184
  - override, 1183-1184
- Worker, 810-814
- classic C, 17**
- classifying data types, 97**
- clear() method, 1258, 1278, 1283**
- clear() stream state method, 1098-1102**
- clearing bits, 1086**
- client files, creating, 533-536**
- client/server model, 520**
- climits header file, 71-73**
- clock() function, 229**
- clog object, 1067**
- close() method, 292**
- code formatting, 39**
  - source code style, 40
  - tokens, 39
  - white space, 39
- code listings**
  - acctabc.cpp, 751
  - acctabc.h, 750
  - addpntrs.cpp, 167
  - address.cpp, 154
  - align.cpp, 1246
  - and.cpp, 263-264
  - append.cpp, 1125
  - arith.cpp, 98
  - arraynew.cpp, 166
  - arrayone.cpp, 117
  - arraytp.h, 843-844
  - arrfun1.cpp, 321
  - arrfun2.cpp, 323
  - arrfun3.cpp, 328, 332
  - arrstruc.cpp, 147-148
  - assgn st.cpp, 145
  - auto.cpp, 456-457
  - bank.cpp, 695-697
  - bigstep.cpp, 205
  - binary.cpp, 1131
  - block.cpp, 212
  - bondini.cpp, 86
  - brass.cpp, 727
  - brass.h, 724
  - callable.cpp, 1192
  - calling.cpp, 306
  - carrots.cpp, 41
  - cctypes.cpp, 271-272
  - chartype.cpp, 81
  - check\_it.cpp, 1096
  - choices.cpp, 188
  - cinexcp.cpp, 1099
  - cinfish.cpp, 283-284
  - cingolf.cpp, 285-286
  - compstr1.cpp, 221
  - compstr2.cpp, 223
  - condit.cpp, 273
  - constcast.cpp, 944
  - conversion functions, stone1.cpp, 616
  - convert.cpp, 57
  - coordin.h, 449-450
  - copyit.cpp, 1004
  - count.cpp, 1121

cubes.cpp, 390  
 defaults.cpp, 1077  
 delete.cpp, 181, 185  
 divide.cpp, 100  
 dma.cpp, 762-764  
 dma.h, 761-762  
 dowhile.cpp, 232  
 enum.cpp, 279  
 equal.cpp, 219  
 error1.cpp, 897-898  
 error2.cpp, 899  
 error3.cpp, 901  
 error4.cpp, 906-907  
 error5.cpp, 910, 913  
 exceed.cpp, 75  
 exceptions, newexcp.cpp, 920  
 exc\_mean.cpp, 905-906  
 express.cpp, 200  
 external.cpp, 465  
 file1.cpp, 451  
 file2.cpp, 452  
 filefunc.cpp, 406-407  
 fileio.cpp, 1117  
 fill.cpp, 1082  
 firstref.cpp, 383  
 floatnum.cpp, 95  
 ftadd.cpp, 96  
 forloop.cpp, 196  
 formore.cpp, 203-204  
 forstr1.cpp, 206  
 forstr2.cpp, 215  
 fowl.cpp, 973  
 frnd2tmp.cpp, 860-861  
 fun\_ptr.cpp, 364, 368  
 funadap.cpp, 1034-1035  
 function overloading, leftover.cpp, 418  
 functions, tempover.cpp, 434-437  
 functions, arguments, twoarg.cpp, 316  
 functions, recursion, recur.cpp, 359  
 functor.cpp, 1028  
 funtemp.cpp, 420  
 getinfo.cpp, 45  
 get\_fun.cpp, 1107  
 hangman.cpp, 962-965  
 hexoct2.cpp, 79  
 if/cpp, 255  
 ifelse.cpp, 257  
 ifelseif.cpp, 259  
 ilit.cpp, 1053  
 init\_ptr.cpp, 158  
 inline functions.cpp, 381  
 inserts.cpp, 1006  
 instr1.cpp, 125  
 instr2.cpp, 127  
 instr3.cpp, 129  
 iomanip.cpp, 1092  
 jump.cpp, 280-281  
 lambda0.cpp, 1186  
 lambda1.cpp, 1190  
 left.cpp, 410-411  
 leftover.cpp, 416-417  
 lexcast.cpp, 1206  
 limits.cpp, 70  
 list.cpp, 1015  
 listrmv.cpp, 1039-1040  
 lotto.cpp, 319  
 manip.cpp, 1079  
 manyfrnd.cpp, 865  
 memb\_pt.cpp, 1244-1245  
 modulus.cpp, 102  
 morechar.cpp, 82  
 more\_and.cpp, 266  
 multmap.cpp, 1024-1025  
 myfirst.cpp, 28  
 mytime0.h, 566  
 mytime1.cpp, 569-570  
 mytime1.h, 569  
 mytime2.cpp, 575  
 mytime2.h, 575  
 namesp.cpp, 493-494  
 namesp.h, 493  
 namespaces, static.cpp, 479  
 nested.cpp, 247, 895-896  
 newstrct.cpp, 179-180  
 not.cpp, 267-268  
 numstr.cpp, 130  
 num\_test.cpp, 198  
 operator overloading  
     mytime0.cpp, 566  
     mytime3.cpp, 585  
     mytime3.h, 584  
     usetime0.cpp, 568  
     usetime3.cpp, 587  
 or.cpp, 261  
 ourfunc.cpp, 54  
 outfile.cpp, 290-291  
 pairs.cpp, 848  
 peeker.cpp, 1111

placenew1.cpp, 671-673  
 placenew2.cpp, 674-675  
 plus\_one.cpp, 207  
 pointer.cpp, 155  
 precise.cpp, 1082  
 protos.cpp, 310  
 ptrstr.cpp, 174-175  
 queue.cpp, 692-694  
 queue.h, 691-692  
 queuetp.h, 893-895  
 random.cpp, 1138-1139  
 randwalk.cpp, 603  
 recur.cpp, 355, 358  
 reference variables as function  
   parameters, swaps.cpp, 389  
 rtti1.cpp, 936-938  
 rtti2.cpp, 939-941  
 ruler.cpp, 360  
 rvref.cpp, 1163  
 sales.cpp, 924  
 sales.h, 922  
 sayings1.cpp, 656  
 sayings2.cpp, 665  
 secref.cpp, 385  
 setf.cpp, 1085  
 setf2.cpp, 1088  
 setops.cpp, 1021-1022  
 showpt.cpp, 1084  
 somedefs.h, 1192  
 sqrt.cpp, 51  
 stack.cpp, 554-555  
 stack.h, 553-554  
 stacker.cpp, 555-557  
 stacktem.cpp, 835-836  
 stacktp.h, 833-834  
 static.cpp, 470-471  
 stcktp1.cpp, 841-842  
 stcktp1.h, 839-840  
 stdmove.cpp, 1174  
 stock00.h, 510  
 stock1.cpp, 531  
 stock1.h, 530  
 stock2.cpp, 543  
 stock2.h, 543  
 stocks.cpp, class member functions, 515  
 stone1.cpp, 615  
 stone.cpp, 610  
 stonewt.cpp, 608  
 stonewt.h, 607  
 stonewt1.cpp, 614-615  
 stonewt1.h, 613  
 str1.cpp, 953  
 str2.cpp, 966-967, 971  
 strctfun.cpp, 348  
 strctptr.cpp, 352-353  
 strfile.cpp, 958-959  
 strgfun.cpp, 340  
 strgstl.cpp, 1038-1039  
 strin.cpp, 1144  
 string1.cpp, 653-656  
 string1.h, 653  
 strings, numeric input,  
   numstr.cpp, 135  
 strings, returning, strgback.cpp, 341  
 strings.cpp, 123  
 strngbad.cpp, 630-631  
 strngbad.h, 629  
 strout.cpp, 1143  
 strquote.cpp, 402-403  
 strtref.cpp, 395  
 strtype1.cpp, 132  
 strtype2.cpp, 134  
 strtype4.cpp, 137  
 structur.cpp, 142  
 studentc.cpp, 793  
 studentc.h, 789-790  
 studenti.cpp, 802-803  
 studenti.h, 799  
 sumafile.cpp, 294-295  
 swaps.cpp, 387-388  
 switch.cpp, 276-277  
 tabtenn0.cpp, 709  
 tabtenn0.h, 708  
 tabtenn1.cpp, 717  
 tabtenn1.h, 716  
 tempmemb.cpp, 852  
 tempparm.cpp, 856-857  
 textin1.cpp, 234  
 textin2.cpp, 236  
 textin3.cpp, 239  
 textin4.cpp, 242  
 tmp2tmp.cpp, 862-864  
 topfive.cpp, 353  
 travel.cpp, 344-345  
 truncate.cpp, 1113  
 tv.cpp, 880-882



- tv.h, 879-880
- tvfm.h, 885-886
- twoarg.cpp, 316
- twod.cpp, 846-847
- twofile1.cpp, 469
- twofile2.cpp, 469
- twoswap.cpp, 427-428
- twotemps.cpp, 422
- typecast.cpp, 108
- use new.cpp, 161
- usealgo.cpp, 1043-1044
- usebrass1.cpp, 732-733
- usebrass2.cpp, 734
- usedma.cpp, 765
- useless.cpp, 1165
- usenmsp.cpp, 494-495
- usesstok2.cpp, 547
- usestok1.cpp, 533
- usetime1.cpp, 571-572
- usetime2.cpp, 577
- usett0.cpp, 710
- usett1.cpp, 717-718
- use\_sales.cpp, 925-927
- use\_stuc.cpp, 795-797
- use\_stui.cpp, 804-805
- use\_tv.cpp, 882
- usestok0.cpp, 519
- valvect.cpp, 1048
- variadic1.cpp, 1199
- vect.cpp, 593
- vect.h, 591
- vect1.cpp, 980
- vect2.cpp, 984-985
- vect3.cpp, 988
- vegnews.cpp, 634
- vslice.cpp, 1049-1050
- waiting.cpp, 229
- while.cpp, 225
- width.cpp, 1080
- Worker0.cpp, 811-812
- Worker0.h, 810-811
- workermi.cpp, 823-825
- workermi.h, 821-822
- workmi.cpp, 826-827
- worktest.cpp, 813
- write.cpp, 1073-1074
- code style, 40**
- colon (`()`), scope-resolution operator (`::`), 514**
- combination assignment operators, 211-212**
- comma operator, 214-217**
  - example, 214-216
  - precedence, 217
- command-line processing, 1119-1120**
- comments, 27, 33**
  - `/*...*/` notation, 33
  - `//` notation, 32
- compare() method, 1264-1265**
- comparing**
  - arrays, vector objects, and array objects, 188-189
  - strings, 960, 1263-1265
    - C-style strings, comparing, 220-223
    - string class strings, comparing, 223-224
- comparison members (String class), 648-649**
- compile time, 155**
- compile time complexity, 1009-1010**
- compilers, 21**
  - CC (UNIX), 21-22
  - definition of, 11
  - g++ (Linux), 22
  - gpp, 22
  - troubleshooting, 24
  - Windows, 23-24
- compiling files separately, 447-449, 453**
- complex class template, 1045**
- composition, 785**
- compound statements (blocks), 212-214**
- compound types, 115-116**
  - enumerations, 150-152
    - enumerators, 150-151
    - value ranges, 153
    - values, setting, 152
  - pointers, 153
    - assigning values to, 171
    - C++ philosophy, 155
    - cautions, 159
    - compared to pointed-to values, 172
    - declaring, 155-159, 171
    - deferencing, 171-172
    - delete operator, 163-164
    - example, 154
    - initializing, 157-159
    - integers, 160
    - new operator, 160-162
    - pointer arithmetic, 167-172
    - pointer notation, 173

- pointers to objects, 665-670
- strings, 173-178
- structures, 140-142
  - arrays, 147-148
  - assignment, 145-146
  - bit fields, 148
  - dynamic structures, 178-180
  - example, 142-144
  - members, 141
  - string class members, 144-145
- unions, 149
  - anonymous unions, 150
  - declaring, 149
- compstr1.cpp, 221**
- compstr2.cpp, 223**
- concatenating strings, 122, 128, 1266**
- concatenating output, 46-47**
- concatenation**
  - output, 1071-1072
  - string class, 133-134
- concatenation operator (+), strings, 133-134**
- concepts**
  - containers, 1007
    - container methods compared to functions, 1039-1041
    - properties, 1008-1010
    - sequence requirements, 1011-1012
  - functors, 1027-1030
  - iterators, models, 1000-1001
- concurrent programming, 1202-1203**
- condit.cpp, 273**
- conditional operator (?:), 273-274**
- const, reference returns, 400**
- const keyword, 90-92, 473-474, 771-772**
  - arrays, 327-328
  - pointers, 334-336
  - reference variables, 401
  - temporary variables, 392-394
- const member functions, 537**
- const modifier as alternative to #define, 1327-1329**
- const objects, returning references to, 662-665**
- constant time, 1009**
- constant time complexity, 1009-1010**
- constants, 78-80. See also strings**
  - char constants. *See* char data type
  - const keyword, 90-92
  - file modes, 1122-1123
  - floating-point constants, 96
  - representing bit values, 1085
  - size\_type, 1251
  - symbolic constants, 72
  - symbolic names, 90-92
- constcast.cpp, 944**
- constructors, 524, 742, 768**
  - calling, 526-527
  - class, 524
  - conversion, 769-770
  - copy constructors, 639, 767
    - deep copying, 642-644
    - limitations, 640-642
    - shallow copying, 640
    - when to use, 639-640
  - declaring, 525-526
  - default constructors, 527-528, 638-639, 766-767
  - defining, 525-526
  - delegating, 1180-1181
  - inheritance, 713-715
  - initialization list, C++11, 1258
  - new operator, 659-661, 677-678
  - Rvalue reference, C++11, 1256
  - string class, 1253
    - constructors that use arrays, 1254
    - constructors that use n copies of character, 1257
    - constructors that use parts of arrays, 1254-1255
    - constructors that use ranges, 1257
    - copy constructors, 1255-1256
    - default constructors, 1254
    - String(), 647-648, 952-956
    - virtual base classes, 817-818
- const\_cast operator, 944**
- const\_iterator type, 1273**
- const\_reference type, 1273**
- contained objects**
  - compared to private inheritance, 806
  - initializing, 791
  - interfaces, 792-795
- container classes, 830**
- container concepts, 1007**
  - container methods compared to functions, 1039-1041
  - properties, 1008-1010
  - sequence requirements, 1011-1012

**container methods, compared to functions, 1039-1041**

**container requirements, C++11, 1010**

**container types**

- deque, 1013
- list, 1014-1017
  - member functions, 1014-1016
- priority\_queue, 1017-1018
- queue, 1017
- stack, 1018
- vector, 1012-1013

**containers, 553**

- associative, 1018, 1026
  - multimap, 1023-1025
  - set, 1019-1022
- C++11, unordered associative
  - containers, 1283
- deques, methods, 1278-1280
- lists, methods, 1278-1280
- maps, methods, 1281-1284
- methods, 1275-1277
- sets, methods, 1281-1284
- stacks, 557
- STL (Standard Template Library), 1161
  - C++11, 1271-1273
- vectors, methods, 1278-1280

**containment, 785**

**continue statement, 280-282**

**conversion constructors, 769-770**

**conversion operators, explicitly, 1159-1160**

**convert.cpp, 57**

**converting**

- class type, 677
- rectangular coordinates to polar
  - coordinates, 348-351
- to standard C++, 1327
  - auto\_ptr template, 1333
  - C++ features, 1331
  - const instead of #define, 1327-1329
  - function prototypes, 1330
  - header files, 1331
  - inline instead of #define, 1329-1330
  - namespaces, 1331-1333
  - STL (Standard Template Library), 1334
  - string class, 1333
  - type casts, 1330-1331

