This page intentionally left blank
C++ Primer
Fifth Edition

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

▲Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City
To Beth,
who makes this,
and all things,
possible.

To Daniel and Anna,
who contain
virtually
all possibilities.
—SBL

To Mark and Mom,
for their
unconditional love and support.
—JL

To Andy,
who taught me
to program
and so much more.
—BEM
This page intentionally left blank
# Contents

**Preface** .................................................. xxiii

**Chapter 1 Getting Started** ............................... 1

1.1 Writing a Simple C++ Program .......................... 2
1.1.1 Compiling and Executing Our Program ............... 3
1.2 A First Look at Input/Output ............................ 5
1.3 A Word about Comments ................................. 9
1.4 Flow of Control ........................................ 11
1.4.1 The `while` Statement ............................... 11
1.4.2 The `for` Statement .................................. 13
1.4.3 Reading an Unknown Number of Inputs ............... 14
1.4.4 The `if` Statement ................................... 17
1.5 Introducing Classes .................................... 19
1.5.1 The `Sales_item` Class ............................. 20
1.5.2 A First Look at Member Functions ................... 23
1.6 The Bookstore Program .................................. 24

Chapter Summary ........................................... 26

Defined Terms ............................................... 26

**Part I The Basics** ......................................... 29

**Chapter 2 Variables and Basic Types** .................. 31

2.1 Primitive Built-in Types ............................... 32
2.1.1 Arithmetic Types .................................... 32
2.1.2 Type Conversions ................................... 35
2.1.3 Literals ............................................. 38
2.2 Variables ................................................. 41
2.2.1 Variable Definitions ................................ 41
2.2.2 Variable Declarations and Definitions ............... 44
2.2.3 Identifiers ........................................... 46
2.2.4 Scope of a Name ..................................... 48
2.3 Compound Types ......................................... 50
2.3.1 References ........................................... 50
2.3.2 Pointers .............................................. 52
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.3</td>
<td>Understanding Compound Type Declarations</td>
</tr>
<tr>
<td>2.4</td>
<td>const Qualifier</td>
</tr>
<tr>
<td>2.4.1</td>
<td>References to const</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Pointers and const</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Top-Level const</td>
</tr>
<tr>
<td>2.4.4</td>
<td>constexpr and Constant Expressions</td>
</tr>
<tr>
<td>2.5</td>
<td>Dealing with Types</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Type Aliases</td>
</tr>
<tr>
<td>2.5.2</td>
<td>The auto Type Specifier</td>
</tr>
<tr>
<td>2.5.3</td>
<td>The decltype Type Specifier</td>
</tr>
<tr>
<td>2.6</td>
<td>Defining Our Own Data Structures</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Defining the Sales_data Type</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Using the Sales_data Class</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Writing Our Own Header Files</td>
</tr>
<tr>
<td>Chapter Summary</td>
<td>78</td>
</tr>
<tr>
<td>Defined Terms</td>
<td>78</td>
</tr>
<tr>
<td><strong>Chapter 3 Strings, Vectors, and Arrays</strong></td>
<td>81</td>
</tr>
<tr>
<td>3.1</td>
<td>Namespace using Declarations</td>
</tr>
<tr>
<td>3.2</td>
<td>Library string Type</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Defining and Initializing strings</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Operations on strings</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Dealing with the Characters in a string</td>
</tr>
<tr>
<td>3.3</td>
<td>Library vector Type</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Defining and Initializing vectors</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Adding Elements to a vector</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Other vector Operations</td>
</tr>
<tr>
<td>3.4</td>
<td>Introducing Iterators</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Using Iterators</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Iterator Arithmetic</td>
</tr>
<tr>
<td>3.5</td>
<td>Arrays</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Defining and Initializing Built-in Arrays</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Accessing the Elements of an Array</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Pointers and Arrays</td>
</tr>
<tr>
<td>3.5.4</td>
<td>C-Style Character Strings</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Interfacing to Older Code</td>
</tr>
<tr>
<td>3.6</td>
<td>Multidimensional Arrays</td>
</tr>
<tr>
<td>Chapter Summary</td>
<td>131</td>
</tr>
<tr>
<td>Defined Terms</td>
<td>131</td>
</tr>
<tr>
<td><strong>Chapter 4 Expressions</strong></td>
<td>133</td>
</tr>
<tr>
<td>4.1</td>
<td>Fundamentals</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Basic Concepts</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Precedence and Associativity</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Order of Evaluation</td>
</tr>
<tr>
<td>4.2</td>
<td>Arithmetic Operators</td>
</tr>
<tr>
<td>4.3</td>
<td>Logical and Relational Operators</td>
</tr>
</tbody>
</table>
## Chapter 4: Operators

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Assignment Operators</td>
<td>144</td>
</tr>
<tr>
<td>4.5</td>
<td>Increment and Decrement Operators</td>
<td>147</td>
</tr>
<tr>
<td>4.6</td>
<td>The Member Access Operators</td>
<td>150</td>
</tr>
<tr>
<td>4.7</td>
<td>The Conditional Operator</td>
<td>151</td>
</tr>
<tr>
<td>4.8</td>
<td>The Bitwise Operators</td>
<td>152</td>
</tr>
<tr>
<td>4.9</td>
<td>The <code>sizeof</code> Operator</td>
<td>156</td>
</tr>
<tr>
<td>4.10</td>
<td>Comma Operator</td>
<td>157</td>
</tr>
<tr>
<td>4.11</td>
<td>Type Conversions</td>
<td>159</td>
</tr>
<tr>
<td>4.11.1</td>
<td>The Arithmetic Conversions</td>
<td>159</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Other Implicit Conversions</td>
<td>161</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Explicit Conversions</td>
<td>162</td>
</tr>
<tr>
<td>4.12</td>
<td>Operator Precedence Table</td>
<td>166</td>
</tr>
</tbody>
</table>

### Chapter Summary

- Defined Terms

## Chapter 5: Statements

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Simple Statements</td>
<td>172</td>
</tr>
<tr>
<td>5.2</td>
<td>Statement Scope</td>
<td>174</td>
</tr>
<tr>
<td>5.3</td>
<td>Conditional Statements</td>
<td>174</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The <code>if</code> Statement</td>
<td>175</td>
</tr>
<tr>
<td>5.3.2</td>
<td>The <code>switch</code> Statement</td>
<td>178</td>
</tr>
<tr>
<td>5.4</td>
<td>Iterative Statements</td>
<td>183</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The <code>while</code> Statement</td>
<td>183</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Traditional <code>for</code> Statement</td>
<td>185</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Range <code>for</code> Statement</td>
<td>187</td>
</tr>
<tr>
<td>5.4.4</td>
<td>The <code>do while</code> Statement</td>
<td>189</td>
</tr>
<tr>
<td>5.5</td>
<td>Jump Statements</td>
<td>190</td>
</tr>
<tr>
<td>5.5.1</td>
<td>The <code>break</code> Statement</td>
<td>190</td>
</tr>
<tr>
<td>5.5.2</td>
<td>The <code>continue</code> Statement</td>
<td>191</td>
</tr>
<tr>
<td>5.5.3</td>
<td>The <code>goto</code> Statement</td>
<td>192</td>
</tr>
<tr>
<td>5.6</td>
<td><code>try</code> Blocks and Exception Handling</td>
<td>193</td>
</tr>
<tr>
<td>5.6.1</td>
<td>A <code>throw</code> Expression</td>
<td>193</td>
</tr>
<tr>
<td>5.6.2</td>
<td>The <code>try</code> Block</td>
<td>194</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Standard Exceptions</td>
<td>197</td>
</tr>
</tbody>
</table>

### Chapter Summary

- Defined Terms

## Chapter 6: Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Function Basics</td>
<td>202</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Local Objects</td>
<td>204</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Function Declarations</td>
<td>206</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Separate Compilation</td>
<td>207</td>
</tr>
<tr>
<td>6.2</td>
<td>Argument Passing</td>
<td>208</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Passing Arguments by Value</td>
<td>209</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Passing Arguments by Reference</td>
<td>210</td>
</tr>
<tr>
<td>6.2.3</td>
<td><code>const</code> Parameters and Arguments</td>
<td>212</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Array Parameters</td>
<td>214</td>
</tr>
</tbody>
</table>

### Chapter Summary

- Defined Terms
Chapter 6 Functions

6.2  Command-Line Options

6.2.5  main: Handling Command-Line Options .................................................. 218

6.2.6  Functions with Varying Parameters ......................................................... 220

6.3  Return Types and the return Statement ...................................................... 222

6.3.1  Functions with No Return Value .................................................................. 223

6.3.2  Functions That Return a Value .................................................................... 223

6.3.3  Returning a Pointer to an Array .................................................................. 228

6.4  Overloaded Functions ...................................................................................... 230

6.4.1  Overloading and Scope ................................................................................ 234

6.5  Features for Specialized Uses ......................................................................... 236

6.5.1  Default Arguments ....................................................................................... 236

6.5.2  Inline and constexpr Functions ................................................................... 238

6.5.3  Aids for Debugging ....................................................................................... 240

6.6  Function Matching ......................................................................................... 242

6.6.1  Argument Type Conversions ...................................................................... 245

6.7  Pointers to Functions ...................................................................................... 247

Chapter Summary ................................................................................................. 251

Defined Terms ....................................................................................................... 251

Chapter 7 Classes ................................................................................................. 253

7.1  Defining Abstract Data Types ......................................................................... 254

7.1.1  Designing the Sales_data Class ................................................................... 254

7.1.2  Defining the Revised Sales_data Class ....................................................... 256

7.1.3  Defining Nonmember Class-Related Functions ........................................... 260

7.1.4  Constructors ................................................................................................. 262

7.1.5  Copy, Assignment, and Destruction .............................................................. 267

7.2  Access Control and Encapsulation .................................................................. 268

7.2.1  Friends .......................................................................................................... 269

7.3  Additional Class Features ............................................................................... 271

7.3.1  Class Members Revisited ............................................................................. 271

7.3.2  Functions That Return *this ......................................................................... 275

7.3.3  Class Types .................................................................................................... 277

7.3.4  Friendship Revisited .................................................................................... 279

7.4  Class Scope ...................................................................................................... 282