**converting data types, 102, 606-612**

- applying automatically, 616-618
- conversion functions, 612-616
- conversion in arguments, 106
- conversion in expressions, 105-106
- conversion on assignment, 103-104
- friends, 618-621
- implicit conversion, 609
- type casts, 107-109, 606-612

**coordin.h, 449-450**

**coordinates**

- converting, 348-351
- polar coordinates, 347
- rectangular coordinates, 346

**copy constructable objects, 1008**

**copy constructors, 639, 767**

- deep copying, 642-644
- limitations, 640-642
- shallow copying, 640
- when to use, 639-640

**copy() function, 1293-1296**

- iterators, 1001-1002

**copy() method, 1269**

**copying**

- deep copying, 642-644
- shallow copying, 640
- strings, 135, 1269

**copying algorithms, 1036**

**copyit.cpp, 1004**

**copy\_backward() function, 1294-1297**

**count() function, 862, 1042, 1287, 1291**

**count() method, 1283**

**count.cpp, 1121**

**count\_if() function, 1287, 1291**

**counts() function, 862**

**cout object, 1067-1069**

- buffers, flushing, 1075-1076
- concatenation, 1071-1072
- field width display, 1080-1081
- fill characters, 1081-1082
- floating-point display precision, 1082-1083
- formatting data types, 1076-1078
- methods, 1071-1075
- number base display, 1078-1079
- printing trailing zeros/decimal points, 1083-1090

**cout statement, 36**

- concatenated output, 46-47

- cout.put() function, 83
- endl manipulator, 37-38
- integer values, displaying, 44-45
- \n newline character, 38-39
- cout.put() function, 83-84**
- covariance of return type, 744**
- cpp filename extension, 28**
- CRC cards, 1207**
- cstring header file, 123-124**
- ctime header file, 229**
- cube() function, 309, 312-313, 391**
- cubes.cpp, 390**
- cumulative totals, calculating, 1320**
- Customer class, 690-691, 694**
- cv-qualifiers, 472-473**
  - const, 473-474
  - volatile, 473
- c\_in\_str() function, 340-341**
- c\_str() method, 1252**

## D

---

- data hiding, 511-513, 523**
- data methods, 1251-1253**
- data objects, pointers, 161**
- data types, 507-508**
  - ADTs (abstract data types), 552-557
  - aliases, creating, 230
  - bool, 90
  - classifying, 97
  - compound types, 116
  - double, 50
  - floating-point numbers, 92
    - advantages/disadvantages, 96-97
    - constants, 96
    - decimal-point notation, 92
    - double, 94-96
    - E notation, 92-93
    - float, 94-96
    - long double, 94-96
  - integers, 68
    - char, 80-89
    - choosing integer types, 76-77
    - climits header file, 71-73
    - constants, 78-80, 90-92
    - initializing, 73
    - int, 68-70
    - long, 68-70
    - short, 68-70
    - sizeof operator, 71-73
    - unsigned, 74-76
    - width of, 68
  - recognized by, 1093-1095
  - type casts, 606-612
  - type conversion, 606-612
    - applying automatically, 616-618
    - conversion functions, 612-616
    - friends, 618-621
    - implicit conversion, 609
  - type conversions, 102
    - conversion in arguments, 106
    - conversion in expressions, 105-106
    - conversion on assignment, 103-104
    - type casts, 107-109
- data types, 140. See also compound types**
- Data() function, 820**
- data() method, 1251-1252**
- Dawes, Beman, 1205**
- dec manipulator, 1090-1091**
- dec manipulators, 1078-1079**
- decimal numbers, 1215**
- decimal points, trailing, 1083-1087, 1090**
- decision making, 253**
- declaration statements, 41-43**
- declaration-statement expressions, 202**
- declarations, 463**
  - aliases, 1157
  - auto, 109, 1155
  - decltype, 1156
  - external, 143
  - return types, 1157
- declarative region, 483**
- declaring**
  - arrays, 116-119
  - arrays of objects, 546
  - classes, 509-513, 522
  - constructors, 525-526
  - function pointers, 362-363
    - example, 364
    - invoking functions with, 363-364
  - pointers, 155-159, 171
  - two-dimensional arrays, 244-246
  - unions, 149
  - variables, 41-43
    - static, 183
  - vector class, 591-592

- decltype, 439**
  - declarations, 1156
- decorating names, 418**
- decrement operator (--), 207-208**
  - pointers, 210-211
  - postfixing, 209-210
  - prefixing, 209-210
- deep copying, 642-644**
- default arguments, 409-412**
- default class constructors, 527-528**
- default constructors, 638-639, 766-767**
- default type template parameters, 849**
- defaulted methods, classes, 1179-1180**
- defaults.cpp, 1077**
- dereferencing operator (\*), pointers, 171-172**
- #define directive, converting to standard C++**
  - const instead of #define, 1327-1329
  - inline instead of #define, 1329-1330
- defining**
  - class member functions, 514-516
  - classes, 47
  - constructors, 525-526
  - functions, 306-309
- defining declarations, 463**
- definitions, 463**
- delegating constructors, 1180-1181**
- delete operator, 163-164, 180-183, 400, 454, 476-477, 668**
- delete.cpp, 181**
- deleted methods, classes, 1179-1180**
- deque class templates, 1013**
- deque containers, 1013**
- deques, methods, 1278-1280**
- dequeue() method, 689**
- dereferencing (\*) operator, 155-159**
- dereferencing operators, 1242-1246**
- derived classes, 405**
  - constructors, 713-715
  - creating, 711-712
  - header files, 716
  - method definitions, 716
  - objects, creating, 717-718
  - relationships with base classes, 718-720
- derived types, 116**
- design**
  - bottom-up, 13
  - top-down, 12
- The Design and Evolution of C++*, 775, 1324**
- destructors, 524, 528-529, 538-539, 768**
  - class, 524
  - virtual destructors, 737, 742-743, 776
- difference\_type type, 1250, 1273**
- directives**
  - #define, converting to standard C++, 1327-1330
  - #ifndef, 451
  - #include, 33
  - using, 35-36, 59-60, 487-490
- displacement vectors, 589**
- divide-and-conquer strategy, 360-361**
- divide.cpp, 100**
- division operator (/), 100-101**
- dma.cpp, 762-764**
- dma.h, 761-762**
- do while loops, 231-233**
- dominance, virtual base classes, 828-829**
- double data type, 50, 94-96**
- double-ended queue, 1013**
- dowhile.cpp, 232**
- downcasting, 738**
- Draw() function, 818**
- dribble() function, 414**
- dynamic storage duration, 454**
- dynamic arrays, 172-173**
  - creating, 164-165
  - new operator, 164
  - sample program, 165-167
- dynamic binding, 164, 172-173, 737-740**
- dynamic cast operator, 934, 941-943**
- dynamic cast operators, 934-939**
- dynamic memory, 476-479, 482**
- dynamic memory allocation, 757**
  - auto\_ptr class, 969, 973-975
  - derived class does use new, 758-760
  - derived class doesn't use new, 757-758
  - example, 761-766
    - dma.cpp, 762-764
    - dma.h, 761-762
    - usedma.cpp, 765
- dynamic structures, creating, 178-180**
- dynamic\_cast operator, 943**
- dynamic variables, 454**

## E

early binding, 737

*Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition*, 1324

*Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, 1324

empty lines in strings, 130

empty() method, 1258, 1275

encapsulation, 512, 523

end() method, 981-984, 1251, 1275

end-of-file conditions, 237-241

endl manipulator, 37-38

enqueue() function, 890

entry-condition loops, 198

enum statement, 278-280

enum variables, 150-152

enumerators, 150-151

value ranges, 153

values, setting, 152

enum.cpp, 279

enumerations, 150-152

enumerators, 150-151

scoped, 1158

C++11, 551-552

value ranges, 153

values, setting, 152

enumerators, 150-151

as labels, 278-280

EOF (end-of-file) conditions, 237-241

eof() function, 238

eof() method, 296

eof() stream state methods, 1097-1102

eofbit stream state, 1097-1102

equal sign (=)

assignment operator (=), 644, 767-768, 772-775

custom definitions, 645-646

enumerator values, setting, 152

overloading, 652-658

potential problems, 645

strings, 133-134

structures, 145-146

when to use, 644

equality operator (==), compared to assignment operator, 218-220

equal() function, 1288-1292

equal.cpp, 219

equality operator (==), 217

compared to assignment operator, 218-220

equal\_range() function, 1024, 1304, 1309

equal\_range() method, 1283, 1285-1286

erase() method, 982-984, 1267, 1278, 1282

erasing strings, 1267-1268

error codes, returning, 898-900

error handling. *See* handling exceptions

error1.cpp, 897-898

error2.cpp, 899

error3.cpp, 901

error4.cpp, 906-907

error5.cpp, 910, 913

escape sequences, 84-85

estimate() function, 362-364

eternal loops, 232

exact matches, 432-434

exceed.cpp, 75

exception class, 917

exception handlers, 900, 933

exception handling, 896-897, 900, 933

abort() function, 897-898

bad\_alloc exceptions, 921

catching exceptions, 900, 916-917

cautions, 931-932

error codes, returning, 898-900

exception class, 917

exception handlers, 900

inheritance, 922-927

sales.cpp, 924

sales.h, 922

use\_sales.cpp, 925-927

invalid\_argument exceptions, 919

length\_error exceptions, 919

logic\_error exceptions, 918

objects as exceptions, 903-908

out\_of\_bounds exceptions, 919

range\_error exceptions, 919

real-world note, 933

runtime\_error exceptions, 919

stdexcept exception classes, 918-920

throwing exceptions, 900, 915-916

try blocks, 901-903

uncaught exceptions, 928-931

unexpected exceptions, 928-931

unwinding the stack, 909-910, 913-914

**exception specifications, C++11, 908**  
**exceptions, 1158**  
**exceptions() stream state method, 1098-1102**  
**exclamation point (!), logical NOT operator, 267-269**  
     alternative representations, 270  
     precedence, 269  
**exc\_mean.cpp, 905-906**  
**executable code, 18**  
**exit() function, 930**  
**explicit, 1159-1160**  
**explicit instantiations, 428-430, 850**  
**explicit keyword, 610**  
**explicit specializations, 425, 850-851**  
     example, 426-428  
     third-generation specialization, 425-426  
**exporting, templates, 1162**  
**express.cpp, 200**  
**expression arguments, 844**  
**expressions, 97, 200-202**  
     combining with comma operator, 214  
     compared to statements, 201  
     conditional operator (?:), 273-274  
     declaration-statement expressions, 202  
     logical AND (&&), 262  
         alternative representations, 270  
         example, 263-265  
         precedence, 269-270  
         ranges, 265-267  
     logical NOT (!), 267-269  
         alternative representations, 270  
         precedence, 269  
     logical OR (||), 260-262  
         alternative representations, 270  
         example, 261-262  
         precedence, 269  
     relational operators, 217-218, 220  
         C-style strings, comparing, 220-223  
         equality operator (==), 218-220  
         string class strings, comparing, 223-224  
         table of, 217  
     sequence points, 208-209  
     side effects, 201, 208-209  
     type conversions, 105-106  
**extern keyword, 467-472**  
     functions, 474

**external declarations, 143**  
**external linkage, 454**  
**external variables, 463, 466-467**  
**external.cpp, 465**  
**ext\_permutation() algorithm, 1038**

---

## F

**factorials, calculating, 203-205**  
**fail() method, 296, 1101**  
**fail() stream state method, 1098-1102**  
**failbit stream state, 1097-1102**  
**failbits, 130**  
**fields**  
     bit fields, 148  
     width, 1080-1081  
**file extensions, 20**  
**file I/O, 1114**  
     checking stream states, 1118-1119  
     command-line processing, 1119-1120  
     file modes, 1122  
         appending data to files, 1125-1127  
         binary, 1127-1133  
         constants, 1122-1123  
         opening files, 1124-1125  
         text, 1129  
     files, random access, 1133-1142  
     opening multiple files, 1119  
     reading, 1116-1118  
     writing, 1115-1118  
**file scope, 454**  
**file1.cpp, 451**  
**file2.cpp, 452**  
**filefunc.cpp, 406-407**  
**fileio.cpp, 1117**  
**files, 1129**  
     associating objects with, 289  
     client files, creating, 533-536  
     climits, 71-73  
     compiling separately, 447-449, 453  
     cpp filename extension, 28  
     ctime, 229  
     EOF (end-of-file) conditions, 237-240  
     header filenames, 34  
     header files, 448-449  
         converting to standard C++, 1331  
         creating, 530  
         cstring, 123-124  
         managing, 451

- implementation files, creating, 530
- include files, 96
- iostream, 33-34, 289, 1064, 1067
- text files, 287-288
  - reading, 292-298
  - writing to, 288-292
- file\_it() function, 408**
- fill characters, 1081-1082**
- fill() function, 1081, 1294, 1299**
  - array objects, 357
- fill.cpp, 1082**
- filling arrays, 326-327**
- fill\_array() function, 325-327, 331**
- fill\_n() function, 1294, 1299**
- fin.clear() function, 1121**
- final, 1183-1184**
- find() function, 1287-1289**
- find() method, 960-961, 965, 1260-1261, 1283**
- finding, 1260**
- find\_arr() function, 994**
- find\_end() function, 1287-1290**
- find\_first\_not\_of() method, 961, 1262-1263**
- find\_first\_of() function, 1287, 1290**
- find\_first\_of() method, 961, 1262**
- find\_if() function, 1287-1289**
- find\_last\_not\_of() method, 1263**
- find\_last\_of() method, 961, 1262**
- firstref.cpp, 383**
- fixed manipulator, 1091**
- flags, 1084**
- flags, setting, 1083**
- float data type, 94-96**
- floating points, display precision, 1082-1087, 1090**
- floating-point data types, default behavior, 1076**
- floating-point numbers, 92**
  - advantages/disadvantages, 96-97
  - constants, 96
  - decimal-point notation, 92
  - double data type, 94-96
  - E notation, 92-93
  - float data type, 94-96
  - long double data type, 94-96
- floatnum.cpp, 95**
- fltadd.cpp, 96**
- flush() function, 1076**
- flushing buffers, 1063**
- for loops**
  - blocks, 212-214
  - body, 196-197
  - combination assignment operators, 211-212
  - comma operator, 214-217
    - example, 214-216
    - precedence, 217
  - compared to while loops, 227-228
  - decrement operator (--), 207-208
    - pointers, 210-211
    - postfixing, 209-210
    - prefixing, 209-210
  - example, 196-197
  - expressions, 200-202
    - compared to statements, 201
    - declaration-statement expressions, 202
  - factorials, calculating, 203-205
  - increment operator (++), 207-208
    - pointers, 210-211
    - postfixing, 209-210
    - prefixing, 209-210
  - initialization, 196-197
  - loop test, 196-197
  - loop updates, 196-198, 205-206
  - nested loops, 244-249
  - nonexpressions, 202
  - range-based, 1161
  - sequence points, 208-209
  - side effects, 201, 208-209
  - step size, 205-206
  - strings, 206-207
  - syntax, 197-199
- for statements, declaration-statement expressions, 203**
- for-init-statement, 203**
- forcing moves, 1173-1174, 1177-1178**
- forever loops, 232**
- forloop.cpp, 196**
- formal arguments, 314**
- formatted input functions, 1094**
- formatting**
  - if else statement, 257-258
  - incore, 1142-1145



- source code, 39
  - source code style, 40
  - tokens, 39
  - white space, 39
- with `cout`, 1076-1077
  - field width display, 1080-1081
  - fill characters, 1081-1082
  - floating-point display precision, 1082-1083
  - manipulators, 1090-1091
  - number base display, 1078-1079
  - trailing zeros/decimal points, 1083-1090
- with `iomanip` header file
  - manipulators, 1091
- formore.cpp, 203-204**
- forstr1.cpp, 206**
- forstr2.cpp, 215**
- forward declaration, 884**
- forward iterators, 998**
- for\_each() function, 987-988, 1287-1289**
- for\_each() STL function, 991**
- fowl.cpp, 973**
- free memory storage, 182**
- free store, 454**
- free store (memory), 182-183**
- freeing memory, delete operator, 163-164**
- friend classes, 578-580, 877-888**
  - base-class friends, accessing, 801-804
  - compared to class member functions, 886
  - templates, 858
    - bound template friend functions, 861-864
    - non-template friend functions, 858-861
    - unbound template friend functions, 864-865
  - Tv class example, 878-879, 883
    - tv.cpp, 880-882
    - tv.h, 879-880
    - tvfm.h, 885-886
    - use\_tv.cpp, 882
- friend functions, 578-580**
  - creating, 579-580
  - type conversion, 618-621
- friend keyword, 579-580**
- friend member functions, 578-580, 883**
  - compared to friend classes, 886
  - example, 885-886
  - forward declaration, 884
  - shared friends, 888-889
- frnd2tmp.cpp, 860-861**
- front insert iterators, 1005-1007**
- front() method, 1278**
- funadap.cpp, 1034-1035**
- function idioms, arrays, 331**
- function objects, 1026**
- function parameter packs, 1197-1198**
  - unpacking, 1198-1199
- function parameters, reference variables, 386-390**
- function pointers, 1184-1188**
  - variations on, 365-370
- function polymorphism, 412**
- function prototype scope, 454**
- function prototypes, 309-311, 1330**
  - benefits, 312-313
  - C++ compared to ANSI C, 312
  - syntax, 311-312
- function wrapper**
  - fixing problems, 1194-1196
  - options for, 1196-1197
  - template inefficiencies, 1191-1194
- functional polymorphism, 564**
- functions, 18, 29, 48-49. See also names of specific functions**
  - adaptable binary, 1035
  - adaptable unary, 1035
  - algorithms, 1035
    - groups, 1035-1036
    - properties, 1036-1037
  - arguments, 31, 49, 53
    - arrays, 322-325
      - multiple, 314-320
    - passing by value, 313-314
    - two-dimensional arrays, 337-339
  - arrays, 320-321
    - as arguments, 322-325
    - const keyword, 327-328
    - design decisions, 325-326
    - displaying contents of, 327-328
    - examples, 328-331
    - filling, 326-327
    - modifying, 328

- pointers, 321–322
- ranges, 332–334
- two-dimensional arrays, 337–339
- binary, 1027, 1030
- body, 29
- C-style strings
  - passing as arguments, 339–341
  - returning, 341–343
- calling, 30, 49, 309, 311
- case sensitivity, 27, 32
- cin.get(), 317
- cin.get(ch), 317
- class member functions
  - const member functions, 537
  - constructors, 524–528, 538–539, 638–639, 659–661, 677–678
  - copy constructors, 639–644
  - definitions, 509, 514–516, 523
  - destructors, 528–529, 538–539
  - friend member functions, 883–889
  - implicit member functions, 637–638
  - inline functions, 517–518
  - invoking, 523
  - object membership, 518
  - private, 513
  - properties, 777–778
  - public, 513
  - qualified names, 514
  - this pointer, 539–546
  - unqualified names, 514
- compared to container methods, 1039–1041
- conversion functions, 677
- defining, 306–309
- definitions, 29
- formatted input, 1094
- friend functions, 578–580
  - creating, 579–580
  - type conversion, 618–621
- function prototypes, 1330
- headers, 29–31
- inline functions, 379–382
  - compared to macros, 382
  - square(), 381–382
- input, unformatted, 1102
- lambda functions, 1184
- language linking, 475–476
- library functions, 52
- linkage properties, 474–475
- non-member, 986–988, 991
- objects, returning, 662–664
  - const objects, 662–665
  - non-const objects, 663
- operator overloading, 587–588
- overloading, 237, 412–414, 564
  - example, 415–418
  - function signatures, 413
  - name decoration, 418
  - overload resolution, 431–438
  - when to use, 418
- pf()364
- pointers, 361–362
  - addresses of functions, obtaining, 362
  - const keyword, 334–336
  - declaring, 362–363
  - example, 364
  - invoking functions with, 363–364
  - pointers to pointers, 335
- prototypes, 50–52
- qualified names, 514
- recursion, 357
  - multiple recursive calls, 359–361
  - single recursive call, 358–359
- return addresses, 909
- return types, 30
- return values, 49
- set\_union(), 1020
- signatures, 413
- string class objects, 353–354
- structures, 343–346
  - passing/returning, 344–351
  - polar coordinates, 347
  - rectangular coordinates, 346
  - structure addresses, passing, 351–353
- templates, 419, 422
  - explicit instantiation, 428–430
  - explicit specializations, 425–428
  - implicit instantiation, 428–430
  - overload resolution, 431–438
  - overloading, 422–424
- transform(), 1031
- unary, 1027, 1030
- unqualified names, 514

- user-defined functions
  - example, 53-54
  - function form, 54-55
  - function headers, 55-56
  - return values, 57-59
  - using directive, 59-60
- virtual functions, pure virtual functions, 748
- void, 307
- functor.cpp, 1028**
- functors, 1026-1027, 1184-1188**
  - adaptable, 1032
  - concepts, 1027-1030
  - predefined, 1030-1032
    - equivalents for operators, 1032
- funtemp.cpp, 420**
- fun\_ptr.cpp, 364, 368**

---

## G

- g++ compiler, 22**
- gcount() member function, 1109-1114**
- generate() function, 1294, 1299**
- generate\_n() function, 1294, 1299**
- generators, 1027**
- generic programming, 14, 419, 951, 978, 992**
  - associative containers, 1018-1026
    - multimap, 1023-1025
    - set, 1019-1022
  - container concepts, 1007
    - container methods compared to functions, 1039-1041
    - properties, 1008-1010
    - sequence requirements, 1011-1012
  - container types
    - deque, 1013
    - list, 1014-1017
    - priority\_queue, 1017-1018
    - queue, 1017
    - stack, 1018
    - vector, 1012-1013
  - iterators, 992-997
    - back insert, 1005-1007
    - bidirectional, 998
    - concepts, 1000-1001
    - copy() function, 1001-1002
    - forward, 998
    - front insert, 1005-1007

- hierarchy, 999-1000
- importance of, 992-996
- input, 997-998
- insert, 1005-1007
- istream iterator template, 1003
- ostream iterator template, 1002-1003
- output, 998
- pointers, 1001
- random access, 999
- reverse, 1003-1005
- types, 997
- get() function, 127-130**
- get() function (cin), 235-237, 241-244**
- get() member function, 1102-1108**
- Get() method, 821**
- getinfo.cpp, 45**
- getline() function, 126-127, 957-960**
- getline() member function, 1106-1108**
- getline() method, 509, 1270**
- getname() function, 180-182**
- get\_allocator() method, 1252**
- get\_fun.cpp, 1107**
- global namespaces, 484**
- global scope, 454**
- global variables, compared to local variables, 467**
- good() method, 294**
- good() stream state method, 1097-1102**
- goodbit stream state, 1097-1102**
- gpp compiler, 22**
- greater than () operator, 217**
- greater than or equal to () operator, 217**

---

## H

- handling exceptions, 896**
- hangman.cpp, 962-965**
- hardware, program portability, 16**
- harmonic mean, 896**
- harpo() function, 410**
- has-a relationships, 721, 788**
- header files, 448-449**
  - limits, 71-73
  - converting to standard C++, 1331
  - creating, 530
  - cstring, 123-124
  - ctime, 229
  - iomanip, manipulators, 1091
  - iostream, 289
  - managing, 451

**headers, filenames, 34****headers (function), 29-31, 55-56****heap operations**

- make\_heap() function, 1305, 1314
- pop\_heap() function, 1305, 1314
- push\_heap() function, 1305, 1314
- sort\_heap() function, 1305, 1315

**heaping**

- popping values off, 1314
- pushing values onto, 1314

**heaps**

- creating, 1314
- defined, 1314
- heap operations
  - make\_heap() function, 1305, 1314
  - pop\_heap() function, 1305, 1314
  - push\_heap() function, 1305, 1314
  - sort\_heap() function, 1305, 1315
- sorting, 1315

**heaps (memory), 182****hex manipulator, 1090-1091****hex manipulators, 1078-1079****hexadecimal numbers, 1216**

- binary equivalents, 1217-1218

**hexoct2.cpp, 79****hierarchy, iterators, 999-1000****high-level languages, 11****history of C++, 10-15**

- C language
  - development history, 11
  - programming philosophy, 11-13
- generic programming, 14
- OOP, 13-14

**hmean() function, 898-905**

|

**I/O (input/output), 1062, 1270**

- buffers, 1063-1067
- redirecting, 1067-1068
- streams, 1063-1067
- text files, 287-288
  - reading, 292-298
  - writing to, 288-292

**identifiers, special meanings, 1223****IDEs (integrated development environments), 19****if else statement, 255**

- example, 256-257
- formatting, 257-258
- if else if else construction, 258-260
- syntax, 255

**if statement, 254**

- bug prevention, 260
- example, 255
- syntax, 254

**if.cpp, 255****ifelse.cpp, 257****ifelseif.cpp, 259****#ifndef directive, 451****ifstream objects, 1116-1119****ignore() member function, 1106-1108****ilist.cpp, 1053****imbuing, I/O with styles, 1077****implementation, changing, 521-522****implementation files, creating, 530****implicit conversion, 609****implicit instantiation, 428-430****implicit instantiations, 850****implicit keyword, 610****implicit member functions, 637-638****implicit upcasting, 807****in-class initialization, 1160****in-place algorithms, 1036****include (#include) directive, 33****include files, 96****includes() function, 1305, 1311****incore formatting, 1142-1145****increment operator (++), 197, 207-208**

- pointers, 210-211
- postfixing, 209-210
- prefixing, 209-210

**indeterminate values, 73****indexes, 117****indirect values, 155****inequality operator (!=), 217****inheritance**

- dynamic memory allocation, 757
  - derived class does use new, 758-760
  - derived class doesn't use new, 757-758
- example, 761-766
- exceptions, 922-927
- sales.cpp, 924

- sales.h, 922
- use\_sales.cpp, 925-927
- references, 405-408
- inheritance (class), 708**
  - ABCs (abstract base classes), 746-749
    - ABC philosophy, 756
    - AcctABC example, 749-755
    - enforcing interface rules with, 757
  - assignment operators, 772-775
  - base classes
    - relationships with derived classes, 718-720
    - TableTennisPlayer example, 708-710
  - Constructors, 713-715
  - derived classes
    - creating, 711-712
    - header files, 716
    - method definitions, 716
    - objects, creating, 717-718
    - relationships with base classes, 718-720
  - has-a relationships, 721
  - is-a relationships, 720-722, 772
  - multiple inheritance, 798, 808-830
    - virtual base classes, 815-829
    - Worker class example, 810-814
  - polymorphic public inheritance, 722-723
    - base-class functions, 777
    - Brass class declaration, 723-726
    - Brass class implementation, 727-731
    - Brass class objects, 732-733
    - BrassPlus class declaration, 723-726
    - BrassPlus class implementation, 727-731
    - BrassPlus class objects, 732-733
    - constructors, 742
    - dynamic binding, 737-740
    - pointer compatibility, 737-739
    - reference type compatibility, 737-739
    - static binding, 737-740
    - virtual destructors, 737, 742-743, 776
    - virtual functions, 734-736, 739-745, 775-776
  - private inheritance, 797
    - base-class components, initializing, 798-799
    - base-class friends, accessing, 801-804
    - base-class methods, accessing, 800-801
    - base-class objects, accessing, 801
    - compared to containment, 806
    - Student class example, 798-805
  - protected classes, 745-746, 775
  - protected inheritance, 806-807
  - public, 806
    - multiple, 826
  - what's not inherited, 772
- inheriting**
  - constructors, 1181-1183
  - delegating, 1181-1183
- initialization, 70**
  - arrays, 117-120
  - arrays of objects, 546
  - automatic variables, 458
  - base-class components, 798-799
  - contained objects, 791
  - for loops, 196-197
  - pointers, 157-159
  - reference variables, 385
  - strings, 121
  - two-dimensional arrays, 246-249
- initialization lists, 119**
  - constructors, C++11, 1258
- initializer\_list, 1053-1054**
  - uniform initialization, 1155
- initializer\_list template, C++11, 1051-1053**
- initializing**
  - arrays, C++11, 120
  - variables, 52, 73
- init\_ptr.cpp, 158**
- inline functions, 379, 517-518**
  - compared to macros, 382
  - square(), 381-382
- inline modifier as alternative to #define, 1329-1330**
- inline qualifier, 517**
- inline.cpp, 381**
- inner\_product() function, 1320-1321**
- inplace\_merge() function, 1305, 1311**

**input, 46**

- cin object, 1093-1095
  - operator overloading, 1095-1097
  - stream states, 1097-1102
- cin statement, 46
- classes, string, 957-960
- istream class, methods, 1109-1114
- single-character, 1102-1106
- strings, 1106-1108

**input functions**

- formatted, 1094
- unformatted, 1102

**input iterators, 997-998****input/output, strings, 287, 1269-1270****insert iterators, 1005-1007****insert() method, 983-984, 1015-1016, 1267, 1277, 1282****inserting strings, 1267****inserts.cpp, 1006****instances, 511****instantiation, 832-836****instantiation**

- explicit, 428-430, 850
- implicit, 428-430, 850

**instr1.cpp, 125****instr2.cpp, 127****instr3.cpp, 129****int data type, 68-70****int main() function header, 30-31****integer values, displaying with cout, 44-45****integers, 68**

- bool, 90
- char, 80-87
  - escape sequences, 84-87
  - signed char, 88-89
  - universal character names, 87-88
  - unsigned char, 88-89
  - wchar\_t, 89
- choosing integer types, 76-77
- climits header file, 71-73
- constants, 78-80
  - const keyword, 90-92
  - symbolic names, 90-92
- initializing, 73
- int, 68-70
- long, 68-70
- pointers, 160
- short, 68-70

sizeof operator, 71-73

unsigned, 74-76

width of, 68

**integrated development environments, 19**  
**interfaces**

- contained objects, 792-795
- defined, 509-510
- public interfaces, 509

**internal linkage, 454****internal manipulator, 1091****internal variables, 467-470****International Standards Organization (ISO), C++ standard, 16****Internet resources, 1325****INT\_MAX constant, 72****INT\_MIN constant, 72****invalid\_argument exception, 919****invoking, 526****iomanip.cpp, 1092****ios class, 1065****iostream class, 1065****iostream file, 33-34, 1064, 1067****iostream header file, 289****ios\_base class, 1065**

constants representing bit values, 1085

**is-a relationships, 720-722, 772, 808****isalnum() function, 272****isalpha() function, 272****isblank() function, 272****iscntrl() function, 272****isdigit() function, 272****isempty() function, 685****isfull() function, 685****isgraph() function, 272****islower() function, 272****ISO (International Standards Organization), C++ standard, 16****ISO 10646, 88****isprint() function, 272****ispunct() function, 272****isspace() function, 272, 1101****istream class, 47, 1065**

data types recognized, 1093-1095

input

- methods, 1109-1114
- single-character, 1102-1106
- strings, 1106-1108

**istream iterator template, 1003****isupper() function, 272**

**isxdigit() function, 273**  
**is\_open() method, 294, 1118-1119, 1125-1127**  
**iterator type, 1273**  
**iterators, 981-982, 992, 997**  
   back insert, 1005-1007  
   bidirectional, 998  
   concepts, models, 1000-1001  
   copy() function, 1001-1002  
   forward, 998  
   front insert, 1005-1007  
   hierarchy, 999-1000  
   importance of, 992-996  
   input, 997-998  
   insert, 1005-1007  
   istream iterator template, 1003  
   ostream iterator template, 1002-1003  
   output, 998  
   pointers, 1001  
   random access, 999  
   reverse, 1003-1005  
   types, 997  
**iter\_swap() function, 1294**