7.4.1  Name Lookup and Class Scope ................................................................... 283

7.5  Constructors Revisited .................................................................................... 288

7.5.1  Constructor Initializer List ............................................................................. 288

7.5.2  Delegating Constructors ............................................................................. 291

7.5.3  The Role of the Default Constructor ............................................................ 293

7.5.4  Implicit Class-Type Conversions ................................................................. 294

7.5.5  Aggregate Classes ......................................................................................... 298

7.5.6  Literal Classes ............................................................................................... 299

7.6  static Class Members ...................................................................................... 300

Chapter Summary ................................................................................................. 305

Defined Terms ....................................................................................................... 305
Part II  The C++ Library 307

Chapter 8  The IO Library ................................................. 309
  8.1  The IO Classes ..................................................... 310
        8.1.1  No Copy or Assign for IO Objects ....................... 311
        8.1.2  Condition States ........................................ 312
        8.1.3  Managing the Output Buffer ................................ 314
  8.2  File Input and Output ........................................... 316
        8.2.1  Using File Stream Objects ................................ 317
        8.2.2  File Modes .............................................. 319
  8.3  string Streams .................................................. 321
        8.3.1  Using an istream*stream ................................. 321
        8.3.2  Using ostream*s ......................................... 323
  Chapter Summary ..................................................... 324
  Defined Terms ........................................................ 324

Chapter 9  Sequential Containers ...................................... 325
  9.1  Overview of the Sequential Containers .......................... 326
  9.2  Container Library Overview .................................... 328
        9.2.1  Iterators .................................................. 331
        9.2.2  Container Type Members .................................. 332
        9.2.3  begin and end Members ................................... 333
        9.2.4  Defining and Initializing a Container .................... 334
        9.2.5  Assignment and swap ...................................... 337
        9.2.6  Container Size Operations ................................. 340
        9.2.7  Relational Operators ...................................... 340
  9.3  Sequential Container Operations ................................ 341
        9.3.1  Adding Elements to a Sequential Container ............... 341
        9.3.2  Accessing Elements ....................................... 346
        9.3.3  Erasing Elements .......................................... 348
        9.3.4  Specialized forward_list Operations ...................... 350
        9.3.5  Resizing a Container ...................................... 352
        9.3.6  Container Operations May Invalidate Iterators .......... 353
  9.4  How a vector Grows ............................................... 355
  9.5  Additional string Operations .................................... 360
        9.5.1  Other Ways to Construct strings .......................... 360
        9.5.2  Other Ways to Change a string ........................... 361
        9.5.3  string Search Operations .................................. 364
        9.5.4  The compare Functions .................................... 366
        9.5.5  Numeric Conversions ....................................... 367
  9.6  Container Adaptors ................................................ 368
  Chapter Summary ..................................................... 372
  Defined Terms ........................................................ 372
**Chapter 10 Generic Algorithms** ................................................. 375  
10.1 Overview ............................................................................. 376  
10.2 A First Look at the Algorithms ............................................. 378  
10.2.1 Read-Only Algorithms .................................................. 379  
10.2.2 Algorithms That Write Container Elements ....................... 380  
10.2.3 Algorithms That Reorder Container Elements .................... 383  
10.3 Customizing Operations .................................................... 385  
10.3.1 Passing a Function to an Algorithm .................................. 386  
10.3.2 Lambda Expressions ..................................................... 387  
10.3.3 Lambda Captures and Returns ....................................... 392  
10.3.4 Binding Arguments ..................................................... 397  
10.4 Revisiting Iterators .......................................................... 401  
10.4.1 Insert Iterators .......................................................... 401  
10.4.2 iostream Iterators ...................................................... 403  
10.4.3 Reverse Iterators ....................................................... 407  
10.5 Structure of Generic Algorithms ......................................... 410  
10.5.1 The Five Iterator Categories ......................................... 410  
10.5.2 Algorithm Parameter Patterns ...................................... 412  
10.5.3 Algorithm Naming Conventions ..................................... 413  
10.6 Container-Specific Algorithms ............................................ 415  
Chapter Summary ...................................................................... 417  
Defined Terms ........................................................................... 417

**Chapter 11 Associative Containers** ........................................... 419  
11.1 Using an Associative Container .......................................... 420  
11.2 Overview of the Associative Containers ................................. 423  
11.2.1 Defining an Associative Container ................................... 423  
11.2.2 Requirements on Key Type ........................................... 424  
11.2.3 The pair Type .................................................................. 426  
11.3 Operations on Associative Containers ................................. 428  
11.3.1 Associative Container Iterators ....................................... 429  
11.3.2 Adding Elements ......................................................... 431  
11.3.3 Erasing Elements ......................................................... 434  
11.3.4 Subscripting a map ........................................................ 435  
11.3.5 Accessing Elements ...................................................... 436  
11.3.6 A Word Transformation Map ......................................... 440  
11.4 The Unordered Containers .................................................. 443  
Chapter Summary ...................................................................... 447  
Defined Terms ........................................................................... 447

**Chapter 12 Dynamic Memory** ................................................... 449  
12.1 Dynamic Memory and Smart Pointers .................................. 450  
12.1.1 The shared_ptr Class .................................................... 450  
12.1.2 Managing Memory Directly ............................................. 458  
12.1.3 Using shared_ptrs with new ......................................... 464  
12.1.4 Smart Pointers and Exceptions ....................................... 467  
12.1.5 unique_ptr ................................................................. 470
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1.6 weak_ptr</td>
<td>473</td>
</tr>
<tr>
<td>12.2 Dynamic Arrays</td>
<td>476</td>
</tr>
<tr>
<td>12.2.1 new and Arrays</td>
<td>477</td>
</tr>
<tr>
<td>12.2.2 The allocator Class</td>
<td>481</td>
</tr>
<tr>
<td>12.3 Using the Library: A Text-Query Program</td>
<td>484</td>
</tr>
<tr>
<td>12.3.1 Design of the Query Program</td>
<td>485</td>
</tr>
<tr>
<td>12.3.2 Defining the Query Program Classes</td>
<td>487</td>
</tr>
<tr>
<td>Chapter Summary</td>
<td>491</td>
</tr>
<tr>
<td>Defined Terms</td>
<td>491</td>
</tr>
</tbody>
</table>

**Part III** Tools for Class Authors 493

**Chapter 13 Copy Control** 495

13.1 Copy, Assign, and Destroy 496
   13.1.1 The Copy Constructor 496
   13.1.2 The Copy-Assignment Operator 500
   13.1.3 The Destructor 501
   13.1.4 The Rule of Three/Five 503
   13.1.5 Using = default 506
   13.1.6 Preventing Copies 507
13.2 Copy Control and Resource Management 510
   13.2.1 Classes That Act Like Values 511
   13.2.2 Defining Classes That Act Like Pointers 513
13.3 Swap 516
13.4 A Copy-Control Example 519
13.5 Classes That Manage Dynamic Memory 524
13.6 Moving Objects 531
   13.6.1 Rvalue References 532
   13.6.2 Move Constructor and Move Assignment 534
   13.6.3 Rvalue References and Member Functions 544
Chapter Summary 549
Defined Terms 549

**Chapter 14 Overloaded Operations and Conversions** 551

14.1 Basic Concepts 552
14.2 Input and Output Operators 556
   14.2.1 Overloading the Output Operator << 557
   14.2.2 Overloading the Input Operator >> 558
14.3 Arithmetic and Relational Operators 560
   14.3.1 Equality Operators 561
   14.3.2 Relational Operators 562
14.4 Assignment Operators 563
14.5 Subscript Operator 564
14.6 Increment and Decrement Operators 566
14.7 Member Access Operators 569
14.8 Function-Call Operator 571
### Library of Function Objects

- **14.8.1** Lambdas Are Function Objects .................................................. 572
- **14.8.2** Library-Defined Function Objects ................................................. 574
- **14.8.3** Callable Objects and function ..................................................... 576

### Overloading, Conversions, and Operators

- **14.9** Overloading, Conversions, and Operators ......................................... 579
  - **14.9.1** Conversion Operators ................................................................. 580
  - **14.9.2** Avoiding Ambiguous Conversions ............................................... 583
  - **14.9.3** Function Matching and Overloaded Operators ..................... 587

**Chapter Summary** ................................................................. 590
**Defined Terms** ................................................................. 590

---

### Chapter 15 Object-Oriented Programming

- **15.1** OOP: An Overview ................................................................. 592
- **15.2** Defining Base and Derived Classes ............................................. 594
  - **15.2.1** Defining a Base Class ............................................................... 594
  - **15.2.2** Defining a Derived Class ............................................................ 596
  - **15.2.3** Conversions and Inheritance .................................................... 601
- **15.3** Virtual Functions ................................................................. 603
- **15.4** Abstract Base Classes ............................................................... 608
- **15.5** Access Control and Inheritance .................................................... 611
- **15.6** Class Scope under Inheritance ..................................................... 617
- **15.7** Constructors and Copy Control .................................................... 622
  - **15.7.1** Virtual Destructors ................................................................. 622
  - **15.7.2** Synthesized Copy Control and Inheritance .......................... 623
  - **15.7.3** Derived-Class Copy-Control Members ...................................... 625
  - **15.7.4** Inherited Constructors ............................................................. 628
- **15.8** Containers and Inheritance ........................................................ 630
  - **15.8.1** Writing a `Basket` Class .......................................................... 631
- **15.9** Text Queries Revisited .............................................................. 634
  - **15.9.1** An Object-Oriented Solution .................................................... 636
  - **15.9.2** The `Query_base` and `Query` Classes ...................................... 639
  - **15.9.3** The Derived Classes ................................................................. 642
  - **15.9.4** The `eval` Functions ............................................................... 645

**Chapter Summary** ................................................................. 649
**Defined Terms** ................................................................. 649