---

## J–K

---

**jump.cpp, 280-281**

**K&R (Kernighan and Ritchie) C standard, 17**

**keywords, 56. See also statements**  
   auto, 472  
   catch, 900  
   class, 831  
   const, 90-92, 473-474, 771-772, 1327-1329  
     arrays, 327-328  
     pointers, 334-336  
     reference variables, 401  
     temporary variables, 392-394  
   decltype, 439  
   explicit, 610  
   extern, 467-472  
     functions, 474  
   friend, 579-580  
   implicit, 610  
   inline, 517, 1329-1330  
   mutable, 472-473  
   namespace, 483-486

  private, 511-513, 798  
   protected, 745-746, 775, 806  
   public, 511-513  
   register, 472  
   static, 183, 472  
     functions, 475  
   struct, 140  
   table of, 1221  
   template, 831  
   throw, 900  
   try, 901  
   typedef, 230  
   typename, 831  
   using, 486-490, 807-808, 1332  
   virtual, 742  
   volatile, 473  
**key\_comp() method, 1282-1285**  
**key\_compare type, 1281-1284**  
**key\_type type, 1281-1284**

---

## L

---

**labels, enumerators as, 278-280**  
**lambda functions, 1184-1188**  
   reasons for, 1188-1191  
**language divergence, 16**  
**language linking, 475-476**  
**languages, evolution of, 1205**  
   Boost, 1205-1207  
   Technical Report, 1206  
**last in-first out (LIFO) stacks, 459**  
**late binding, 737**  
**layering, 785**  
**leaks (memory), 183**  
**Lee, Meng, 978**  
**left manipulator, 1091**  
**left shift operator (<), 1235**  
   overloading, 581-587, 676  
**left() function, 409-410, 415-418**  
**left.cpp, 410-411**  
**leftover.cpp, 416-417**  
**length() functions, 960**  
**length() method, 1249-1251**  
**length\_error exception, 919**  
**lessthanlessthan (left shift operator), overloading, 581-587, 676**  
**lessthansignlessthansign (left shift operator), 1236**

**lessthansignlessthansignequalsign (left shift and assign) operator, 1236**

**lexicographical\_compare() function, 1306, 1318**

**libraries, 18, 378**

- C++11, 1203
- cctype, 270-273
- multiple library linking, 453
- STL (Standard Template Library), 1334

**library functions, 52**

**LIFO (last in-first out) stacks, 459**

**limits.cpp, 70**

**linear time, 1009**

**linear time complexity, 1009-1010**

**linkage**

- external, 454
- functions, 474-475
- internal, 454
- language linking, 475-476
- static variables
  - external linkage, 463-467
  - internal linkage, 467-470
  - no linkage, 470-472

**linked lists, 680**

**linking multiple libraries, 453**

**Linux, g++ compiler, 22**

**list class templates, 1014-1017**

- member functions, 1014-1016

**list containers, 1014-1017**

- member functions, 1014-1016

**list initialization, C++11, 537**

**list.cpp, 1015-1016**

**listmv.cpp, 1039-1040**

**lists**

- linked lists, 680
- methods, 1278-1280

**literal operators, 1204**

**Little Endian, 1218**

**local scope, 454**

- variables, 455-457

**local variables, 314-315**

- compared to global variables, 467

**logical AND operator (&&), 262**

- alternative representations, 270
- example, 263-265
- precedence, 269-270
- ranges, 265-267

**logical bitwise operators, 1237-1240**

**logical NOT operator (!), 267-269**

- alternative representations, 270
- precedence, 269

**logical operators, 260**

- alternative representations, 270
- AND (&&), 262
  - example, 263-265
  - precedence, 269-270
  - ranges, 265-267
- function equivalents, 1031-1032
- NOT (!), 267-269
- OR (||), 260-262
  - Example, 261-262

**logical OR operator (||), 260-262**

- alternative representations, 270
- example, 261-262
- precedence, 269

**logic\_error exception, 918**

**long data type, 68-70**

**long double data type, 94-96**

**long long type, 1153**

**LONG\_MAX constant, 72**

**LONG\_MIN constant, 72**

**loops, 195**

- break statement, 280-282
- continue statement, 280-282
- do while, 231-233
- entry-condition loops, 198
- for loops
  - blocks, 212-214
  - body, 196-197
  - combination assignment operators, 211-212
  - comma operator, 214-217
  - compared to while loops, 227-228
  - decrement operator ( $\text{--}$ ), 207-211
  - example, 196-197
  - expressions, 200-202
  - factorials, calculating, 203-205
  - increment operator ( $\text{++}$ ), 207-211
  - initialization, 196-197
  - loop test, 196-197
  - loop updates, 196-198, 205-206
  - nonexpressions, 202
  - sequence points, 208-209
  - side effects, 201, 208-209
  - step size, 205-206



- strings, 206–207
- syntax, 197–199
- forever loops, 232
- nested loops, 244–249
- number-reading loops, 283–286
- range-based, C++11, 233–234
- text input, cin object, 234–235
  - cin.get() function, 235–237, 241–244
  - end-of-file conditions, 237–241
  - sentinel characters, 234
- while loops, 224–227
  - compared to for loops, 227–228
  - example, 225–226
  - syntax, 224
  - time-delay loops, 229–230
- lotto probabilities, calculating, 317–320
- lotto.cpp, 319
- low-level languages, 11
- low-level programming, 1203–1204
- lower\_bound() function, 1024, 1304, 1309
- lower\_bound() method, 1021, 1283
- lvalue reference, 1162

---

## M

- machine language, definition of, 18
- Macintosh, C++, 25
- macros, compared to inline functions, 382
- magval() method, 602
- main() function, 29–30
  - calling, 30
  - importance of, 32
  - int main() header, 30–31
- make\_heap() function, 1305, 1314
- malloc() function, 160
- mangling names, 418
- manip.cpp, 1079
- manipulators, 38, 1090–1091
  - endl, 37–38
  - iomanip header file, 1091
  - number base display, 1078–1079
- mantissas, 93
- manyfrnd.cpp, 865
- mapped\_type type, 1281, 1284
- maps, methods, 1281–1284
- math operators, 97. *See also* arithmetic operators
- max() function, 1305, 1316
- max() method, 787, 1046
- maxsize() method, 1275
- max\_element() function, 1305, 1317
- max\_size() method, 1251
- mean, harmonic, 896
- means() function, 909–914
- member dereferencing operators, 1242–1246
- member functions. *See also* constructors
  - const member functions, 537
  - constructors, 524, 538–539
    - calling, 526–527
    - declaring, 525–526
    - default constructors, 527–528, 638–639
    - defining, 525–526
    - new operator, 659–661, 677–678
  - copy constructors, 639
  - deep copying, 642–644
  - limitations, 640–642
  - shallow copying, 640
  - when to use, 639–640
- definitions, 509, 514, 516, 523
- destructors, 528–529, 538–539
- friend member functions, 578–580, 883
  - compared to friend classes, 886
  - example, 885–886
  - forward declaration, 884
  - shared friends, 888–889
- implicit member functions, 637–638
- inline functions, 517–518
- invoking, 523
- object membership, 518
- objects, returning, 662–664
  - const objects, 662–665
  - non-const objects, 663
- private, 513
- properties, 777–778
- public, 513
- qualified names, 514
- template classes, list, 1014–1016
- this pointer, 539–546
- unqualified names, 514

- member in-class initialization, 1160
- member initializer lists, 683, 715
- member templates, 854–855
- members, structures, 141
- memberwise assignment, 145

**memberwise copying, 640**

**memb\_pt.cpp, 1244-1245**

**memory. See also buffers**

allocating

bad\_alloc exceptions, 921

new operator, 160-162

automatic storage, 182

cv-qualifiers, 472-474

dynamic, 476-482

dynamic memory allocation, 757

derived class does use new, 758-760

derived class doesn't use new,

757-758

example, 761-766

free store, 182-183

freezing delete operator, 163-164

function linkage, 474-475

language linking, 475-476

leaks, 183

memory-related methods, 1258

multifile programs, compiling separately,

447-453

named, 160

stack, 458-459

unwinding, 909-914

static storage, 183

storage class specifiers, 472-473

storage duration, 453-454

automatic variables, 455-459

scope and linkage, 454

static variables, 459-463, 466-472

storage methods, 182

**memory allocation, dynamic (auto\_ptr class), 969, 973-975**

**memory leaks, 163**

**merge() function, 1305, 1310-1311**

**merge() method, 1016-1017, 1280**

**merging**

inplace\_merge() function, 1305, 1311

merge() function, 1305, 1310-1311

**methods. See also specific methods**

base-class methods, accessing, 800-801

defaulted and deleted methods, classes,

1179-1180

end(), 984

inheritance, multiple, 826

insert(), 1015-1016

STL, 1161

virtual base classes, 818-828

**MI (multiple inheritance), 798**

**min() function, 1305, 1316**

**min() method, 787, 1046**

**minimum values**

finding 1316-1317

**minus sign (-), decrement operator (--), 207-208**

pointers, 210-211

postfixing, 209-210

prefixing, 209-210

**min\_element() function, 1305, 1317**

**mismatch() function, 1288, 1291**

**mixtypes.cpp, 185**

**models, concepts of iterators, 1000-1001**

**modifiers**

const, as alternative to #define,

1327-1329

inline, as alternative to #define,

1329-1330

**modulus operator (%), 101-102**

**modulus.cpp, 102**

**morechar.cpp, 82**

**more\_and.cpp, 266**

**move assignment operator, 1173**

**move constructors, 1165**

**move semantics, 1164-1171**

observations, 1171-1172

**Move() method, 748**

**moves, forcing, 1173-1178**

**MS-DOS, gpp compiler, 22**

**multifile programs, compiling separately, 447-453**

**multimap associative containers, 1023-1025**

**multiple arguments, 314-320**

n\_chars() example, 314-317

probability() example, 318-320

**multiple class representations, 599**

**multiple inheritance, 798, 808-809, 829-830**

virtual base classes, 815-817

combining with nonvirtual base

classes, 828

constructors, 817-818

dominance, 828-829

methods, 818-828

Worker class example, 810-814

**multiple library linking, 453**

**multiple public inheritance, methods, 826**

**multiple type parameters, 847**

**multiplication operator (\*), overloading, 574-578****multisets, set operations**

- includes() function, 1311
- set\_difference() function, 1313
- set\_intersection() function, 1312
- set\_union() function, 1312

**multimap.cpp, 1024-1025****mutable keyword, 472-473****mutating sequence operations**

- copy() function, 1293-1296
- copy\_backward() function, 1294-1297
- fill() function, 1294, 1299
- fill\_n() function, 1294, 1299
- generate() function, 1294, 1299
- generate\_n() function, 1294, 1299
- iter\_swap() function, 1294
- partition() function, 1295, 1302-1303
- random\_shuffle() function, 1295, 1302
- remove() function, 1295, 1299
- remove\_copy() function, 1295, 1300
- remove\_copy\_if() function, 1295, 1300
- remove\_if() function, 1295, 1300
- replace() function, 1294, 1298, 1302
- replace\_copy() function, 1294, 1298
- replace\_copy\_if() function, 1294, 1298
- replace\_if() function, 1294, 1298
- reverse() function, 1295
- reverse\_copy() function, 1295, 1301
- rotate() function, 1295, 1301
- rotate\_copy() function, 1295, 1302
- stable\_partition() function, 1295, 1303
- swap() function, 1294, 1297
- swap\_ranges() function, 1294, 1297
- transform() function, 1294, 1297
- unique() function, 1295, 1300
- unique\_copy() function, 1295, 1301

**myfirst.cpp program, 27-29**

- comments, 32-33
- header filenames, 34
- iostream file, 33-34
- main() function, 29-30
  - calling, 30
  - importance of, 32
  - int main() header, 30-31
- namespaces, 35-36

- output, displaying with cout, 36
  - endl manipulator, 37-38
  - \n newline character, 38-39
- source code formatting, 39
  - source code style, 40
  - tokens, 39
  - white space, 39

**mytime0.h, 566****mytime1.cpp, 569-570****mytime1.h, 569****mytime2.cpp, 575****mytime2.h, 575**


---

**N**


---

**name decoration, 418****name mangling, 418****named memory, 160****names**

- aliases, creating, 230
- array names, 172
- function qualified names, 514
- function unqualified names, 514
- name decoration, 418
- namespace aliases, 491
- reserved names, 1222-1223

**namesp.cpp, 493-494****namesp.h, 493****namespace keyword, 483-486****namespace scope, 454****namespaces, 35-36, 482-483, 1331-1333**

- aliases, 491
- creating, 483-486
- declarative region, 483
- example, 492-496
  - namesp.cpp, 493-494
  - namesp.h, 493
  - usenmsp.cpp, 494-495
- global, 484
- guidelines, 496-497
- nesting, 490-491
- open, 485
- potential scope, 483
- std, 59
- unnamed, 491-492
- using declaration, 486-490
- using directive, 487-490
- using-declaration, 491
- using-directive, 491

**naming conventions, 60**

- header files, 34
- source files, 20
- symbolic names, 90-92
- universal character names, 87-88
- variables, 66-68

**narrowing uniform initialization, 1154****navigating files, 133-1141**

- temporary files, 1141-1142

**nested classes, 682, 889-891**

- access control, 892
- scope, 891-892
- templates, 892-896

**nested loops, 244-249****nested structures, 682****nested.cpp, 247, 895-896****nesting namespaces, 490-491****new, 921****new operator, 180-182, 454, 476-479, 482, 668**

- bad\_alloc exceptions, 921
- constructors, 659-661, 677-678
- dynamic arrays, 164-167
- dynamic structures, 178-180
- free store, 183
- memory allocation, 160-162
- placement new, 671-676
- reference variables, 400

**newline character (\n), 38-39****newstrct.cpp, 179-180****next\_permutation() algorithm, 1039****next\_permutation() function, 1306, 1319****noboolalpha manipulator, 1090****noexcept, C++11, 1248****non-const objects, returning**

- references to, 663

**non-member functions, 986-991****non-type arguments (arrays), 843-845****nonexpressions, for loops, 202****nonmodifying sequence operations, 1286**

- adjacent\_find() function, 1287, 1290
- count() function, 1287, 1291
- count\_if() function, 1287, 1291
- equal() function, 1288, 1291-1292
- find() function, 1287-1289
- find\_end() function, 1287-1290
- find\_first\_of() function, 1287, 1290
- find\_if() function, 1287-1289

- for\_each() function, 1287-1289
- mismatch() function, 1288, 1291
- search() function, 1288, 1292-1293
- search\_n() function, 1288, 1293

**noshowbase manipulator, 1090****noshowpoint manipulator, 1091****noshowpos manipulator, 1091****NOT operators, logical NOT (!), 267-269**

- alternative representations, 270
- precedence, 269

**not.cpp, 267-268****nouppercase manipulator, 1091****nth\_element() function, 1304, 1308, 1315****null characters, 121****null pointers, 163****nullptr, 1158****number base display, 1078-1079****number-reading loops, 283-286****numbers, 1215**

- ASCII character set, table of, 1225-1229
- Big Endian/Little Endian, 1218
- binary numbers, 1217
  - hexadecimal equivalents, 1217-1218
- decimal numbers, 1215
- factorials, calculating, 203-205
- floating-point numbers, 92
  - advantages/disadvantages, 96-97
  - constants, 96
  - decimal-point notation, 92
  - double data type, 94-96
  - E notation, 92-93
  - float data type, 94-96
  - long double data type, 94-96
- harmonic mean, 896
- hexadecimal numbers, 1216
  - binary equivalents, 1217-1218
- integers, 68
  - bool, 90
  - char, 80-89
  - choosing integer types, 76-77
  - limits header file, 71-73
  - constants, 78-80, 90-92
  - initializing, 73
  - int, 68-70
  - long, 68-70
  - short, 68-70
  - sizeof operator, 71-73
  - unsigned, 74-76
  - width of, 68

number-reading loops, 283-286  
 octal numbers, 1215-1216  
 pointers, 160  
 pseudorandom numbers, 605  
**numeric operations, 1319-1320**  
 accumulate() function, 1320  
 adjacent\_difference() function,  
 1321-1322  
 inner\_product() function, 1320-1321  
 partial\_sum() function, 1321  
**numstr.cpp, 130**  
**num\_test.cpp, 198**  
**n\_chars() function, 314-317**

---

## O

**object code, definition of, 18**  
**object types, 47**  
*Object-Oriented Analysis and Design, Second Edition, 1323*  
**object-oriented programming. See OOP (object-oriented programming)**  
**objects, 511, 786**  
 arrays, 355, 546-549  
   declaring, 546  
   example, 547-549  
   fill function, 357  
   initializing, 546  
 as exceptions, 903-908  
 assignable, 1008  
 associating with files, 289  
 base-class objects, accessing, 801  
 cerr, 1067  
 cin, 1067, 1093-1095  
   cin.get() function, 235-237, 241-244  
   get() function, 128-130  
   getline() function, 126-127  
   loops, 234-235  
   operator overloading, 1095-1097  
   stream states, 1097-1102  
 class, 788  
 clog, 1067  
 contained objects  
   compared to private inheritance, 806  
   initializing, 791  
   interfaces, 792-795  
 copy constructable, 1008  
 cout, 1067-1069  
   concatenation, 1071-1072  
   field width display, 1080-1081  
   fill characters, 1081-1082  
   floating-point display precision,  
   1082-1083  
   flushing buffers, 1075-1076  
   formatting data types, 1076-1078  
   methods, 1071-1075  
   number base display, 1078-1079  
   printing trailing zeros/decimal  
   points, 1083-1090  
 creating, 523  
 defined, 13, 36  
 functions. *See* function objects  
 ifstream, 1116-1119  
 ofstream, 1115-1119  
 ostream, 1142-1145  
 passing by reference, 770  
 passing by value, 770  
 pointers, 665-670  
 reference variables, 401-405  
 returning, 662-664, 770-771  
   const objects, 662-665  
   non-const objects, 663  
 stream, 1067  
 string, 965  
 subobjects, 797  
 this pointer, 539-546  
 valarray, 1045-1051  
 vector, 979-991, 1045-1051  
   adding elements to, 982-983  
   past-the-end iterators, 981-982  
   removing ranges of, 982  
   shuffling elements in, 987  
   sorting, 987  
   vect1.cpp example, 980-981  
   vect2.cpp sample program, 984-986  
   vect3.cpp sample program,  
   988-991  
**oct manipulators, 1078-1079, 1090-1091**  
**octal numbers, 1215-1216**  
**ofstream objects, 290, 1115-1119**  
**one definition rule, 475**  
**one-dimensional arrays, 244**

**OOP (object-oriented programming), 13, 506-507, 512, 1207**

- classes, 47-48
- client/server model, 520
- overview, 13-14

**open namespaces, 485****open() method, 291-293, 967, 1119, 1122-1125****opening files, 1124-1125**

- multiple, 1119

**operands, 97****operator functions, 565****operator overloading, 37, 564-566**

- addition operator (+), 569-572
- cin object input, 1095-1097
- example, 565-569
  - mytime0.cpp, 566
  - mytime0.h, 566
  - usetime0.cpp, 568-569
- left shift operator (<), 581-587

- member versus nonmember functions, 587-588

- multiplication operator (\*), 574-578

- operator functions, 565

- operator\*(), 574-578
- operator+(), 569-572
- operator-(), 574-578
- operator<<(), 581-585, 587

- restrictions, 573-574

- subtraction operator (-), 574-578

- vector class, 588-590, 600

- adding vectors, 590
- declaring, 591-592
- displacement vectors, 589
- implementation comments, 602
- member functions, 592, 597
- multiple representations, 599
- overloaded arithmetic operators, 599-600

- overloading overloaded operators, 601

- Random Walk sample program, 602, 605-606

- state members, 597-599

- with classes, string, 965

**operator\*() function, 574-578****operator+() function, 569-572****operator+() method, 1266****operator-() function, 574-578****operators, 1235**

- addition operator (+), overloading, 569-572

- alternative tokens, 1222

- arithmetic operators, 97-99

- associativity, 99-100
- division (/), 100-101
- modulus (%), 101-102
- order of precedence, 99-100

- assignment (=), 43-44, 644, 767-768, 772-775

- custom definitions, 645-646
- enumerator value ranges, 153
- enumerator values, setting, 152
- overloading, 652-658
- potential problems, 645
- strings, 133-134
- structures, 145-146
- when to use, 644

- associativity, 1231

- examples, 1234
- table of, 1232-1234

- binary operators, 601, 1234

- bitwise operators, 1235

- alternative representations, 1240
- logical bitwise operators, 1237-1240
- shift operators, 1235-1237
- testing bit values, 1241-1242
- toggleing, 1241
- turning bits off, 1241
- turning bits on, 1241

- combination assignment operators, 211-212

- comma, 214-217

- example, 214-216
- precedence, 217

- concatenation (+), strings, 133-134

- conditional (?:), 273-274

- const\_cast, 944

- decrement (—), 207-208

- pointers, 210-211
- postfixing, 209-210
- prefixing, 209-210

- deferencing (\*), pointers, 171-172
- defined, 70

- delete, 163-164, 180-183, 400, 454, 476-477, 668

- dereferencing (\*), 155-159

- dynamic cast, 934-943

- functor equivalents for arithmetic,
  - logical, and relational operators, 1031-1032
- increment (++), 197, 207-208
  - pointers, 210-211
  - postfixing, 209-210
  - prefixing, 209-210
- left shift operator (<<)
  - overloading, 581-587, 676
- literal operators, 1204
- member dereferencing operators, 1242-1246
- multiplication operator (\*), overloading, 574-578
- new, 180-182, 454, 476-482, 668
  - bad\_alloc exceptions, 921
  - constructors, 659-661, 677-678
  - dynamic arrays, 164-167
  - dynamic structures, 178-180
  - free store, 183
  - memory allocation, 160-162
  - placement new, 671-676
  - reference variables, 400
- operator functions, 565
  - operator\*(), 574-578
  - operator+(), 569-572
  - operator-(), 574-578
  - operator<<(), 581-587
- overloading, 101, 564-565
  - addition operator (+), 569-572
  - assignment operator, 652-658
  - example, 565-569
  - left shift operator (<<), 581-587, 676
  - member versus nonmember functions, 587-588
  - multiplication operator (\*), 574-578
  - operator functions, 565
  - overloading overloaded operators, 601
  - restrictions, 573-574
  - subtraction operator (-), 574-578
- precedence, 1231
  - examples, 1234
  - table of, 1232-1234
- reference operator (&), 383-386
- reinterpret\_cast, 946
- relational operators, 217-220
  - C-style strings, comparing, 220-223
  - equality operator (==), 218-220
  - string class strings, comparing, 223-224
  - table of, 217
- scope resolution (::), 467
- scope-resolution (::), 514
- scope-resolution operator, 1332
- sizeof, 71-73
- static\_cast, 945-946
- subtraction operator (-), overloading, 574-578
- type cast, 943-944
- type info structure, 934
- typeid, 934, 939-944
- unary minus, 601
- unary operators, 601, 1234
- operator[]() method, 787, 1259, 1283-1286**
- OR operators**
  - bitwise OR (|), 1237-1238
  - logical OR (||), 260-262
    - alternative representations, 270
    - example, 261-262
    - precedence, 269
- or.cpp, 261**
- ordering**
  - strict weak, 988
  - total, 988
- ostream class, 47, 1065**
- ostream iterator template, 1002-1003**
- ostream methods, 1071-1075**
- ostreamstring class, 1142-1145**
- ourfunc.cpp, 54**
- outfile.cpp, 290-291**
- output**
  - buffers, flushing, 1075-1076
  - classes, ostream, 1070-1075
  - concatenating, 46-47, 1071-1072
  - cout
    - field width display, 1080-1081
    - fill characters, 1081-1082
    - floating-point display precision, 1082-1083
    - formatting data types, 1076-1078
    - number base display, 1078-1079
    - printing trailing zeros/decimal points, 1083-1090

- cout object, 1069
- displaying with cout, 36
  - concatenated output, 46-47
  - endl manipulator, 37-38
  - integer values, 44-45
  - \n newline character, 38-39
- output iterators, 998**
- out\_of\_bounds exception, 919**
- overload resolution, 431-432**
  - best matches, 432-434
  - exact matches, 432-434
  - multiple arguments, 438
  - partial ordering rules, 434-436
- overloading**
  - functions, 237, 412-414, 564
    - example, 415-418
    - function signatures, 413
    - name decoration, 418
    - overload resolution, 431-438
    - when to use, 418
  - operators, 101, 564-565
    - addition operator (+), 569-572
    - assignment operator, 652-658
    - example, 565-569
    - left shift operator (<<), 581-587, 676
    - member versus nonmember functions, 587-588
    - multiplication operator (\*), 574-578
    - operator functions, 565
    - restrictions, 573-574
    - subtraction operator (-), 574-578
    - vector class, 588-590
  - reference parameters, 415
  - templates, 422-424
    - overload resolution, 431-438
- override, 1183-1184**
- ownership, 973**

---

## P

- pairs.cpp, 848**
- palindromes, 1057**
- pam() function, 362-363**
- parameter lists, 30**
- parameterized types, 419**
- parameters, 314**
  - templates, 855-858
  - type, 834
- partial ordering rules, 434-436**
- partial specializations, 851-852**
- partial\_sort() function, 1304, 1307**
- partial\_sort\_copy() function, 1304, 1307-1308**
- partial\_sum() function, 1321**
- partition() function, 1295, 1302-1303**
- passing**
  - structure addresses, 351-353
  - structures, 344-351
- passing by reference, 386, 389-390**
- passing objects**
  - by reference, 770
  - by value, 770
- past-the-end iterators, 981-982**
- peek() member function, 1109-1114**
- peeker.cpp, 1111**
- period (.), 255**
- permutations, 1038**
  - defined, 1318
  - functions
    - next\_permutation(), 1306, 1319
    - prev\_permutation(), 1319
- pf() function, 364**
- pipe character (|), 1237-1238**
  - logical OR operator (||), 260-262
    - alternative representations, 270
    - example, 261-262
    - precedence, 269
- placement new operator, 478-482, 671-676**
- placeneu1.cpp, 671-673**
- placeneu2.cpp, 674-675**
- plus sign (+)**
  - addition operator (+), overloading, 569-572
  - concatenation operator (+), strings, 133-134
  - increment operator (++), 207-208
    - pointers, 210-211
    - postfixing, 209-210
    - prefixing, 209-210
- plus\_one.cpp, 207**
- pointed-to values, 172**
- pointer arithmetic, 167-172**



**pointer.cpp, 155****pointers, 153, 321-322, 837**

- assigning values to, 171
- auto\_ptr, 969, 973-975
- C++ philosophy, 155
- cautions, 159
- compared to pointed-to values, 172
- const keyword, 334-336
- declaring, 155-159, 171
- deferencing, 171-172
- delete operator, 163-164
- example, 154
- function pointers, 361-362
  - addresses of functions, obtaining, 362
  - declaring, 362-363
- increment/decrement operators, 210-211
- inheritance, 737-739
- initializing, 157-159
- integers, 160
- iterators, 1001
- member dereferencing operators, 1242-1246
- new operator, 160-162
- passing variables, 386-390
- pointer arithmetic, 167-172
- pointer notation, 173
- pointers to objects, 665-670
- pointers to pointers, 335
- stacks of pointers, 837-843
- strings, 173-178
- this, 539-546

**polar coordinates, 347**

- converting rectangular coordinates to, 348-351

**polymorphic public inheritance, 722-723**

- base-class functions, 777
- Brass class declaration, 723-726
- Brass class implementation, 727-731
- Brass class objects, 732-733
- BrassPlus class declaration, 723-726
- BrassPlus class implementation, 727-731
- BrassPlus class objects, 732-733
- constructors, 742
- dynamic binding, 737-740
- pointer compatibility, 737-739

- reference type compatibility, 737-739

- static binding, 737, 740

- virtual destructors, 737, 742-743, 776

- virtual functions, 739-742, 775-776

- behavior, 734-736

- friends, 743, 776

- memory and execution speed, 742

- redefinition, 743-745

- virtual function tables, 740

**pop() method, 837****popping values off heap, 1314****pop\_heap() function, 1305****portability of C++, 15-18**

- ANSI/ISO standard, 16

- limitations, 16

**postfixing, 209-210****potential scope, 483****pound sign (#), 234****pow() function, 53****precedence**

- comma operator, 217

- logical operators, 269-270

**precedence of operators, 1231**

- examples, 1234

- table of, 1232-1234

**precise.cpp, 1082****precision() function, 1082-1083****precision() method, 408****predefined functors, 1030-1032**

- equivalents for operators, 1032

**predicates**

- adaptable, 1035

- binary, 1027-1030

- adaptable, 1035

- unary, 1027, 1030

**prefixing, 209-210****preprocessors, 33-34****prev\_permutation() function, 1319****print() functions, 413****printf() function, 29, 44****priority\_queue class templates, 1017-1018****priority\_queue containers, 1017-1018****private inheritance, 797**