---

### Chapter 16 Templates and Generic Programming

- **16.1** Defining a Template ................................................................. 652
  - **16.1.1** Function Templates ................................................................. 652
  - **16.1.2** Class Templates ................................................................. 658
  - **16.1.3** Template Parameters ............................................................. 668
  - **16.1.4** Member Templates ................................................................. 672
  - **16.1.5** Controlling Instantiations ...................................................... 675
  - **16.1.6** Efficiency and Flexibility .................................................... 676
- **16.2** Template Argument Deduction ..................................................... 678
  - **16.2.1** Conversions and Template Type Parameters .......................... 679
  - **16.2.2** Function-Template Explicit Arguments ..................................... 681
  - **16.2.3** Trailing Return Types and Type Transformation .................. 683
Chapter 18 Specialized Tools and Techniques ........................................ 802
  18.2.2 Using Namespace Members ........................................ 792
  18.2.3 Classes, Namespaces, and Scope .................................. 796
  18.2.4 Overloading and Namespaces ...................................... 800
  18.3 Multiple and Virtual Inheritance .................................... 802
    18.3.1 Multiple Inheritance ........................................ 803
    18.3.2 Conversions and Multiple Base Classes ....................... 805
    18.3.3 Class Scope under Multiple Inheritance ....................... 807
    18.3.4 Virtual Inheritance ........................................ 810
    18.3.5 Constructors and Virtual Inheritance ......................... 813
  Chapter Summary ......................................................... 816
  Defined Terms .................................................................... 816

Chapter 19 Specialized Tools and Techniques ....................................... 819
  19.1 Controlling Memory Allocation ....................................... 820
    19.1.1 Overloading new and delete ................................ 820
    19.1.2 Placement new Expressions .................................... 823
  19.2 Run-Time Type Identification ....................................... 825
    19.2.1 The dynamic_cast Operator ................................... 825
    19.2.2 The typeid Operator ........................................ 826
    19.2.3 Using RTTI ................................................ 828
    19.2.4 The type_info Class ......................................... 831
  19.3 Enumerations .......................................................... 832
  19.4 Pointer to Class Member .............................................. 835
    19.4.1 Pointers to Data Members .................................... 836
    19.4.2 Pointers to Member Functions ................................ 838
    19.4.3 Using Member Functions as Callable Objects ................. 841
  19.5 Nested Classes ........................................................ 843
  19.6 union: A Space-Saving Class ........................................ 847
  19.7 Local Classes .......................................................... 852
  19.8 Inherently Nonportable Features .................................... 854
    19.8.1 Bit-fields ....................................................... 854
    19.8.2 volatile Qualifier ............................................. 856
    19.8.3 Linkage Directives: extern "C" .................................. 857
  Chapter Summary ......................................................... 862
  Defined Terms .................................................................... 862

Appendix A The Library ..................................................................... 865
  A.1 Library Names and Headers ............................................ 866
  A.2 A Brief Tour of the Algorithms ...................................... 870
    A.2.1 Algorithms to Find an Object ................................ 871
    A.2.2 Other Read-Only Algorithms ................................... 872
    A.2.3 Binary Search Algorithms ..................................... 873
    A.2.4 Algorithms That Write Container Elements ................. 873
    A.2.5 Partitioning and Sorting Algorithms ......................... 875
    A.2.6 General Reordering Operations ................................. 877
    A.2.7 Permutation Algorithms ....................................... 879
    A.2.8 Set Algorithms for Sorted Sequences ......................... 880
Contents

A.2.9 Minimum and Maximum Values ........................................... 880
A.2.10 Numeric Algorithms ..................................................... 881
A.3 Random Numbers ............................................................ 882
  A.3.1 Random Number Distributions ....................................... 883
  A.3.2 Random Number Engines .............................................. 884

Index ................................................................................. 887
This page intentionally left blank
New Features in C++11

2.1.1 long long Type ............................................. 33
2.2.1 List Initialization ........................................... 43
2.3.2 nullptr Literal ............................................. 54
2.4.4 constexpr Variables ......................................... 66
2.5.1 Type Alias Declarations ..................................... 68
2.5.2 The auto Type Specifier ...................................... 68
2.5.3 The decltype Type Specifier ................................. 70
2.6.1 In-Class Initializers ......................................... 73
3.2.2 Using auto or decltype for Type Abbreviation ............. 88
3.2.3 Range for Statement .......................................... 91
3.3 Defining a vector of vectors .................................... 97
3.3.1 List Initialization for vectors .............................. 98
3.4.1 Container cbegin and cend Functions ...................... 109
3.5.3 Library begin and end Functions ............................ 118
3.6 Using auto or decltype to Simplify Declarations .......... 129
4.2 Rounding Rules for Division .................................... 141
4.4 Assignment from a Braced List of Values ..................... 145
4.9 sizeof Applied to a Class Member ............................. 157
5.4.3 Range for Statement .......................................... 187
6.2.6 Library initializer_list Class .............................. 220
6.3.2 List Initializing a Return Value ............................. 226
6.3.3 Declaring a Trailing Return Type ........................... 229
6.3.3 Using decltype to Simplify Return Type Declarations ... 230
6.5.2 constexpr Functions .......................................... 239
7.1.4 Using = default to Generate a Default Constructor .... 265
7.3.1 In-class Initializers for Members of Class Type .......... 274
7.5.2 Delegating Constructors ..................................... 291
7.5.6 constexpr Constructors ....................................... 299
8.2.1 Using strings for File Names .................................. 317
9.1 The array and forward_list Containers ....................... 327
9.2.3 Container cbegin and cend Functions ..................... 334
9.2.4 List Initialization for Containers .......................... 336
9.2.5 Container Nonmember swap Functions ...................... 339
9.3.1 Return Type for Container insert Members ................. 344
9.3.1 Container emplace Members ................................. 345
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.4 shrink_to_fit</td>
<td>357</td>
</tr>
<tr>
<td>9.5.5 Numeric Conversion Functions for strings</td>
<td>367</td>
</tr>
<tr>
<td>10.3.2 Lambda Expressions</td>
<td>388</td>
</tr>
<tr>
<td>10.3.3 Trailing Return Type in Lambda Expressions</td>
<td>396</td>
</tr>
<tr>
<td>10.3.4 The Library bind Function</td>
<td>397</td>
</tr>
<tr>
<td>11.2.1 List Initialization of an Associative Container</td>
<td>423</td>
</tr>
<tr>
<td>11.2.3 List Initializing pair Return Type</td>
<td>427</td>
</tr>
<tr>
<td>11.3.2 List Initialization of a pair</td>
<td>431</td>
</tr>
<tr>
<td>11.4 The Unordered Containers</td>
<td>443</td>
</tr>
<tr>
<td>12.1 Smart Pointers</td>
<td>450</td>
</tr>
<tr>
<td>12.1.1 The shared_ptr Class</td>
<td>450</td>
</tr>
<tr>
<td>12.1.2 List Initialization of Dynamically Allocated Objects</td>
<td>459</td>
</tr>
<tr>
<td>12.1.2 auto and Dynamic Allocation</td>
<td>459</td>
</tr>
<tr>
<td>12.1.5 The unique_ptr Class</td>
<td>470</td>
</tr>
<tr>
<td>12.1.6 The weak_ptr Class</td>
<td>473</td>
</tr>
<tr>
<td>12.2.1 Range for Doesn’t Apply to Dynamically Allocated Arrays</td>
<td>477</td>
</tr>
<tr>
<td>12.2.1 List Initialization of Dynamically Allocated Arrays</td>
<td>478</td>
</tr>
<tr>
<td>12.2.1 auto Can’t Be Used to Allocate an Array</td>
<td>478</td>
</tr>
<tr>
<td>12.2.2 allocator::construct Can Use any Constructor</td>
<td>482</td>
</tr>
<tr>
<td>13.1.5 Using = default for Copy-Control Members</td>
<td>506</td>
</tr>
<tr>
<td>13.1.6 Using = delete to Prevent Copying Class Objects</td>
<td>507</td>
</tr>
<tr>
<td>13.5 Moving Instead of Copying Class Objects</td>
<td>529</td>
</tr>
<tr>
<td>13.6.1 Rvalue References</td>
<td>532</td>
</tr>
<tr>
<td>13.6.1 The Library move Function</td>
<td>533</td>
</tr>
<tr>
<td>13.6.2 Move Constructor and Move Assignment</td>
<td>534</td>
</tr>
<tr>
<td>13.6.2 Move Constructors Usually Should Be noexcept</td>
<td>535</td>
</tr>
<tr>
<td>13.6.2 Move Iterators</td>
<td>543</td>
</tr>
<tr>
<td>13.6.3 Reference Qualified Member Functions</td>
<td>546</td>
</tr>
<tr>
<td>14.8.3 The function Class Template</td>
<td>577</td>
</tr>
<tr>
<td>14.9.1 explicit Conversion Operators</td>
<td>582</td>
</tr>
<tr>
<td>15.2.2 override Specifier for Virtual Functions</td>
<td>596</td>
</tr>
<tr>
<td>15.2.2 Preventing Inheritance by Defining a Class as final</td>
<td>600</td>
</tr>
<tr>
<td>15.3 override and final Specifiers for Virtual Functions</td>
<td>606</td>
</tr>
<tr>
<td>15.7.2 Deleted Copy Control and Inheritance</td>
<td>624</td>
</tr>
<tr>
<td>15.7.4 Inherited Constructors</td>
<td>628</td>
</tr>
<tr>
<td>16.1.2 Declaring a Template Type Parameter as a Friend</td>
<td>666</td>
</tr>
<tr>
<td>16.1.2 Template Type Aliases</td>
<td>666</td>
</tr>
<tr>
<td>16.1.3 Default Template Arguments for Template Functions</td>
<td>670</td>
</tr>
<tr>
<td>16.1.5 Explicit Control of Instantiation</td>
<td>675</td>
</tr>
<tr>
<td>16.2.3 Template Functions and Trailing Return Types</td>
<td>684</td>
</tr>
<tr>
<td>16.2.5 Reference Collapsing Rules</td>
<td>688</td>
</tr>
<tr>
<td>16.2.6 static_cast from an Lvalue to an Rvalue</td>
<td>691</td>
</tr>
<tr>
<td>16.2.7 The Library forward Function</td>
<td>694</td>
</tr>
<tr>
<td>16.4 Variadic Templates</td>
<td>699</td>
</tr>
<tr>
<td>16.4 The sizeof... Operator</td>
<td>700</td>
</tr>
<tr>
<td>16.4.3 Variadic Templates and Forwarding</td>
<td>704</td>
</tr>
</tbody>
</table>
17.1 The Library Tuple Class Template .................. 718
17.2.2 New bitset Operations .......................... 726
17.3 The Regular Expression Library ..................... 728
17.4 The Random Number Library ........................ 745
17.5.1 Floating-Point Format Control ................... 757
18.1.4 The noexcept Exception Specifier ................ 779
18.1.4 The noexcept Operator ........................... 780
18.2.1 Inline Namespaces ............................... 790
18.3.1 Inherited Constructors and Multiple Inheritance ..... 804
19.3 Scoped enums ..................................... 832
19.3 Specifying the Type Used to Hold an enum .......... 834
19.3 Forward Declarations for enums ..................... 834
19.4.3 The Library mem_fn Class Template .............. 843
19.6 Union Members of Class Types ...................... 848
Countless programmers have learned C++ from previous editions of C++ Primer. During that time, C++ has matured greatly: Its focus, and that of its programming community, has widened from looking mostly at machine efficiency to devoting more attention to programmer efficiency.