- base-class components, initializing, 798-799

- base-class friends, accessing, 801-804

- base-class methods, accessing, 800-801

- base-class objects, accessing, 801
- compared to containment, 806
- Student class example, 798, 804–805
- private keyword, 511–513, 798**
- private member functions, 513**
- probability() function, 318–320**
- problem domains, 1207**
- procedural languages, 11–12**
- procedural programming, 506–507**
- procedures. See functions**
- programming**
  - concurrent programming, 1202–1203
  - low-level programming, 1203–1204
  - programming, generic, 992**
    - associative containers, 1018–1026
      - multimap, 1023–1025
      - set, 1019–1022
    - container concepts, 1007–1012
      - container methods compared to functions, 1039–1041
      - properties, 1008–1009
      - sequence requirements, 1011–1012
    - container types, 1013, 1017–1018
      - deque, 1013
      - list, 1014–1017
      - priority\_queue, 1017–1018
      - queue, 1017
      - stack, 1018
      - vector, 1012–1013
  - iterators, 992, 997–1005
    - back insert iterators, 1005–1007
    - bidirectional iterators, 998
    - concepts, 1000
    - copy() function, 1001–1002
    - forward iterators, 998
    - front insert iterators, 1005–1007
    - hierarchy, 999–1000
    - importance of, 992–996
    - input iterators, 997–998
    - insert iterators, 1005–1007
    - istream iterator template, 1003
    - models, 1001
    - ostream iterator template, 1002
    - output iterators, 998
    - pointers, 1001
    - random access iterators, 999
    - reverse iterators, 1003–1005
    - types, 997
- programming exercises**
  - chapter 2, 62–63
  - chapter 3, 111–113
  - chapter 4, 192–193
  - chapter 5, 251–252
  - chapter 6, 301–303
  - chapter 7, 374–377
  - chapter 8, 444–446
  - chapter 9, 501–503
  - chapter 10, 559–562
  - chapter 11, 623–624
  - chapter 12, 702–705
  - chapter 13, 780–783
  - chapter 14, 871–876
  - chapter 15, 949
  - chapter 16, 1057–1058
  - chapter 17, 1148–1151
  - chapter 18, 1212–1213
- programs**
  - comments, 33
    - /\*...\*/ notation, 33
    - // notation, 32
  - creating main() function, 31
  - header filenames, 34
  - istream file, 33–34
  - main() function, 29–30
    - calling, 30
    - importance of, 32
    - int main() header, 30–31
  - myfirst.cpp example, 27–29
    - comments, 32–33
    - header filenames, 34
    - istream file, 33–34
    - main() function, 29–32
    - namespaces, 35–36
    - output, displaying with cout, 36–39
      - source code formatting, 39–40
    - namespaces, 35–36
    - output, displaying with cout, 36
      - concatenated output, 46–47
      - endl notation, 37–38
      - integer values, 44–45
      - \n newline character, 38–39
    - source code formatting, 39
      - tokens, 39–40
      - white space, 39
  - properties**
    - algorithms, 1036–1037
    - class member functions, 777–778

containers, 1008-1010  
 reference variables, 390-391

**protected classes, 745-746, 775**

**protected inheritance, 806-807**

**protected keyword, 745-746, 775, 806**

**protos.cpp, 310**

**prototypes (functions), 50-52, 309-311**  
 benefits, 312-313  
 C++ compared to ANSI C, 312  
 function prototypes, 1330  
 syntax, 311-312

**pseudorandom numbers, 605**

**ptrstr.cpp, 174-175**

**public derivation, 711**

**public inheritance, 806, 808, 829. See also MI (multiple inheritance)**  
 multiple inheritance, methods, 826

**public inheritance (polymorphic), 722-723**  
 base-class functions, 777  
 Brass class declaration, 723-726  
 Brass class implementation, 727, 730-731  
 Brass class objects, 732-733  
 BrassPlus class declaration, 723-726  
 BrassPlus class implementation, 727, 730-731  
 BrassPlus class objects, 732-733  
 constructors, 742  
 dynamic binding, 737-740  
 pointer compatibility, 737-739  
 reference type compatibility, 737-739  
 static binding, 737, 740  
 virtual destructors, 737, 742-743, 776  
 virtual functions, 739-742, 775-776  
   behavior, 734-736  
   friends, 743, 776  
   memory and execution speed, 742  
   redefinition, 743-745  
   virtual function tables, 740

**public interfaces, 509**

**public keyword, 511-513**

**public member functions, 513**

**pure virtual functions, 748**

**push\_back() function, 1041**

**push\_back() method, 982-984**

**push\_heap() function, 1305**

**pushing values onto heap, 1314**

**put() method, 1071-1075**

**putback() member function, 1109-1114**

---

## Q

**qualified, inline, 517**

**qualified names, 486**

**qualified names (functions), 514**

**qualifiers**  
 cv-qualifiers, 472-473  
   const, 473-474  
   volatile, 473  
 keywords, 473

**Queue class**  
 class declaration, 691, 694  
 design, 679  
 implementation, 680-682  
 methods, 682-690  
 public interface, 679-680

**queue class templates, 1017**

**queue containers, 1017**

**Queue() function, 683**

**queue simulation, 678**  
 bank.cpp simulation, 694-698  
 Customer class, 690-691, 694  
 Queue class  
   class declaration, 691, 694  
   design, 679  
   implementation, 680-682  
   methods, 682-690  
   public interface, 679-680

**queue.cpp, 692-694**

**queue.h, 691-692**

**queuecount() function, 685**

**queuetp.h, 893-895**

**quotation marks (), 36**

---

## R

**rand() function, 53, 605**

**random access, files, 1133-1141**  
 temporary files, 1141-1142

**random access iterators, 999**

**Random Walk sample program, 603**

**random.cpp, 1138-1139**

**random\_shuffle() function, 987-988, 1295, 1302**

**random\_shuffle() STL function, 991**

**randwalk.cpp, 603**

**range-based for loop**  
 C++11, 233-234  
 templates, 1161

- range\_error** exception, 919
- ranges**, logical AND operator (&&), 265-267
- ranges** (arrays), 332-334
- RatedPlayer** class, 711-712
  - header files, 716
  - method definitions, 716
  - RatedPlayer object, 717-718
- Rating()** method, 719
- rbegin()** method, 1251, 1275
- rdstate()** stream state method, 1098-1102
- read()** member function, 1109-1114, 1130-1133
- reading**
  - C-style strings
    - get(), 127-130
    - getline(), 126-127
  - from files, 1116-1118
  - string class strings, 136-140
  - text files, 292-298
  - text with loops, 234
    - cin.get() function, 235-237, 241-244
    - cin object, 234-235
    - end-of-file conditions, 237-241
    - sentinel characters, 234
- real numbers**, 50
- recommended reading**, 1323-1324
- rect\_to\_polar()** function, 348-349
- rectangular coordinates**, 346
  - converting to polar coordinates, 348-351
- recur.cpp**, 355, 358
- recurs()** function, 357-359
- recursion**, 357
  - multiple recursive calls, 359-361
  - single recursive call, 358-359
  - variadic template functions, 1199-1202
- recursive use of templates**, 846-847
- redefining virtual functions**, 743-745
- redirecting I/O**, 1067-1068
- redirection**, 238
- refcube()** function, 391-393
- reference**, passing by, 343, 386, 389-390, 770
- reference arguments**, 392-394, 408-409
- reference counting**, 973
- reference operator (&)**, 383-386
- reference parameters**, overloading, 415
- reference returns**, const, 400
- reference type**, 1273
- reference variables**, 383
  - arguments, 408-409
  - class objects, 401-405
  - creating, 383-386
  - function parameters, 386, 389-390
  - inheritance, 405-408
  - initialization, 385
  - properties, 390-391
  - structures, 394, 397-399
    - const keyword, 401
  - return references, 399-400
- references**
  - inheritance, 737-739
  - returning, 399, 770-771
- referencing declarations**, 463
- refinement**, 1001
- register keyword**, 472
- reinterpret\_cast** operators, 946
- relational operators**, 217-220
  - C-style strings, comparing, 220-223
  - equality operator (==), 218-220
  - functor equivalents, 1031-1032
  - string class strings, comparing, 223-224
  - table of, 217
- relationships**
  - has-a, 721, 788
  - is-a, 720-722, 772, 808
- remodel()** function, 971
- remote\_access()** function, 468
- remove\_copy()** function, 1295, 1300
- remove\_copy\_if()** function, 1295, 1300
- remove()** function, 1295, 1299
- remove\_if()** function, 1295, 1300
- remove\_if()** method, 1280
- remove()** method, 1280
- rend()** method, 1251-1252, 1275
- replace\_copy()** function, 1294, 1298
- replace\_copy\_if()** function, 1294, 1298
- replace()** function, 1294, 1298, 1302
- replace\_if()** function, 1294, 1298
- replace()** method, 1268-1269
- replacing strings**, 1268-1269
- report()** function, 859
- reserve()** method, 966, 1258, 1279
- reserved names**, 1222-1223

**reserved words, 1221**  
 alternative tokens, table of, 1222  
 keywords, table of, 1221  
 reserved names, 1222-1223

**ResetRanking() method, 718**

**resize() method, 1047, 1258, 1278**

**return addresses, 909**

**return statements, 30**

**return types, declarations, 1157**

**return types (functions), 30**

**return values, 31**

**return values (functions), 49, 57-59**

**returning**  
 C-style strings, 341-343  
 references, 399  
 structures, 344-345, 348-351

**returning objects, 662-664**  
 const objects, 662-665  
 non-const objects, 663

**reverse\_copy() function, 1295, 1301**

**reverse() function, 1295**

**reverse iterators, 1003-1005**

**reverse() method, 1252, 1280**

**reversible containers**  
 associative, multimap, 1023-1025  
 list, 1014, 1017  
 member functions, 1014-1016  
 vector, 1012-1013

**review questions**  
 chapter 2, 62  
 chapter 3, 110-111  
 chapter 4, 191-192  
 chapter 5, 250  
 chapter 6, 298-300  
 chapter 7, 372-373  
 chapter 8, 443-444  
 chapter 9, 498-501  
 chapter 10, 558-559  
 chapter 11, 623  
 chapter 12, 700-702  
 chapter 13, 779-780  
 chapter 14, 869-870  
 chapter 15, 947-949  
 chapter 16, 1056-1057  
 chapter 17, 1146-1147  
 chapter 18, 1209-1212

**rewrite rule, 517**

**rfind() method, 961, 1261**

**right manipulator, 1091**

**right shift and assign operator ([]=), 1237**

**right shift operator ([]), 1236**

**Ritchie, Dennis, 11**

**rotate\_copy() function, 1295, 1302**

**rotate() function, 1295, 1301**

**RTTI (runtime type information), 933-934**  
 incorrect usage, 941-943  
 operators  
 dynamic cast, 934-939  
 typeid, 934, 939, 941, 944  
 type info class, 939-941, 944  
 type info structure, 934

**rtti1.cpp, 936-938**

**rtti2.cpp, 939-941**

**ruler.cpp, 360**

**runtime, 155**

**runtime\_error exception, 919**

**runtime type information, 933**

**rvalue reference, 1162-1164**  
 constructors, C++11, 1256

**rvalues, 400**

---

## S

**Sales class**  
 sales.cpp, 924  
 sales.h, 922  
 use\_sales.cpp, 925-927

**sales.cpp, 924**

**sales.h, 922**

**sayings1.cpp, 656**

**sayings2.cpp, 665**

**SCHAR\_MAX constant, 72**

**SCHAR\_MIN constant, 72**

**scientific manipulator, 1091**

**scope, 454, 483**  
 class, 454, 514  
 class scope, 549-551  
 function prototype, 454  
 global, 454  
 local, 454-457  
 namespace, 454  
 nested classes, 891-892  
 potential, 483

**scope-resolution operator (::), 467, 514, 1332**

**scoped enumerations, 1158**  
 C++11, 551-552

**search() function, 1288, 1292-1293**

- search\_n() function, 1288, 1293**
- searching strings, 960-961, 1260**
  - find\_first\_not\_of() method, 1262-1263
  - find\_first\_of() method, 1262
  - find\_last\_not\_of() method, 1263
  - find\_last\_of() method, 1262
  - find() method, 1260-1261
  - rfind() method, 1261
- secretf.cpp, 385**
- seekg() method, 1134-1136**
- seekp() method, 1134-1136**
- selecting smart pointers, 977-978**
- sell() function, 516**
- semantics, move semantics, 1164-1165, 1168-1171**
  - observations, 1171-1172
- semicolon (), 29-30**
- sending messages, OOP, 518**
- sentinel characters, 234**
- separate compilation, 447-449, 453**
- sequence points, 208-209**
- sequence requirements, container concepts, 1011-1012**
- sequences**
  - mutating sequence operations
    - copy\_backward() function, 1294-1297
    - copy() function, 1293-1296
    - fill() function, 1294, 1299
    - fill\_n() function, 1294, 1299
    - generate() function, 1294, 1299
    - generate\_n() function, 1294, 1299
    - iter\_swap() function, 1294
    - partition() function, 1295, 1302-1303
    - random\_shuffle() function, 1295, 1302
    - remove\_copy() function, 1295, 1300
    - remove\_copy\_if() function, 1295, 1300
    - remove\_if() function, 1295, 1300
    - remove() function, 1295, 1299
    - replace\_copy() function, 1294, 1298
    - replace\_copy\_if() function, 1294, 1298
    - replace() function, 1294, 1298, 1302
    - replace\_if() function, 1294, 1298
    - reverse\_copy() function, 1295, 1301
    - reverse() function, 1295
    - rotate\_copy() function, 1295, 1302
    - rotate() function, 1295, 1301
    - stable\_partition() function, 1295, 1303
    - swap() function, 1294, 1297
    - swap\_ranges() function, 1294, 1297
    - transform() function, 1294, 1297
    - unique\_copy() function, 1295, 1301
    - unique() function, 1295, 1300
  - nonmodifying sequence operations
    - adjacent\_find() function, 1287, 1290
    - count() function, 1287, 1291
    - count\_if() function, 1287, 1291
    - equal() function, 1288, 1291-1292
    - find\_end() function, 1287-1290
    - find\_first\_of() function, 1287, 1290
    - find() function, 1287-1289
    - find\_if() function, 1287-1289
    - for\_each() function, 1287-1289
    - mismatch() function, 1288, 1291
    - search() function, 1288, 1292-1293
    - search\_n() function, 1288, 1293
- set associative containers, 1019-1022**
- set flag. See setf() function**
- set\_difference() function, 1021, 1305, 1313**
- set\_intersection() function, 1021, 1305, 1312**
- set() method, 596, 820**
- set operations**
  - includes() function, 1305, 1311
  - set\_difference() function, 1305, 1313
  - set\_intersection() function, 1305, 1312
  - set\_symmetric\_difference() function, 1305, 1313
  - set\_union() function, 1305, 1312
- set\_symmetric\_difference() function, 1305, 1313**
- set\_terminate() function, 928**
- set\_tot() function, 517-518**
- set\_unexpected() function, 929**
- set\_union() function, 1020-1021, 1305, 1312**
- setf.cpp, 1085**
- setf2.cpp, 1088**

- setf() function, 1083-1087, 1090**
  - arguments, 1087-1089
  - manipulators, 1090-1091
- setf() method, 408**
- setfill() function, 1091**
- setops.cpp, 1021-1022**
- setprecision() function, 1091**
- sets**
  - methods, 1281-1284
  - set operations
    - includes() function, 1305, 1311
    - set\_difference() function, 1305, 1313
    - set\_intersection() function, 1305, 1312
    - set\_symmetric\_difference() function, 1305, 1313
    - set\_union() function, 1305, 1312
- setstate() stream state method, 1098-1102**
- setw() function, 1091**
- shallow copying, 640**
- shared friends, 888-889**
- shift operators, 1235-1237**
  - overloading, 581-587
- short data type, 68-70**
- show(), array objects, 357**
- show\_array() function, 327-328**
- Show() function, 818-820**
- show() method, 514, 537**
- show\_polar() function, 347, 351**
- show\_time() function, 344-345**
- showbase manipulator, 1090**
- showperks() function, 744**
- showpoint manipulator, 1091**
- showpos manipulator, 1091**
- showpt.cpp, 1084**
- SHRT\_MAX constant, 72**
- SHRT\_MIN constant, 72**
- side effects of expressions, 201, 208-209**
- signatures (functions), 413**
- signed char data type, 88-89**
- singly linked lists, 680**
- size, string size**
  - finding, 960
  - automatic sizing feature, 966-967
- size() function, 136, 960**
- size() method, 509, 787, 981, 984, 1251-1252, 1275**
- size\_type constant, 1251**
- size\_type type, 1250, 1273**
- sizeof operator, 71-73**
- smart pointers, 1158**
  - selecting, 977-978
- sort() function, 987-988, 1041, 1304, 1307**
- sort\_heap() function, 1305**
- sort() method, 1016-1017, 1280**
- sort() STL function, 991**
- sorting**
  - heaps, 1315
  - nth\_element() function, 1304, 1308, 1315
  - partial\_sort\_copy() function, 1304, 1307-1308
  - partial\_sort() function, 1304, 1307
  - sort() function, 1304, 1307
  - stable\_sort() function, 1304, 1307
  - vectors, 987
- source code, 19**
  - definition of, 18
  - file extensions, 20
- special meanings, identifiers, 1223**
- special member functions, 1178-1179**
- specializations**
  - explicit, 425, 850-851
    - example, 426-428
    - third-generation specialization, 425-426
  - explicit instantiations, 850
  - implicit instantiations, 850
  - partial specializations, 851-852
- specifiers**
  - storage class, 472-473
- splice() method, 1016, 1280**
- sqrt.cpp, 51**
- sqrt() function, 50-52**
- square() function, 381-382**
- stable\_partition() function, 1295, 1303**
- stable\_sort() function, 1304, 1307**
- stack, unwinding, 909-910, 913-914**
- stack class, 831-836**
  - pointers, 837-843
  - templates, 1018
- stack containers, 1018**
- stack.cpp, 554-555**
- stack.h, 553-554**
- stacker.cpp, 556-557**
- stacks, 553-557**
  - automatic variables, 458-459
- stacktem.cpp, 835-836**

**stacktp.h, 834****standard C++, converting to, 1327**

- auto\_ptr template, 1333
- C++ features, 1331
- const instead of #define, 1327-1329
- function prototypes, 1330
- header files, 1331
- inline instead of #define, 1329-1330
- namespaces, 1331-1333
- STL (Standard Template Library), 1334
- string class, 1333
- type casts, 1330-1331

**Standard Input/Output, ANSI C, 1062****Standard Template Library. See STL****state members, 597-599****statements, 41**

- assignment statements, 43-44
- blocks, 212-214
- break, 280-282
- cin, 46
  - loops, 234-235
- compared to expressions, 201
- continue, 280-282
- cout, 36
  - concatenated output, 46-47
  - cout.put() function, 83-84
  - endl manipulator, 37-38
  - integer values, displaying, 44-45
  - \n newline character, 38-39
- declaration statements, 41-43
- defined, 29-30
- enum, 278-280
- examples, 41, 45
- for, declaration-statement expressions, 203
- if, 254
  - bug prevention, 260
  - example, 255
  - syntax, 254
- if else, 255
  - example, 256-257
  - formatting, 257-258
  - if else if else construction, 258-260
  - syntax, 255
- return statements, 30
- switch, 274-278
  - enumerators as labels, 278-280
  - example, 275-278
  - syntax, 275

- terminators, 30

- void, 53

**static assert, 1204****static binding, 164, 172-173, 737, 740****static\_cast operator, 945-946****static class members, 628-637****static keyword, 183, 472**

- functions, 475

**static memory storage, 182-183****static storage duration, 453****static type checking, 313****static variables, 453, 459-462, 466**

- external, 466
  - external linkage, 463, 466-467
  - internal linkage, 467-470
  - no linkage, 470-472

**static.cpp, 470-471****stcktp1.h, 839-840****std namespace, 59****stdexcept exception classes, 918-920****Stepanov, Alexander, 978, 1205****stkoptr1.cpp, 841-842****STL (Standard Template Library), 978, 1041, 1044, 1271, 1334**

- algorithms, 1035
  - groups, 1035-1036
  - properties, 1036-1037
- associative containers, 1018, 1026
  - multimap, 1023-1025
  - set, 1019-1022
- binary search operations
  - binary\_search() function, 1304, 1310
  - equal\_range() function, 1304, 1309
  - lower\_bound() function, 1304, 1309
  - upper\_bound() function, 1304, 1309
- C++11, 1271
  - containers, 1271-1273
- container concepts, 1007
  - container methods compared to functions, 1039-1041
  - properties, 1008-1010
  - sequence requirements, 1011-1012
- container methods, 1275-1277
- container types
  - deque, 1013
  - list, 1014-1017
  - priority\_queue, 1017-1018
  - queue, 1017



- stack, 1018
  - vector, 1012-1013
- containers, 1161
- deque methods, 1278-1280
- functions, 1286
  - adjacent\_find(), 1287, 1290
  - copy(), 1293-1296
  - copy\_backward(), 1294-1297
  - count(), 1287, 1291
  - count\_if(), 1287, 1291
  - equal(), 1288, 1291-1292
  - fill(), 1294, 1299
  - fill\_n(), 1294, 1299
  - find(), 1287-1289
  - find\_end(), 1287-1290
  - find\_first\_of(), 1287, 1290
  - find\_if(), 1287-1289
  - for\_each(), 1287-1289
  - generate(), 1294, 1299
  - generate\_n(), 1294, 1299
  - iter\_swap(), 1294
  - mismatch(), 1288, 1291
  - partition(), 1295, 1302-1303
  - random\_shuffle(), 1295, 1302
  - remove(), 1295, 1299
  - remove\_copy(), 1295, 1300
  - remove\_copy\_if(), 1295, 1300
  - remove\_if(), 1295, 1300
  - replace(), 1294, 1298, 1302
  - replace\_copy(), 1294, 1298
  - replace\_copy\_if(), 1294, 1298
  - replace\_if(), 1294, 1298
  - reverse(), 1295
  - reverse\_copy(), 1295, 1301
  - rotate(), 1295, 1301
  - rotate\_copy(), 1295, 1302
  - search(), 1288, 1292-1293
  - search\_n(), 1288, 1293
  - stable\_partition(), 1295, 1303
  - swap(), 1294, 1297
  - swap\_ranges(), 1294, 1297
  - transform(), 1294, 1297
  - unique(), 1295, 1300
  - unique\_copy(), 1295, 1301
- functors, 1026-1027
  - adaptable, 1032
  - concepts, 1027-1028, 1030
  - predefined, 1030-1032
- generic programming, 992

- heap operations
  - make\_heap() function, 1305, 1314
  - pop\_heap() function, 1305, 1314
  - push\_heap() function, 1305, 1314
  - sort\_heap() function, 1305, 1315
- iterators, 997
  - concepts, 1001
  - pointers, 1001
- list methods, 1278-1280
- map methods, 1281-1284
- merge operations
  - inplace\_merge() function, 1305, 1311
  - merge() function, 1305, 1310-1311
- methods, 1161
- minimum/maximum value operations
  - lexicographical\_compare() function, 1306, 1318
  - max\_element() function, 1305, 1317
  - max() function, 1305, 1316
  - min\_element() function, 1305, 1317
  - min() function, 1305, 1316
- numeric operations, 1319-1320
  - accumulate() function, 1320
  - adjacent\_difference() function, 1321-1322
  - inner\_product() function, 1320-1321
  - partial\_sum() function, 1321
- permutation operations
  - next\_permutation() function, 1306, 1319
  - prev\_permutation() function, 1319
- set methods, 1281-1284
- set operations
  - includes() function, 1305, 1311
  - set\_difference() function, 1305, 1313
  - set\_intersection() function, 1305, 1312
  - set\_symmetric\_difference() function, 1305, 1313
  - set\_union() function, 1305, 1312
- sorting operations
  - nth\_element() function, 1304, 1308, 1315
  - partial\_sort() function, 1304, 1307
  - partial\_sort\_copy() function, 1304, 1307-1308
  - sort() function, 1304, 1307
  - stable\_sort() function, 1304, 1307

- string class, 1038-1039
- types, 1273-1274
- usealgo.cpp sample program, 1042-1044
- using, 1041
- vector methods, 1278-1280
- Stock class, 511**
- stock00.h, 510**
- stock1.cpp, 531**
- stock1.h, 530**
- stocks.cpp, class member function, 515**
- stone.cpp, 610**
- stone1.cpp, 615**
- stonetob() function, 58**
- stonewt.cpp, 608**
- stonewt.h, 607**
- stonewt1.cpp, 614-615**
- stonewt1.h, 613**
- storage class qualifiers, 473**
- storage class specifiers, 472-473**
- storage duration, 453-454**
  - automatic variables, 455-457
    - example, 455-457
    - initializing, 458
    - stacks, 458-459
  - scope and linkage, 454
  - static variables, 459-462
    - external linkage, 463, 466-467
    - internal linkage, 467-470
    - no linkage, 470-472
- str1.cpp, 953**
- str2.cpp, 966-967, 971**
- strcat() function, 136**
- strcmp() function, 221-222, 648**
- strcpy() function, 177-178, 633**
- strctfun.cpp, 348**
- strctptr.cpp, 352-353**
- stream objects, 1067**
- stream states, 1097-1098**
  - effects, 1100-1102
  - exceptions, 1099-1100
  - file I/O, 1118-1119
  - get() and getline() input effects, 1108
  - setting, 1098
- streambuf class, 1065**
- streams, 1063-1064, 1067**
  - istream class, 47
  - ostream class, 47
- strfile.cpp, 958-959**
- strgfun.cpp, 340**
- strgstl.cpp, 1038-1039**
- strict weak ordering, 988**
- strin.cpp, 1144**
- string class, 131-133, 353-354, 647, 952, 960, 965-966, 1249-1250, 1333**
  - append methods, 1265-1266
  - assignment, 133-134
  - assignment methods, 1260, 1266
  - assignment operator, overloading, 652-658
    - sayings1.cpp, 656
    - string1.cpp, 653-656
    - string1.h, 652-653
  - automatic sizing, 966-967
  - bracket notation, 649-651
  - comparing, 960
  - comparison members, 648-649
  - comparison methods, 1263-1265
  - complex operations, 135-136
  - concatenation, 133-134
  - concatenation methods, 1266
  - constants, 1251
  - constructors, 952-956, 1253
    - copy constructors, 1255-1256
    - default constructors, 1254
    - that use arrays, 1254
    - that use n copies of characters, 1257
    - that use parts of arrays, 1254-1255
    - that use ranges, 1257
  - copy methods, 1269
  - data methods, 1251-1253
  - default constructor, 647-648
  - erase methods, 1267-1268
  - finding size of, 960
  - Hangman sample program, 962-965
  - input, 957-960
  - input/output, 1269-1270
  - insertion methods, 1267
  - memory-related methods, 1258
  - reading line by line, 136-140
  - replacement methods, 1268-1269
  - search methods, 1260
    - find(), 1260-1261
    - find\_first\_not\_of(), 1262-1263
    - find\_first\_of(), 1262
    - find\_last\_not\_of(), 1263
    - find\_last\_of(), 1262
    - rfind(), 1261

- searching, 960-961
- static class member functions, 651-652
- STL interface, 1038-1039
- string access methods, 1259
- string comparisons, 223-224
- structures, 144-145
- template definition, 1249
- types, 1250-1251
- string() constructors, 952-956, 1253**
  - copy constructors, 1255-1256
  - default constructors, 1254
  - that use arrays, 1254
  - that use n copies of character, 1257
  - that use parts of arrays, 1254-1255
  - that use ranges, 1257
- string1.cpp, 653-656**
- string1.h, 653**
- StringBad class, 628**
  - constructors, 632-633
  - destructor, 633
  - strngbad.cpp, 630-631
  - strngbad.h, 628-629
  - vegnews.cpp sample program, 633-637
- strings**
  - accessing, 1259
  - accessing with for loops, 206-207
  - appending, 1265-1266
  - assigning, 1266
  - C-style, 120-122
    - combining with numeric input, 130-131
    - concatenating, 122
    - empty lines, 130
    - failbits, 130
    - in arrays, 123-124
    - null characters, 121
    - passing as arguments, 339-341
    - pointers, 173-178
    - returning from functions, 341-343
    - string input, entering, 124-126
    - string input, reading with get(), 127-130
    - string input, reading with getline(), 126-127
  - comparing, 1263-1265
    - C-style strings, 220-223
    - string class strings, 223-224
  - concatenating, 128, 1266
  - copying, 1269
  - erasing, 1267-1268
  - initializing, 121
  - input, 1106-1108
  - input/output, 1269-1270
  - inserting, 1267
  - palindromes, 1057
  - replacing, 1268-1269
  - searching, 1260
    - find\_first\_not\_of() method, 1262-1263
    - find\_first\_of() method, 1262
    - find\_last\_not\_of() method, 1263
    - find\_last\_of() method, 1262
    - find() method, 1260-1261
    - rfind() method, 1261
  - string access methods, 1259
  - string class, 131-133, 647, 960, 966, 1249-1250, 1333
    - append methods, 1265-1266
    - appending, 133-134
    - assignment, 133-134
    - assignment methods, 1260, 1266
    - assignment operator, overloading, 652-658
    - automatic sizing, 966-967
    - bracket notation, 649-651
    - comparing, 960
    - comparison members, 648-649
    - comparison methods, 1263-1265
    - complex operations, 135-136
    - concatenation, 133-134
    - concatenation methods, 1266
    - constants, 1251
    - constructors, 952-956, 1253-1257
    - copy methods, 1269
    - data methods, 1251-1253
    - default constructor, 647-648
    - erase methods, 1267-1268
    - finding size of, 960
    - Hangman sample program, 962-965
    - input, 957-960
    - input/output, 1269-1270
    - insertion methods, 1267
    - memory-related methods, 1258
    - reading line by line, 136-140
    - replacement methods, 1268-1269
    - search methods, 1260-1263

- searching, 960-961
- static class member functions, 651-652
- STL interface, 1038-1039
- string access methods, 1259
- structures, 144-145
- template definition, 1249
- types, 1250-1251
- string class objects, 353-354
- StringBad class, 628
  - constructors, 632-633
  - destructor, 633
  - strngbad.cpp, 630-631
  - strngbad.h, 628-629
  - vegnews.cpp sample program, 633-637
  - swapping, 1269
- strings.cpp, 123**
- strlen() function, 123-124, 136, 177, 306, 632**
- strngbad.cpp, 630-631**
- strngbad.h, 629**
- Stroustrup, Bjarne, 14-15**
- strout.cpp, 1143**
- strquote.cpp, 402-403**
- strtref.cpp, 395**
- strtype1.cpp, 132**
- strtype2.cpp, 134**
- strtype4.cpp, 137**
- struct keyword, 140**
- structur.cpp, 142**
- structure initialization, C++11, 144**
- structure members, 141**
- structured programming, 12**
- structures, 140-142, 343, 346**
  - addresses, passing, 351-353
  - arrays, 147-148
  - assignment, 145-146
  - bit fields, 148
  - compared to classes, 514
  - dynamic structures, new operator, 178-180
  - example, 142-144
  - nested structures, 682
  - passing/returning structures, 344-345, 348-351
  - polar coordinates, 347
  - rectangular coordinates, 346
  - reference variables, 394, 397-399
    - const keyword, 401
    - return references, 399-400
  - string class members, 144-145
- Student class**
  - contained object interfaces, 792-795
  - contained objects, initializing, 791
  - design, 787-788
  - methods, 793-795
  - private inheritance, 798, 804-805
    - base-class components, initializing, 798-799
    - base-class friends, accessing, 801-804
    - base-class methods, accessing, 800-801
    - base-class objects, accessing, 801
  - sample program, 795-797
  - studentc.h, 789-790
- studentc.cpp, 793**
- studentc.h, 789-790**
- studenti.cpp, 802-803**
- studenti.h, 799**
- subdivide() function, 360-361**
- sube() function, 311**
- subobjects, 797**
- subroutines. See functions**
- subscripts, 117**
- substrings, finding, 1260**
  - find\_first\_not\_of() method, 1262-1263
  - find\_first\_of() method, 1262
  - find\_last\_not\_of() method, 1263
  - find\_last\_of() method, 1262
  - find() method, 1260-1261
  - rfind() method, 1261
- subtraction operator (-), overloading, 574-578**
- sum\_arr() function, 320-322, 325**
- sum() function, 344-345, 565-567**
- sum() method, 787, 807, 1046**
- sumafire.cpp, 294-295**
- Swap() function, 420-421, 1294, 1297**
- swap() method, 981, 1269, 1276**
- swap\_ranges() function, 1294, 1297**
- swapp() function, 389**
- swapping strings, 1269**
- swapr() function, 389**
- swaps.cpp, 387-388**
- swapv() function, 389**