In 2011, the C++ standards committee issued a major revision to the ISO C++ standard. This revised standard is latest step in C++’s evolution and continues the emphasis on programmer efficiency. The primary goals of the new standard are to

- Make the language more uniform and easier to teach and to learn
- Make the standard libraries easier, safer, and more efficient to use
- Make it easier to write efficient abstractions and libraries

In this edition, we have completely revised the C++ Primer to use the latest standard. You can get an idea of how extensively the new standard has affected C++ by reviewing the New Features Table of Contents, which lists the sections that cover new material and appears on page xxi.

Some additions in the new standard, such as auto for type inference, are pervasive. These facilities make the code in this edition easier to read and to understand. Programs (and programmers!) can ignore type details, which makes it easier to concentrate on what the program is intended to do. Other new features, such as smart pointers and move-enabled containers, let us write more sophisticated classes without having to contend with the intricacies of resource management. As a result, we can start to teach how to write your own classes much earlier in the book than we did in the Fourth Edition. We—and you—no longer have to worry about many of the details that stood in our way under the previous standard.

We’ve marked those parts of the text that cover features defined by the new standard, with a marginal icon. We hope that readers who are already familiar with the core of C++ will find these alerts useful in deciding where to focus their attention. We also expect that these icons will help explain error messages from compilers that might not yet support every new feature. Although nearly all of the examples in this book have been compiled under the current release of the GNU compiler, we realize some readers will not yet have access to completely updated compilers. Even though numerous capabilities have been added by the latest standard, the core language remains unchanged and forms the bulk of the material that we cover. Readers can use these icons to note which capabilities may not yet be available in their compiler.
Why Read This Book?

Modern C++ can be thought of as comprising three parts:

- The low-level language, much of which is inherited from C
- More advanced language features that allow us to define our own types and to organize large-scale programs and systems
- The standard library, which uses these advanced features to provide useful data structures and algorithms

Most texts present C++ in the order in which it evolved. They teach the C subset of C++ first, and present the more abstract features of C++ as advanced topics at the end of the book. There are two problems with this approach: Readers can get bogged down in the details inherent in low-level programming and give up in frustration. Those who do press on learn bad habits that they must unlearn later.

We take the opposite approach: Right from the start, we use the features that let programmers ignore the details inherent in low-level programming. For example, we introduce and use the library string and vector types along with the built-in arithmetic and array types. Programs that use these library types are easier to write, easier to understand, and much less error-prone.

Too often, the library is taught as an “advanced” topic. Instead of using the library, many books use low-level programming techniques based on pointers to character arrays and dynamic memory management. Getting programs that use these low-level techniques to work correctly is much harder than writing the corresponding C++ code using the library.

Throughout C++ Primer, we emphasize good style: We want to help you, the reader, develop good habits immediately and avoid needing to unlearn bad habits as you gain more sophisticated knowledge. We highlight particularly tricky matters and warn about common misconceptions and pitfalls.

We also explain the rationale behind the rules—explaining the why not just the what. We believe that by understanding why things work as they do, readers can more quickly cement their grasp of the language.

Although you do not need to know C in order to understand this book, we assume you know enough about programming to write, compile, and run a program in at least one modern block-structured language. In particular, we assume you have used variables, written and called functions, and used a compiler.

Changes to the Fifth Edition

New to this edition of C++ Primer are icons in the margins to help guide the reader. C++ is a large language that offers capabilities tailored to particular kinds of programming problems. Some of these capabilities are of great import for large project teams but might not be necessary for smaller efforts. As a result, not every programmer needs to know every detail of every feature. We’ve added these marginal icons to help the reader know which parts can be learned later and which topics are more essential.

We’ve marked sections that cover the fundamentals of the language with an image of a person studying a book. The topics covered in sections marked this
way form the core part of the language. Everyone should read and understand
these sections.

We’ve also indicated those sections that cover advanced or special-purpose top-
ics. These sections can be skipped or skimmed on a first reading. We’ve marked
such sections with a stack of books to indicate that you can safely put down the
book at that point. It is probably a good idea to skim such sections so you know
that the capability exists. However, there is no reason to spend time studying these
topics until you actually need to use the feature in your own programs.

To help readers guide their attention further, we’ve noted particularly tricky
concepts with a magnifying-glass icon. We hope that readers will take the time
to understand thoroughly the material presented in the sections so marked. In at
least some of these sections, the import of the topic may not be readily apparent;
but we think you’ll find that these sections cover topics that turn out to be essential
to understanding the language.

Another aid to reading this book, is our extensive use of cross-references. We
hope these references will make it easier for readers to dip into the middle of the
book, yet easily jump back to the earlier material on which later examples rely.

What remains unchanged is that C++ Primer is a clear, correct, and thorough
tutorial guide to C++. We teach the language by presenting a series of increasingly
sophisticated examples, which explain language features and show how to make
the best use of C++.

Structure of This Book

We start by covering the basics of the language and the library together in Parts I
and II. These parts cover enough material to let you, the reader, write significant
programs. Most C++ programmers need to know essentially everything covered
in this portion of the book.

In addition to teaching the basics of C++, the material in Parts I and II serves an-
other important purpose: By using the abstract facilities defined by the library, you
will become more comfortable with using high-level programming techniques.
The library facilities are themselves abstract data types that are usually written
in C++. The library can be defined using the same class-construction features that
are available to any C++ programmer. Our experience in teaching C++ is that by
first using well-designed abstract types, readers find it easier to understand how
to build their own types.

Only after a thorough grounding in using the library—and writing the kinds of
abstract programs that the library allows—do we move on to those C++ features
that will enable you to write your own abstractions. Parts III and IV focus on
writing abstractions in the form of classes. Part III covers the fundamentals; Part IV
covers more specialized facilities.

In Part III, we cover issues of copy control, along with other techniques to make
classes that are as easy to use as the built-in types. Classes are the foundation for
object-oriented and generic programming, which we also cover in Part III. C++
Primer concludes with Part IV, which covers features that are of most use in struc-
turing large, complicated systems. We also summarize the library algorithms in
Appendix A.
Aids to the Reader

Each chapter concludes with a summary, followed by a glossary of defined terms, which together recap the chapter’s most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

We’ve also incorporated a number of other learning aids in the body of the text:

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in *bold italics*. Each term appears in the chapter’s Defined Terms section.

- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips.

- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.

- We provide sidebar discussions on important concepts and for topics that new C++ programmers often find most difficult.

- Learning any programming language requires writing programs. To that end, the Primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

  http://www.informit.com/title/0321714113

A Note about Compilers

As of this writing (July, 2012), compiler vendors are hard at work updating their compilers to match the latest ISO standard. The compiler we use most frequently is the GNU compiler, version 4.7.0. There are only a few features used in this book that this compiler does not yet implement: inheriting constructors, reference qualifiers for member functions, and the regular-expression library.

Acknowledgments

In preparing this edition we are very grateful for the help of several current and former members of the standardization committee: Dave Abrahams, Andy Koenig, Stephan T. Lavavej, Jason Merrill, John Spicer, and Herb Sutter. They provided invaluable assistance to us in understanding some of the more subtle parts of the new standard. We’d also like to thank the many folks who worked on updating the GNU compiler making the standard a reality.

As in previous editions of C++ Primer, we’d like to extend our thanks to Bjarne Stroustrup for his tireless work on C++ and for his friendship to the authors during most of that time. We’d also like to thank Alex Stepanov for his original insights that led to the containers and algorithms at the core of the standard library. Finally, our thanks go to all the C++ Standards committee members for their hard work in clarifying, refining, and improving C++ over many years.
We extend our deep-felt thanks to our reviewers, whose helpful comments led us to make improvements great and small throughout the book: Marshall Clow, Jon Kalb, Nevin Liber, Dr. C. L. Tondo, Daveed Vandevoorde, and Steve Vinoski.

This book was typeset using \LaTeX\ and the many packages that accompany the \LaTeX\ distribution. Our well-justified thanks go to the members of the \LaTeX\ community, who have made available such powerful typesetting tools.

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Peter Gordon, our editor, who provided the impetus for us to revise \textit{C++ Primer} once again; Kim Boedigheimer, who keeps us all on schedule; Barbara Wood, who found lots of editing errors for us during the copy-edit phase, and Elizabeth Ryan, who was again a delight to work with as she guided us through the design and production process.
This page intentionally left blank
The programs we’ve written so far have used objects that have well-defined lifetimes. Global objects are allocated at program start-up and destroyed when the program ends. Local, automatic objects are created and destroyed when the block in which they are defined is entered and exited. Local static objects are allocated before their first use and are destroyed when the program ends.

In addition to supporting automatic and static objects, C++ lets us allocate objects dynamically. Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed.

Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs. To make using dynamic objects safer, the library defines two smart pointer types that manage dynamically allocated objects. Smart pointers ensure that the objects to which they point are automatically freed when it is appropriate to do so.
Our programs have used only static or stack memory. Static memory is used for local static objects (§ 6.1.1, p. 205), for class static data members (§ 7.6, p. 300), and for variables defined outside any function. Stack memory is used for nonstatic objects defined inside functions. Objects allocated in static or stack memory are automatically created and destroyed by the compiler. Stack objects exist only while the block in which they are defined is executing; static objects are allocated before they are used, and they are destroyed when the program ends.

In addition to static or stack memory, every program also has a pool of memory that it can use. This memory is referred to as the free store or heap. Programs use the heap for objects that they dynamically allocate—that is, for objects that the program allocates at run time. The program controls the lifetime of dynamic objects; our code must explicitly destroy such objects when they are no longer needed.

Although necessary at times, dynamic memory is notoriously tricky to manage correctly.

12.1 Dynamic Memory and Smart Pointers

In C++, dynamic memory is managed through a pair of operators: new, which allocates, and optionally initializes, an object in dynamic memory and returns a pointer to that object; and delete, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory.

Dynamic memory is problematic because it is surprisingly hard to ensure that we free memory at the right time. Either we forget to free the memory—in which case we have a memory leak—or we free the memory when there are still pointers referring to that memory—in which case we have a pointer that refers to memory that is no longer valid.

To make using dynamic memory easier (and safer), the new library provides two smart pointer types that manage dynamic objects. A smart pointer acts like a regular pointer with the important exception that it automatically deletes the object to which it points. The new library defines two kinds of smart pointers that differ in how they manage their underlying pointers: shared_ptr, which allows multiple pointers to refer to the same object, and unique_ptr, which “owns” the object to which it points. The library also defines a companion class named weak_ptr that is a weak reference to an object managed by a shared_ptr. All three are defined in the memory header.

12.1.1 The shared_ptr Class

Like vectors, smart pointers are templates (§ 3.3, p. 96). Therefore, when we create a smart pointer, we must supply additional information—in this case, the type to which the pointer can point. As with vector, we supply that type inside angle brackets that follow the name of the kind of smart pointer we are defining:

```cpp
shared_ptr<string> p1;  // shared_ptr that can point at a string
```
shared_ptr<list<int>> p2; // shared_ptr that can point at a list of ints

A default initialized smart pointer holds a null pointer (§ 2.3.2, p. 53). In § 12.1.3 (p. 464), we’ll cover additional ways to initialize a smart pointer.

We use a smart pointer in ways that are similar to using a pointer. Dereferencing a smart pointer returns the object to which the pointer points. When we use a smart pointer in a condition, the effect is to test whether the pointer is null:

```cpp
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi"; // if so, dereference p1 to assign a new value to that string
```

Table 12.1 (overleaf) lists operations common to shared_ptr and unique_ptr. Those that are particular to shared_ptr are listed in Table 12.2 (p. 453).

**The make_shared Function**

The safest way to allocate and use dynamic memory is to call a library function named make_shared. This function allocates and initializes an object in dynamic memory and returns a shared_ptr that points to that object. Like the smart pointers, make_shared is defined in the memory header.

When we call make_shared, we must specify the type of object we want to create. We do so in the same way as we use a template class, by following the function name with a type enclosed in angle brackets:

```cpp
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);

// p4 points to a string with value "9999999999"
shared_ptr<string> p4 = make_shared<string>(10, '9');

// p5 points to an int that is value initialized (§ 3.3.1 (p. 98)) to 0
shared_ptr<int> p5 = make_shared<int>();
```

Like the sequential-container emplace members (§ 9.3.1, p. 345), make_shared uses its arguments to construct an object of the given type. For example, a call to make_shared<string> must pass argument(s) that match one of the string constructors. Calls to make_shared<int> can pass any value we can use to initialize an int. And so on. If we do not pass any arguments, then the object is value initialized (§ 3.3.1, p. 98).

Of course, ordinarily we use auto (§ 2.5.2, p. 68) to make it easier to define an object to hold the result of make_shared:

```cpp
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
```

**Copying and Assigning shared_ptrs**

When we copy or assign a shared_ptr, each shared_ptr keeps track of how many other shared_ptrs point to the same object:

```cpp
auto p = make_shared<int>(42); // object to which p points has one user
auto q(p); // p and q point to the same object
           // object to which p and q point has two users
```
### Table 12.1: Operations Common to `shared_ptr` and `unique_ptr`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shared_ptr&lt;T&gt; sp</code></td>
<td>Null smart pointer that can point to objects of type T.</td>
</tr>
<tr>
<td><code>unique_ptr&lt;T&gt; up</code></td>
<td>Use <code>p</code> as a condition; true if <code>p</code> points to an object.</td>
</tr>
<tr>
<td><code>p</code></td>
<td>Dereference <code>p</code> to get the object to which <code>p</code> points.</td>
</tr>
<tr>
<td><code>p-&gt;mem</code></td>
<td>Returns the pointer in <code>p</code>. Use with caution; the object to which the</td>
</tr>
<tr>
<td></td>
<td>returned pointer points will disappear when the smart pointer deletes it.</td>
</tr>
<tr>
<td><code>p.swap(q)</code></td>
<td>Swaps the pointers in <code>p</code> and <code>q</code>.</td>
</tr>
</tbody>
</table>

We can think of a `shared_ptr` as if it has an associated counter, usually referred to as a **reference count**. Whenever we copy a `shared_ptr`, the count is incremented. For example, the counter associated with a `shared_ptr` is incremented when we use it to initialize another `shared_ptr`, when we use it as the right-hand operand of an assignment, or when we pass it to (§ 6.2.1, p. 209) or return it from a function by value (§ 6.3.2, p. 224). The counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed, such as when a local `shared_ptr` goes out of scope (§ 6.1.1, p. 204).

Once a `shared_ptr`’s counter goes to zero, the `shared_ptr` automatically frees the object that it manages:

```cpp
auto r = make_shared<int>(42); // int to which r points has one user
r = q; // assign to r, making it point to a different address
// increase the use count for the object to which q points
// reduce the use count of the object to which r had pointed
// the object r had pointed to has no users; that object is automatically freed
```

Here we allocate an `int` and store a pointer to that `int` in `r`. Next, we assign a new value to `r`. In this case, `r` is the only `shared_ptr` pointing to the one we previously allocated. That `int` is automatically freed as part of assigning `q` to `r`.

It is up to the implementation whether to use a counter or another data structure to keep track of how many pointers share state. The key point is that the class keeps track of how many `shared_ptrs` point to the same object and automatically frees that object when appropriate.

**shared_ptrs Automatically Destroy Their Objects . . .**

When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` class automatically destroys the object to which that `shared_ptr` points. It does so through another special member function known as a **destructor**. Analogous to its constructors, each class has a destructor. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed.
### Table 12.2: Operations Specific to `shared_ptr`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>make_shared&lt;T&gt;(args)</code></td>
<td>Returns a <code>shared_ptr</code> pointing to a dynamically allocated object of type <code>T</code>. Uses <code>args</code> to initialize that object.</td>
</tr>
<tr>
<td><code>shared_ptr&lt;T&gt;p(q)</code></td>
<td><code>p</code> is a copy of the <code>shared_ptr</code> <code>q</code>; increments the count in <code>q</code>. The pointer in <code>q</code> must be convertible to <code>T*</code> (§ 4.11.2, p. 161).</td>
</tr>
<tr>
<td><code>p = q</code></td>
<td><code>p</code> and <code>q</code> are <code>shared_ptr</code>s holding pointers that can be converted to one another. Decrements <code>p</code>'s reference count and increments <code>q</code>'s count; deletes <code>p</code>'s existing memory if <code>p</code>'s count goes to 0.</td>
</tr>
<tr>
<td><code>p.unique()</code></td>
<td>Returns <code>true</code> if <code>p.use_count()</code> is one; <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>p.use_count()</code></td>
<td>Returns the number of objects sharing with <code>p</code>; may be a slow operation, intended primarily for debugging purposes.</td>
</tr>
</tbody>
</table>

Destructors generally free the resources that an object has allocated. For example, the `string` constructors (and other `string` members) allocate memory to hold the characters that compose the `string`. The `string` destructor frees that memory. Similarly, several `vector` operations allocate memory to hold the elements in the `vector`. The destructor for `vector` destroys those elements and frees the memory used for the elements.

The destructor for `shared_ptr` decrements the reference count of the object to which that `shared_ptr` points. If the count goes to zero, the `shared_ptr` destructor destroys the object to which the `shared_ptr` points and frees the memory used by that object.

### ...and Automatically Free the Associated Memory

The fact that the `shared_ptr` class automatically frees dynamic objects when they are no longer needed makes it fairly easy to use dynamic memory. For example, we might have a function that returns a `shared_ptr` to a dynamically allocated object of a type named `Foo` that can be initialized by an argument of type `T`:

```cpp
// factory returns a shared_ptr pointing to a dynamically allocated object
shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate
    // shared_ptr will take care of deleting this memory
    return make_shared<Foo>(arg);
}
```

Because `factory` returns a `shared_ptr`, we can be sure that the object allocated by `factory` will be freed when appropriate. For example, the following function stores the `shared_ptr` returned by `factory` in a local variable:

```cpp
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory to which p points is automatically freed
```
Because \( p \) is local to \( \text{use\_factory} \), it is destroyed when \( \text{use\_factory} \) ends (§ 6.1.1, p. 204). When \( p \) is destroyed, its reference count is decremented and checked. In this case, \( p \) is the only object referring to the memory returned by \( \text{factory} \). Because \( p \) is about to go away, the object to which \( p \) points will be destroyed and the memory in which that object resides will be freed.

The memory will not be freed if there is any other \( \text{shared\_ptr} \) pointing to it:

```cpp
shared_ptr<
Foo> use_factory(T arg)
{
    shared_ptr<
Foo> p = factory(arg);
    // use p
    return p;  // reference count is incremented when we return p
}  // p goes out of scope; the memory to which p points is not freed
```

In this version, the return statement in \( \text{use\_factory} \) returns a copy of \( p \) to its caller (§ 6.3.2, p. 224). Copying a \( \text{shared\_ptr} \) adds to the reference count of that object. Now when \( p \) is destroyed, there will be another user for the memory to which \( p \) points. The \( \text{shared\_ptr} \) class ensures that so long as there are any \( \text{shared\_ptrs} \) attached to that memory, the memory itself will not be freed.

Because memory is not freed until the last \( \text{shared\_ptr} \) goes away, it can be important to be sure that \( \text{shared\_ptrs} \) don’t stay around after they are no longer needed. The program will execute correctly but may waste memory if you neglect to destroy \( \text{shared\_ptrs} \) that the program does not need. One way that \( \text{shared\_ptrs} \) might stay around after you need them is if you put \( \text{shared\_ptrs} \) in a container and subsequently reorder the container so that you don’t need all the elements. You should be sure to \text{erase} \( \text{shared\_ptr} \) elements once you no longer need those elements.