**switch.cpp, 276-277**  
**switch statement, 274-278**  
 compared to if else statement, 279-280  
 enumerators as labels, 278-279  
 example, 275-278  
 syntax, 275  
**symbolic constants, 72**  
**symbolic names, 90-92**

---

**T**

---

**TableTennisPlayer class, 708-710**  
 tabtenn0.cpp, 709  
 tabtenn0.h, 708  
 usett0.cpp, 710  
**tabtenn0.cpp, 709**  
**tabtenn0.h, 708**  
**tabtenn1.cpp, 717**  
**tabtenn1.h, 716**  
**tags, 140**  
**Technical Report 1 (TR1), 1206**  
**template aliases, C++11, 866**  
**template classes, 830-837**  
 arrays, non-type arguments, 843-845  
 auto\_ptr, 969, 973-975  
 complex, 1045  
 deque, 1013  
 explicit instantiations, 850  
 explicit specializations, 850-851  
 implicit instantiations, 850  
 list, 1014-1017  
     member functions, 1014-1016  
 members, 854-855  
 partial specializations, 851-852  
 pointers, stacks of pointers, 837-843  
 priority\_queue, 1017-1018  
 queue, 1017  
 stack, 1018  
 valarray, 1045-1046, 1049-1051  
 vector, 979-991, 1012-1013, 1045-1046,  
 1049-1051  
     adding elements to, 982-983  
     past-the-end iterators, 981-982  
     removing ranges of, 982  
     shuffling elements in, 987  
     sorting, 987  
     vect1.cpp example, 980-981  
     vect2.cpp sample program, 984-986  
     vect3.cpp sample program, 988-991

    versatility, 845-846  
     default type parameters, 849  
     multiple type parameters, 847  
     recursive use, 846-847  
**template functions, 438**  
 alternative function syntax, 441  
 decltype, 439  
 type, 439  
**template keyword, 831**  
**template parameter packs, 1197-1198**  
 unpacking, 1198-1199  
**templates. See also STL (Standard Template Library)**  
 angle brackets, 1162  
 auto\_ptr, 1333  
 export, 1162  
 friend classes, 858  
     bound template friend functions,  
         861-864  
     non-template friend functions,  
         858-861  
     unbound template friend functions,  
         864-865  
 function templates, 419, 422  
     explicit instantiation, 428-430  
     explicit specializations, 425-428  
     implicit instantiation, 428-430  
     overload resolution, 431-438  
     overloading, 422-424  
 inefficiencies, function wrapper,  
     1191-1194  
 initializer\_list, C++11, 1051-1053  
 istream iterator, 1003  
 nested classes, 892-896  
 ostream iterator, 1002-1003  
 parameters, 855-858  
 range-based for loop, 1161  
 STL (Standard Template Library), 1334  
 string template class, 1249-1250  
     append methods, 1265-1266  
     assignment methods, 1260, 1266  
     comparison methods, 1263-1265  
     concatenation methods, 1266  
     constants, 1251  
     constructors, 1253-1257  
     copy methods, 1269  
     data methods, 1251-1253  
     erase methods, 1267-1268  
     input/output, 1269-1270

- insertion methods, 1267
- memory-related methods, 1258
- replacement methods, 1268-1269
- search methods, 1260-1263
- string access methods, 1259
- template definition, 1249
- types, 1250-1251
- valarray, 1162
- variadic templates, 866, 1197
  - recursion, 1199-1202
  - template and function parameter packs, 1197-1198
  - unpacking the packs, 1198-1199
- tempmemb.cpp, 852**
- temporary files, random access, 1141-1142**
- temporary variables, 392-394**
- tempover.cpp, 434-437**
- tempparm.cpp, 856-857**
- terminate() function, 928-930**
- terminators, 30**
- testing bit values, 1241-1242**
- tests, loop tests, 196-197**
- text, reading with loops, 234**
  - cin.get() function, 235-237, 241-244
  - cin object, 234-235
  - end-of-file conditions, 237-241
  - sentinel characters, 234
- text files, 287-288, 1129**
  - reading, 292-298
  - writing to, 288-292
- textin1.cpp, 234**
- textin2.cpp, 236**
- textin3.cpp, 239**
- textin4.cpp, 242**
- third-generation specialization, 425-426**
- this pointer, 539**
- throw keyword, 900**
- throwing exceptions, 900, 915-916**
- tilde, 529, 1237**
- time, 1009**
- time-delay loops, 229-230**
- tmp2tmp.cpp, 862-864**
- toasting bits, 1241**
- tokens, 39**
  - alternative tokens, table of, 1222
- tolower() function, 273, 1041**
- top-down design, 12**
- top-down programming, 331**
- topfive.cpp, 353**
- topval() method, 543**
- total ordering, 988**
- totals, calculating cumulative totals, 1320**
- toupper() function, 273**
- trailing zeros/decimal points, printing, 1083-1087, 1090**
- traits type, 1250**
- transform() function, 1030-1031, 1041, 1294, 1297**
- translation units, compiling separately, 447-449, 453**
- translator (cfront), 21**
- travel.cpp, 344-345**
- trivial conversations for exact matches, 432**
- troubleshooting compilers, 24**
- truncate.cpp, 1113**
- try blocks, 901-903**
- try keyword, 901**
- turning off bits, 1241**
- turning on bits, 1241**
- Tv class, 878-879, 883**
  - tv.cpp, 880-882
  - tv.h, 879-880
  - tvfm.h, 885-886
  - use\_tv.cpp, 882
- tv.cpp, 880-882**
- tv.h, 879-880**
- tvfm.h, 885-886**
- two-dimensional arrays, 244-249**
  - declaring, 244-246
  - initializing, 246-249
- twoarg.cpp, 316**
- twod.cpp, 846-847**
- twofile2.cpp, 469**
- twoswap.cpp, 427-428**
- twotemps.cpp, 422**
- type cast operators, 943-944**
- type casts, 606-612, 1330-1331**
- type conversion, 606-612**
  - applying automatically, 616-618
  - conversion functions, 612-616
  - friends, 618-621
  - implicit conversion, 609
- type info class, 939-941, 944**
- type info structure, 934**
- type parameters, 834**
- type of template functions, 439**
- typecast.cpp, 108**

**typedef, 371**  
**typedef keyword, 230**  
**typeid operators, 934, 939-944**  
**typename keyword, 831**  
**types**  
   char\_type, 1250  
   const\_iterator, 1273  
   const\_reference, 1273  
   difference\_type, 1250, 1273  
   iterators, 997, 1273  
   key\_compare, 1281, 1284  
   key\_type, 1281, 1284  
   mapped\_type, 1281, 1284  
   reference, 1273  
   size\_type, 1250, 1273  
   traits, 1250  
   type casts, 1330-1331  
   value\_compare, 1281, 1284  
   value\_type, 1273

---

## U

**UCHAR\_MAX constant, 72**  
**UINT\_MAX constant, 72**  
**ULONG\_MAX constant, 72**  
**UML (Unified Modeling Language), 08**  
**unary functions, 1027, 1030**  
**unary minus operator, 601**  
**unary operators, 601, 1234**  
**unbound template friend functions, 864-865**  
**uncaught exceptions, 928-931**  
**underscore (\_), 1222**  
**unexpected exceptions, 928-931**  
**unexpected() function, 929**  
**unformatted input functions, 1102**  
**Unicode, 88**  
**Unified Modeling Language (UML), 1208**  
*Unified Modeling Language User Guide, 1323*  
**uniform initialization, 1154**  
   initializer\_list, 1155  
   narrowing, 1154  
**unions, 149**  
   anonymous unions, 150  
   declaring, 149  
**unique\_copy() function, 1295, 1301**  
**unique() function, 1041, 1295, 1300**  
**unique() method, 1016-1017, 1280**  
**unique\_ptr versus auto\_ptr, 975-977**  
**universal character names, 87-88**  
**UNIX, CC compiler, 21-22**  
**unnamed namespaces, 491-492**  
**unordered associative containers, C++11, 1283**  
**unqualified names (functions), 486, 514**  
**unsetf() function, 1090**  
**unsigned char data type, 88-89**  
**unsigned integers, 74-76**  
**unsigned long long type, 1153**  
**unwinding the stack, 909-910, 913-914**  
**upcasting, 738, 944**  
   implicit upcasting, 807  
**update() function, 466, 514**  
**updates, loop updates, 196-198, 205-206**  
**uppercase manipulator, 1091**  
**upper\_bound() functions, 1024, 1304, 1309**  
**upper\_bound() method, 1021, 1283**  
**use-case analysis, 1207**  
**use() function, 394, 397-399**  
**usealgo.cpp, 1043-1044**  
**usebrass1.cpp, 732-733**  
**usebrass2.cpp, 734**  
**usedma.cpp, 765**  
**usenmsp.cpp, 494-495**  
**user-defined functions**  
   example, 53-54  
   function form, 54-55  
   function headers, 55-56  
   return values, 57-59  
   using directive, 59-60  
**usestok0.cpp, 519**  
**usestok1.cpp, 533**  
**usestok2.cpp, 547**  
**usetime1.cpp, 571-572**  
**usetime2.cpp, 577**  
**usett0.cpp, 710**  
**usett1.cpp, 717-718**  
**use\_new.cpp, 161**  
**use\_ptr() method, 1244**  
**use\_sales.cpp, 925-927**  
**use\_stuc.cpp, 795-797**  
**use\_stui.cpp, 804-805**  
**use\_tv.cpp, 882**  
**USHRT\_MAX constant, 72**  
**using-declaration, 491**  
**using directive, 35-36, 59-60**  
**using keyword, 486-490, 807-808, 1332**

## V

**valarray class, 786-787, 1045-1046, 1049-1051**

templates, 1162

**value, passing by, 313-314, 770**

**value\_comp() method, 1282**

**value\_compare type, 1281, 1284**

**value\_type type, 1273**

**values**

assigning to pointers, 171

indirect, 155

**valvect.cpp, 1048**

**variable arrays, 1329**

**variables, 66**

assigning values to, 43

automatic, 182, 453-457

example, 455-457

initializing, 458

stacks, 458-459

automatic variables, 314

declaring, 41-43

dynamic, 454

dynamic memory, 476-479, 482

enum, 150-152

enumerators, 150-151

value ranges, 153

values, setting, 152

floating-point numbers, 92

advantages/disadvantages, 96-97

constants, 96

decimal-point notation, 92

double data type, 94-96

E notation, 92-93

float data type, 94-96

long double data type, 94-96

global, compared to local

variables, 467

indeterminate values, 73

initializing, 52, 73

integers, 68

bool, 90

char, 80-89

choosing integer types, 76-77

climits header file, 71-73

constants, 78-80, 90-92

initializing, 73

int, 68-70

long, 68-70

short, 68-70

size of operator, 71-73

unsigned, 74-76

width of, 68

keywords, static, 183

local variables, 314-315

compared to global variables, 467

naming conventions, 66-68

pointers, 153

assigning values to, 171

C++ philosophy, 155

cautions, 159

compared to pointed-to values, 172

declaring, 155-159, 171

deferencing, 171-172

delete operator, 163-164

example, 154

initializing, 157-159

integers, 160

new operator, 160-162

pointer arithmetic, 167-172

pointer notation, 173

strings, 173-178

reference variables, 383

arguments, 408-409

class objects, 401-405

creating, 383-386

function parameters, 386, 389-390

inheritance, 405-408

properties, 390-391

structures, 394, 397-401

scope, 454

global, 454

local, 454-457

namespace, 454

static, 453, 459-462

external, 466

external linkage, 463, 466-467

internal linkage, 467-470

no linkage, 470-472

temporary variables, 392-394

type conversions, 102

in arguments, 106

in expressions, 105-106

on assignment, 103-104

type casts, 107-109



**variadic templates, 866, 1197**  
 recursion, 1199-1202  
 template and function parameter packs, 1197-1198  
 unpacking the packs, 1198-1199  
**variadic2.cpp, 1201**  
**variations on function pointers, 365-370**  
**vect.cpp, 593**  
**vect.h, 591**  
**vect1.cpp, 980**  
**vect2.cpp, 984-985**  
**vect3.cpp, 988**  
**vector class, 588-590, 600, 979-988, 991, 1045-1046, 1049-1051**  
 adding elements to, 982-983  
 adding vectors, 590  
 declaring, 591-592  
 displacement vectors, 589  
 implementation comments, 602  
 member functions, 592, 597  
 multiple representations, 599  
 overloaded arithmetic operators, 599-600  
 overloading overloaded operators, 601  
 past-the-end iterators, 981-982  
 Random Walk sample program, 602, 605-606  
 removing ranges of, 982  
 shuffling elements in, 987  
 sorting, 987  
 state members, 597-599  
 vect1.cpp example, 980-981  
 vect2.cpp sample program, 984-986  
 vect3.cpp sample program, 988-991  
**vector class templates, 1012-1013**  
**vector containers, 1012-1013**  
**vector objects versus arrays, 188-189**  
**vector template class, 120, 186-187**  
**vectors, methods, 1278-1280**  
**vegnews.cpp, 634**  
**versatility of templates, 845-846**  
**version1() function, 403**  
**version2() function, 404**  
**version3() function, 405**  
**ViewAcct() function, 725-726, 730**  
**virtual base classes, 815-817**  
 combining with nonvirtual base classes, 828  
 constructors, 817-818

dominance, 828-829  
 methods, 818-828  
**virtual destructors, 737, 742-743, 776**  
**virtual function tables (vtbl), 740**  
**virtual functions, 739-742, 775-776**  
 behavior, 734-736  
 friends, 743, 776  
 memory and execution speed, 742  
 pure virtual functions, 748  
 redefinition, 743-745  
 virtual function tables, 740  
**virtual keyword, 742**  
**virtual methods, 1183-1184**  
**void functions, 307**  
**void statement, 53**  
**volatile keyword, 473**  
**vslice.cpp, 1049-1050**  
**vtbl (virtual function table), 740**

---

## W

**waiting.cpp, 229**  
**wchar\_t data type, 89, 1064**  
**width of integers, 68**  
**Web resources, 1325**  
**what() function, 917**  
**while loops, 224-227**  
 compared to for loops, 227-228  
 example, 225-226  
 syntax, 224  
 time-delay loops, 229-230  
**while.cpp, 225**  
**white space, 39**  
**width() method, 1080-1081**  
**width.cpp, 1080**  
**Windows, compilers, 23-24**  
**Withdraw() method, 731, 745**  
**Worker class, 810-814**  
**Worker0.cpp, 811-812**  
**Worker0.h, 810-811**  
**workermi.cpp, 823-825**  
**workermi.h, 821-822**  
**workmi.cpp, 826-827**  
**worktest.cpp, 813**  
**WorseThan() function, 988**  
**wow() function, 409**  
**wrapped.cpp, 1195**

**wrappers, 1191**

function wrapper

fixing problems, 1194-1196

options for, 1196-1197

template inefficiencies, 1191-1194

**write.cpp, 1073-1074**

**write() member function, 1130-1133**

**write() method, 1071-1075**

**writing**

to files, 1115-1118

to text files, 288-292

---

**X-Z**

---

**XOR operator, bitwise XOR (^), 1238**

**zeros, trailing, 1083-1087, 1090**