### Classes with Resources That Have Dynamic Lifetime

Programs tend to use dynamic memory for one of three purposes:

1. They don’t know how many objects they’ll need
2. They don’t know the precise type of the objects they need
3. They want to share data between several objects

The container classes are an example of classes that use dynamic memory for the first purpose and we’ll see examples of the second in Chapter 15. In this section, we’ll define a class that uses dynamic memory in order to let several objects share the same underlying data.

So far, the classes we’ve used allocate resources that exist only as long as the corresponding objects. For example, each \text{vector} “owns” its own elements. When we copy a \text{vector}, the elements in the original \text{vector} and in the copy are separate from one another:
vector<string> v1;  // empty vector
{  // new scope
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2;  // copies the elements from v2 into v1
}  // v2 is destroyed, which destroys the elements in v2
// v1 has three elements, which are copies of the ones originally in v2

The elements allocated by a vector exist only while the vector itself exists. When a vector is destroyed, the elements in the vector are also destroyed.

Some classes allocate resources with a lifetime that is independent of the original object. As an example, assume we want to define a class named Blob that will hold a collection of elements. Unlike the containers, we want Blob objects that are copies of one another to share the same elements. That is, when we copy a Blob, the original and the copy should refer to the same underlying elements.

In general, when two objects share the same underlying data, we can’t unilaterally destroy the data when an object of that type goes away:

Blob<string> b1;  // empty Blob
{  // new scope
    Blob<string> b2 = {"a", "an", "the"};
    b1 = b2;  // b1 and b2 share the same elements
}  // b2 is destroyed, but the elements in b2 must not be destroyed
// b1 points to the elements originally created in b2

In this example, b1 and b2 share the same elements. When b2 goes out of scope, those elements must stay around, because b1 is still using them.

One common reason to use dynamic memory is to allow multiple objects to share the same state.

### Defining the StrBlob Class

Ultimately, we’ll implement our Blob class as a template, but we won’t learn how to do so until § 16.1.2 (p. 658). For now, we’ll define a version of our class that can manage strings. As a result, we’ll name this version of our class StrBlob.

The easiest way to implement a new collection type is to use one of the library containers to manage the elements. That way, we can let the library type manage the storage for the elements themselves. In this case, we’ll use a vector to hold our elements.

However, we can’t store the vector directly in a Blob object. Members of an object are destroyed when the object itself is destroyed. For example, assume that b1 and b2 are two Blobs that share the same vector. If that vector were stored in one of those Blobs—say, b2—then that vector, and therefore its elements, would no longer exist once b2 goes out of scope. To ensure that the elements continue to exist, we’ll store the vector in dynamic memory.

To implement the sharing we want, we’ll give each StrBlob a shared_ptr to a dynamically allocated vector. That shared_ptr member will keep track of how many StrBlobs share the same vector and will delete the vector when the last StrBlob using that vector is destroyed.
We still need to decide what operations our class will provide. For now, we'll implement a small subset of the vector operations. We'll also change the operations that access elements (e.g., front and back): In our class, these operations will throw an exception if a user attempts to access an element that doesn't exist.

Our class will have a default constructor and a constructor that has a parameter of type initializer_list<string> (§ 6.2.6, p. 220). This constructor will take a braced list of initializers.

```cpp
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const std::string &t) { data->push_back(t); }
    void pop_back();
    // element access
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
```

Inside the class we implemented the size, empty, and push_back members. These members forward their work through the data pointer to the underlying vector. For example, size() on a StrBlob calls data->size(), and so on.

**StrBlob Constructors**

Each constructor uses its constructor initializer list (§ 7.1.4, p. 265) to initialize its data member to point to a dynamically allocated vector. The default constructor allocates an empty vector:

```cpp
    StrBlob::StrBlob(): data(make_shared<vector<string>>()) { }
    StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) { }
```

The constructor that takes an initializer_list passes its parameter to the corresponding vector constructor (§ 2.2.1, p. 43). That constructor initializes the vector's elements by copying the values in the list.

**Element Access Members**

The pop_back, front, and back operations access members in the vector. These operations must check that an element exists before attempting to access that element. Because several members need to do the same checking, we've given
our class a private utility function named check that verifies that a given index is in range. In addition to an index, check takes a string argument that it will pass to the exception handler. The string describes what went wrong:

```cpp
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

The pop_back and element access members first call check. If check succeeds, these members forward their work to the underlying vector operation:

```cpp
string& StrBlob::front()
{
    // if the vector is empty, check will throw
    check(0, "front on empty StrBlob");
    return data->front();
}

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

The front and back members should be overloaded on const (§ 7.3.2, p. 276). Defining those versions is left as an exercise.

**Copying, Assigning, and Destroying StrBlobs**

Like our Sales_data class, StrBlob uses the default versions of the operations that copy, assign, and destroy objects of its type (§ 7.1.5, p. 267). By default, these operations copy, assign, and destroy the data members of the class. Our StrBlob has only one data member, which is a shared_ptr. Therefore, when we copy, assign, or destroy a StrBlob, its shared_ptr member will be copied, assigned, or destroyed.

As we’ve seen, copying a shared_ptr increments its reference count; assigning one shared_ptr to another increments the count of the right-hand operand and decrements the count in the left-hand operand; and destroying a shared_ptr decrements the count. If the count in a shared_ptr goes to zero, the object to which that shared_ptr points is automatically destroyed. Thus, the vector allocated by the StrBlob constructors will be automatically destroyed when the last StrBlob pointing to that vector is destroyed.
EXERCISES SECTION 12.1.1

Exercise 12.1: How many elements do b1 and b2 have at the end of this code?

```cpp
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
    b1 = b2;
    b2.push_back("about");
}
```

Exercise 12.2: Write your own version of the StrBlob class including the const versions of front and back.

Exercise 12.3: Does this class need const versions of push_back and pop_back? If so, add them. If not, why aren’t they needed?

Exercise 12.4: In our check function we didn’t check whether i was greater than zero. Why is it okay to omit that check?

Exercise 12.5: We did not make the constructor that takes an initializer_list explicit (§ 7.5.4, p. 296). Discuss the pros and cons of this design choice.

12.1.2 Managing Memory Directly

The language itself defines two operators that allocate and free dynamic memory. The new operator allocates memory, and delete frees memory allocated by new.

For reasons that will become clear as we describe how these operators work, using these operators to manage memory is considerably more error-prone than using a smart pointer. Moreover, classes that do manage their own memory—unlike those that use smart pointers—cannot rely on the default definitions for the members that copy, assign, and destroy class objects (§ 7.1.4, p. 264). As a result, programs that use smart pointers are likely to be easier to write and debug.

⚠️ WARNING ⚠️

Until you have read Chapter 13, your classes should allocate dynamic memory only if they use smart pointers to manage that memory.

Using new to Dynamically Allocate and Initialize Objects

Objects allocated on the free store are unnamed, so new offers no way to name the objects that it allocates. Instead, new returns a pointer to the object it allocates:

```cpp
int *pi = new int;  // pi points to a dynamically allocated, unnamed, uninitialized int
```

This new expression constructs an object of type int on the free store and returns a pointer to that object.

By default, dynamically allocated objects are default initialized (§ 2.2.1, p. 43), which means that objects of built-in or compound type have undefined value; objects of class type are initialized by their default constructor:
string *ps = new string; // initialized to empty string
int *pi = new int;       // pi points to an uninitialized int

We can initialize a dynamically allocated object using direct initialization (§ 3.2.1, p. 84). We can use traditional construction (using parentheses), and under the new standard, we can also use list initialization (with curly braces):

```cpp
int *pi = new int(1024);  // object to which pi points has value 1024
string *ps = new string(10, '9'); // *ps is "9999999999"
vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

We can also value initialize (§ 3.3.1, p. 98) a dynamically allocated object by following the type name with a pair of empty parentheses:

```cpp
string *ps1 = new string; // default initialized to the empty string
string *ps = new string();  // value initialized to the empty string
int *pi1 = new int; // default initialized; *pi1 is undefined
int *pi2 = new int();  // value initialized to 0; *pi2 is 0
```

For class types (such as `string`) that define their own constructors (§ 7.1.4, p. 262), requesting value initialization is of no consequence; regardless of form, the object is initialized by the default constructor. In the case of built-in types the difference is significant; a value-initialized object of built-in type has a well-defined value but a default-initialized object does not. Similarly, members of built-in type in classes that rely on the synthesized default constructor will also be uninitialized if those members are not initialized in the class body (§ 7.1.4, p. 263).

```
For the same reasons as we usually initialize variables, it is also a good idea to initialize dynamically allocated objects.
```

When we provide an initializer inside parentheses, we can use `auto` (§ 2.5.2, p. 68) to deduce the type of the object we want to allocate from that initializer. However, because the compiler uses the initializer’s type to deduce the type to allocate, we can use `auto` only with a single initializer inside parentheses:

```cpp
auto p1 = new auto(obj);  // p points to an object of the type of obj
auto p2 = new auto{a,b,c}; // error: must use parentheses for the initializer
```

The type of `p1` is a pointer to the auto-deduced type of `obj`. If `obj` is an `int`, then `p1` is `int*`; if `obj` is a `string`, then `p1` is a `string*`; and so on. The newly allocated object is initialized from the value of `obj`.

**Dynamically Allocated const Objects**

It is legal to use `new` to allocate `const` objects:

```cpp
// allocate and initialize a const int
const int *pci = new const int(1024);
// allocate a default-initialized const empty string
const string *pcs = new const string;
```
Like any other const, a dynamically allocated const object must be initialized. A const dynamic object of a class type that defines a default constructor (§ 7.1.4, p. 263) may be initialized implicitly. Objects of other types must be explicitly initialized. Because the allocated object is const, the pointer returned by new is a pointer to const (§ 2.4.2, p. 62).

**Memory Exhaustion**

Although modern machines tend to have huge memory capacity, it is always possible that the free store will be exhausted. Once a program has used all of its available memory, new expressions will fail. By default, if new is unable to allocate the requested storage, it throws an exception of type bad_alloc (§ 5.6, p. 193). We can prevent new from throwing an exception by using a different form of new:

```cpp
// if allocation fails, new returns a null pointer
int *p1 = new int; // if allocation fails, new throws std::bad_alloc
int *p2 = new (nothrow) int; // if allocation fails, new returns a null pointer
```

For reasons we’ll explain in § 19.1.2 (p. 824) this form of new is referred to as **placement new**. A placement new expression lets us pass additional arguments to new. In this case, we pass an object named nothrow that is defined by the library. When we pass nothrow to new, we tell new that it must not throw an exception. If this form of new is unable to allocate the requested storage, it will return a null pointer. Both bad_alloc and nothrow are defined in the new header.

**Freeing Dynamic Memory**

In order to prevent memory exhaustion, we must return dynamically allocated memory to the system once we are finished using it. We return memory through a **delete expression**. A delete expression takes a pointer to the object we want to free:

```cpp
delete p; // p must point to a dynamically allocated object or be null
```

Like new, a delete expression performs two actions: It destroys the object to which its given pointer points, and it frees the corresponding memory.

**Pointer Values and delete**

The pointer we pass to delete must either point to dynamically allocated memory or be a null pointer (§ 2.3.2, p. 53). Deleting a pointer to memory that was not allocated by new, or deleting the same pointer value more than once, is undefined:

```cpp
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i; // error: i is not a pointer
delete pi1; // undefined: pi1 refers to a local
delete pd; // ok
delete pd2; // undefined: the memory pointed to by pd2 was already freed
delete pi2; // ok: it is always ok to delete a null pointer
```
The compiler will generate an error for the `delete` of `i` because it knows that `i` is not a pointer. The errors associated with executing `delete` on `pi1` and `pd2` are more insidious: In general, compilers cannot tell whether a pointer points to a statically or dynamically allocated object. Similarly, the compiler cannot tell whether memory addressed by a pointer has already been freed. Most compilers will accept these `delete` expressions, even though they are in error.

Although the value of a `const` object cannot be modified, the object itself can be destroyed. As with any other dynamic object, a `const` dynamic object is freed by executing `delete` on a pointer that points to that object:

```cpp
const int *pci = new const int(1024);
delete pci; // ok: deletes a const object
```

### Dynamically Allocated Objects Exist until They Are Freed

As we saw in § 12.1.1 (p. 452), memory that is managed through a `shared_ptr` is automatically deleted when the last `shared_ptr` is destroyed. The same is not true for memory we manage using built-in pointers. A dynamic object managed through a built-in pointer exists until it is explicitly deleted.

Functions that return pointers (rather than smart pointers) to dynamic memory put a burden on their callers—the caller must remember to delete the memory:

```cpp
// factory returns a pointer to a dynamically allocated object
Foo* factory(T arg)
{
    // process arg as appropriate
    return new Foo(arg); // caller is responsible for deleting this memory
}
```

Like our earlier `factory` function (§ 12.1.1, p. 453), this version of `factory` allocates an object but does not delete it. Callers of `factory` are responsible for freeing this memory when they no longer need the allocated object. Unfortunately, all too often the caller forgets to do so:

```cpp
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p but do not delete it
} // p goes out of scope, but the memory to which p points is not freed!
```

Here, our `use_factory` function calls `factory`, which allocates a new object of type `Foo`. When `use_factory` returns, the local variable `p` is destroyed. That variable is a built-in pointer, not a smart pointer.

Unlike class types, nothing happens when objects of built-in type are destroyed. In particular, when a pointer goes out of scope, nothing happens to the object to which the pointer points. If that pointer points to dynamic memory, that memory is not automatically freed.

> **WARNING** Dynamic memory managed through built-in pointers (rather than smart pointers) exists until it is explicitly freed.
In this example, p was the only pointer to the memory allocated by factory. Once use_factory returns, the program has no way to free that memory. Depending on the logic of our overall program, we should fix this bug by remembering to free the memory inside use_factory:

```c
void use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    delete p;  // remember to free the memory now that we no longer need it
}
```
or, if other code in our system needs to use the object allocated by use_factory, we should change that function to return a pointer to the memory it allocated:

```c
Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // use p
    return p;  // caller must delete the memory
}
```

**CAUTION: MANAGING DYNAMIC MEMORY IS ERROR-PRONE**

There are three common problems with using new and delete to manage dynamic memory:

1. Forgetting to delete memory. Neglecting to delete dynamic memory is known as a “memory leak,” because the memory is never returned to the free store. Testing for memory leaks is difficult because they usually cannot be detected until the application is run for a long enough time to actually exhaust memory.

2. Using an object after it has been deleted. This error can sometimes be detected by making the pointer null after the delete.

3. Deleting the same memory twice. This error can happen when two pointers address the same dynamically allocated object. If delete is applied to one of the pointers, then the object’s memory is returned to the free store. If we subsequently delete the second pointer, then the free store may be corrupted.

These kinds of errors are considerably easier to make than they are to find and fix.

You can avoid all of these problems by using smart pointers exclusively. The smart pointer will take care of deleting the memory only when there are no remaining smart pointers pointing to that memory.

**Resetting the Value of a Pointer after a delete...**

When we delete a pointer, that pointer becomes invalid. Although the pointer is invalid, on many machines the pointer continues to hold the address of the (freed) dynamic memory. After the delete, the pointer becomes what is referred to as a
dangling pointer. A dangling pointer is one that refers to memory that once held an object but no longer does so.

Dangling pointers have all the problems of uninitialized pointers (§ 2.3.2, p. 54). We can avoid the problems with dangling pointers by deleting the memory associated with a pointer just before the pointer itself goes out of scope. That way there is no chance to use the pointer after the memory associated with the pointer is freed. If we need to keep the pointer around, we can assign `nullptr` to the pointer after we use `delete`. Doing so makes it clear that the pointer points to no object.

...Provides Only Limited Protection

A fundamental problem with dynamic memory is that there can be several pointers that point to the same memory. Resetting the pointer we use to `delete` that memory lets us check that particular pointer but has no effect on any of the other pointers that still point at the (freed) memory. For example:

```c++
int *p(new int(42)); // p points to dynamic memory
auto q = p;           // p and q point to the same memory
delete p;            // invalidates both p and q
p = nullptr;          // indicates that p is no longer bound to an object
```

Here both `p` and `q` point at the same dynamically allocated object. We `delete` that memory and set `p` to `nullptr`, indicating that the pointer no longer points to an object. However, resetting `p` has no effect on `q`, which became invalid when we deleted the memory to which `p` (and `q`!) pointed. In real systems, finding all the pointers that point to the same memory is surprisingly difficult.

### Exercises Section 12.1.2

**Exercise 12.6:** Write a function that returns a dynamically allocated vector of ints. Pass that vector to another function that reads the standard input to give values to the elements. Pass the vector to another function to print the values that were read. Remember to `delete` the vector at the appropriate time.

**Exercise 12.7:** Redo the previous exercise, this time using `shared_ptr`.

**Exercise 12.8:** Explain what if anything is wrong with the following function.

```c++
bool b() {
    int* p = new int;
    // ...
    return p;
}
```

**Exercise 12.9:** Explain what happens in the following code:

```c++
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```
12.1.3 Using `shared_ptr`s with `new`  

As we’ve seen, if we do not initialize a smart pointer, it is initialized as a null pointer. As described in Table 12.3, we can also initialize a smart pointer from a pointer returned by `new`:

```cpp
shared_ptr<double> p1; // shared_ptr that can point at a double
shared_ptr<int> p2(new int(42)); // p2 points to an int with value 42
```

The smart pointer constructors that take pointers are explicit (§ 7.5.4, p. 296). Hence, we cannot implicitly convert a built-in pointer to a smart pointer; we must use the direct form of initialization (§ 3.2.1, p. 84) to initialize a smart pointer:

```cpp
shared_ptr<int> p1 = new int(1024); // error: must use direct initialization
shared_ptr<int> p2(new int(1024)); // ok: uses direct initialization
```

The initialization of `p1` implicitly asks the compiler to create a `shared_ptr` from the `int*` returned by `new`. Because we can’t implicitly convert a pointer to a smart pointer, this initialization is an error. For the same reason, a function that returns a `shared_ptr` cannot implicitly convert a plain pointer in its return statement:

```cpp
shared_ptr<int> clone(int p) {
    return new int(p); // error: implicit conversion to shared_ptr<int>
}
```

We must explicitly bind a `shared_ptr` to the pointer we want to return:

```cpp
shared_ptr<int> clone(int p) {
    // ok: explicitly create a shared_ptr<int> from int*
    return shared_ptr<int>(new int(p));
}
```

By default, a pointer used to initialize a smart pointer must point to dynamic memory because, by default, smart pointers use `delete` to free the associated object. We can bind smart pointers to pointers to other kinds of resources. However, to do so, we must supply our own operation to use in place of `delete`. We’ll see how to supply our own deletion code in § 12.1.4 (p. 468).

Don’t Mix Ordinary Pointers and Smart Pointers …

A `shared_ptr` can coordinate destruction only with other `shared_ptr`s that are copies of itself. Indeed, this fact is one of the reasons we recommend using `make_shared` rather than `new`. That way, we bind a `shared_ptr` to the object at the same time that we allocate it. There is no way to inadvertently bind the same memory to more than one independently created `shared_ptr`.

Consider the following function that operates on a `shared_ptr`:

```cpp
// ptr is created and initialized when process is called
void process(shared_ptr<int> ptr)
{
    // use ptr
} // ptr goes out of scope and is destroyed
```
Table 12.3: Other Ways to Define and Change shared_ptrs

<table>
<thead>
<tr>
<th>Shared Pointer Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shared_ptr&lt;T&gt; p(q)</code></td>
<td><code>p</code> manages the object to which the built-in pointer <code>q</code> points; <code>q</code> must point to memory allocated by <code>new</code> and must be convertible to <code>T*</code>.</td>
</tr>
<tr>
<td><code>shared_ptr&lt;T&gt; p(u)</code></td>
<td><code>p</code> assumes ownership from the <code>unique_ptr u</code>; makes <code>u</code> null.</td>
</tr>
<tr>
<td><code>shared_ptr&lt;T&gt; p(q, d)</code></td>
<td><code>p</code> assumes ownership for the object to which the built-in pointer <code>q</code> points. <code>q</code> must be convertible to <code>T*</code> (§ 4.11.2, p. 161). <code>p</code> will use the callable object <code>d</code> (§ 10.3.2, p. 388) in place of <code>delete</code> to free <code>q</code>.</td>
</tr>
<tr>
<td><code>p.reset()</code></td>
<td>If <code>p</code> is the only <code>shared_ptr</code> pointing at its object, <code>reset</code> frees <code>p</code>'s existing object. If the optional built-in pointer <code>q</code> is passed, makes <code>p</code> point to <code>q</code>, otherwise makes <code>p</code> null. If <code>d</code> is supplied, will call <code>d</code> to free <code>q</code> otherwise uses <code>delete</code> to free <code>q</code>.</td>
</tr>
<tr>
<td><code>p.reset(p2, d)</code></td>
<td><code>p</code> is a copy of the <code>shared_ptr p2</code> as described in Table 12.2 except that <code>p</code> uses the callable object <code>d</code> in place of <code>delete</code>.</td>
</tr>
</tbody>
</table>

The parameter to `process` is passed by value, so the argument to `process` is copied into `ptr`. Copying a `shared_ptr` increments its reference count. Thus, inside `process` the count is at least 2. When `process` completes, the reference count of `ptr` is decremented but cannot go to zero. Therefore, when the local variable `ptr` is destroyed, the memory to which `ptr` points will not be deleted.

The right way to use this function is to pass it a `shared_ptr`:

```cpp
shared_ptr<int> p(new int(42)); // reference count is 1
process(p); // copying p increments its count; in process the reference count is 2
int i = *p; // ok: reference count is 1
```

Although we cannot pass a built-in pointer to `process`, we can pass `process` a (temporary) `shared_ptr` that we explicitly construct from a built-in pointer. However, doing so is likely to be an error:

```cpp
int *x(new int(1024)); // dangerous: x is a plain pointer, not a smart pointer
process(x); // error: cannot convert int* to shared_ptr<int>
process(shared_ptr<int>(x)); // legal, but the memory will be deleted!
int j = *x; // undefined: x is a dangling pointer!
```

In this call, we passed a temporary `shared_ptr` to `process`. That temporary is destroyed when the expression in which the call appears finishes. Destroying the temporary decrements the reference count, which goes to zero. The memory to which the temporary points is freed when the temporary is destroyed.

But `x` continues to point to that (freed) memory; `x` is now a dangling pointer. Attempting to use the value of `x` is undefined.

When we bind a `shared_ptr` to a plain pointer, we give responsibility for that memory to that `shared_ptr`. Once we give `shared_ptr` responsibility for a pointer, we should no longer use a built-in pointer to access the memory to which the `shared_ptr` now points.
It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.

...and Don’t Use get to Initialize or Assign Another Smart Pointer

The smart pointer types define a function named get (described in Table 12.1 (p. 452)) that returns a built-in pointer to the object that the smart pointer is managing. This function is intended for cases when we need to pass a built-in pointer to code that can’t use a smart pointer. The code that uses the return from get must not delete that pointer.

Although the compiler will not complain, it is an error to bind another smart pointer to the pointer returned by get:

```cpp
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get(); // ok: but don’t use q in any way that might delete its pointer
{ // new block
  // undefined: two independent shared_ptrs point to the same memory
  shared_ptr<int>(q);
} // block ends, q is destroyed, and the memory to which q points is freed
int foo = *p; // undefined; the memory to which p points was freed
```

In this case, both p and q point to the same memory. Because they were created independently from each other, each has a reference count of 1. When the block in which q was defined ends, q is destroyed. Destroying q frees the memory to which q points. That makes p into a dangling pointer, meaning that what happens when we attempt to use p is undefined. Moreover, when p is destroyed, the pointer to that memory will be deleted a second time.

Use get only to pass access to the pointer to code that you know will not delete the pointer. In particular, never use get to initialize or assign to another smart pointer.

Other shared_ptr Operations

The shared_ptr class gives us a few other operations, which are listed in Table 12.2 (p. 453) and Table 12.3 (on the previous page). We can use reset to assign a new pointer to a shared_ptr:

```cpp
p = new int(1024); // error: cannot assign a pointer to a shared_ptr
p.reset(new int(1024)); // ok: p points to a new object
```

Like assignment, reset updates the reference counts and, if appropriate, deletes the object to which p points. The reset member is often used together with unique to control changes to the object shared among several shared_ptrs. Before changing the underlying object, we check whether we’re the only user. If not, we make a new copy before making the change:

```cpp
if (!p.unique())
  p.reset(new string(*p)); // we aren’t alone; allocate a new copy
  *p += newVal; // now that we know we’re the only pointer, okay to change this object
```
EXERCISES SECTION 12.1.3

Exercise 12.10: Explain whether the following call to the `process` function defined on page 464 is correct. If not, how would you correct the call?

```cpp
shared_ptr<int> p(new int(42));
process(shared_ptr<int>(p));
```

Exercise 12.11: What would happen if we called `process` as follows?

```cpp
process(shared_ptr<int>(p.get()));
```

Exercise 12.12: Using the declarations of `p` and `sp` explain each of the following calls to `process`. If the call is legal, explain what it does. If the call is illegal, explain why:

```cpp
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

Exercise 12.13: What happens if we execute the following code?

```cpp
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 Smart Pointers and Exceptions

In § 5.6.2 (p. 196) we noted that programs that use exception handling to continue processing after an exception occurs need to ensure that resources are properly freed if an exception occurs. One easy way to make sure resources are freed is to use smart pointers.

When we use a smart pointer, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely:

```cpp
void f()
{
    shared_ptr<int> sp(new int(42)); // allocate a new object
    // code that throws an exception that is not caught inside f
} // shared_ptr freed automatically when the function ends
```

When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed. In this case, `sp` is a `shared_ptr`, so destroying `sp` checks its reference count. Here, `sp` is the only pointer to the memory it manages; that memory will be freed as part of destroying `sp`.

In contrast, memory that we manage directly is not automatically freed when an exception occurs. If we use built-in pointers to manage memory and an exception occurs after a `new` but before the corresponding `delete`, then that memory won’t be freed:
void f()
{
    int *ip = new int(42);    // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;               // free the memory before exiting
}

If an exception happens between the new and the delete, and is not caught inside f, then this memory can never be freed. There is no pointer to this memory outside the function f. Thus, there is no way to free this memory.

Smart Pointers and Dumb Classes

Many C++ classes, including all the library classes, define destructors (§ 12.1.1, p. 452) that take care of cleaning up the resources used by that object. However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.

Classes that allocate resources—and that do not define destructors to free those resources—can be subject to the same kind of errors that arise when we use dynamic memory. It is easy to forget to release the resource. Similarly, if an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

We can often use the same kinds of techniques we use to manage dynamic memory to manage classes that do not have well-behaved destructors. For example, imagine we’re using a network library that is used by both C and C++. Programs that use this library might contain code such as

```cpp
struct destination;    // represents what we are connecting to
struct connection;     // information needed to use the connection
connection connect(destination*); // open the connection
void disconnect(connection);   // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done
    connection c = connect(&d);
    // use the connection
    // if we forget to call disconnect before exiting f, there will be no way to close c
}
```

If connection had a destructor, that destructor would automatically close the connection when f completes. However, connection does not have a destructor. This problem is nearly identical to our previous program that used a shared_ptr to avoid memory leaks. It turns out that we can also use a shared_ptr to ensure that the connection is properly closed.

Using Our Own Deletion Code

By default, shared_ptrs assume that they point to dynamic memory. Hence, by default, when a shared_ptr is destroyed, it executes delete on the pointer it
Section 12.1 Dynamic Memory and Smart Pointers

holds. To use a `shared_ptr` to manage a connection, we must first define a function to use in place of `delete`. It must be possible to call this `deleter` function with the pointer stored inside the `shared_ptr`. In this case, our deleter must take a single argument of type `connection*`:

```cpp
void end_connection(connection *p) { disconnect(*p); }
```

When we create a `shared_ptr`, we can pass an optional argument that points to a deleter function (§ 6.7, p. 247):

```cpp
void f(destination &d /* other parameters */) {
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

When `p` is destroyed, it won’t execute `delete` on its stored pointer. Instead, `p` will call `end_connection` on that pointer. In turn, `end_connection` will call `disconnect`, thus ensuring that the connection is closed. If `f` exits normally, then `p` will be destroyed as part of the return. Moreover, `p` will also be destroyed, and the connection will be closed, if an exception occurs.

---

**CAUTION: SMART POINTER PITFALLS**

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- Don’t use the same built-in pointer value to initialize (or `reset`) more than one smart pointer.
- Don’t `delete` the pointer returned from `get()`.
- Don’t use `get()` to initialize or `reset` another smart pointer.
- If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a deleter (§ 12.1.4, p. 468, and § 12.1.5, p. 471).

---

**EXERCISES SECTION 12.1.4**

**Exercise 12.14:** Write your own version of a function that uses a `shared_ptr` to manage a connection.

**Exercise 12.15:** Rewrite the first exercise to use a lambda (§ 10.3.2, p. 388) in place of the `end_connection` function.
12.1.5 unique_ptr

A unique_ptr “owns” the object to which it points. Unlike shared_ptr, only one unique_ptr at a time can point to a given object. The object to which a unique_ptr points is destroyed when the unique_ptr is destroyed. Table 12.4 lists the operations specific to unique_ptrs. The operations common to both were covered in Table 12.1 (p. 452).

Unlike shared_ptr, there is no library function comparable to make_shared that returns a unique_ptr. Instead, when we define a unique_ptr, we bind it to a pointer returned by new. As with shared_ptrs, we must use the direct form of initialization:

```cpp
define unique_ptr<T> p1; // unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); // p2 points to int with value 42
```

Because a unique_ptr owns the object to which it points, unique_ptr does not support ordinary copy or assignment:

```cpp
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1); // error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2; // error: no assign for unique_ptr
```

### Table 12.4: unique_ptr Operations (See Also Table 12.1 (p. 452))

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique_ptr&lt;T&gt; u1</td>
<td>Null unique_ptrs that can point to objects of type T. u1 will use delete to free its pointer; u2 will use a callable object of type D to free its pointer.</td>
</tr>
<tr>
<td>unique_ptr&lt;T, D&gt; u2</td>
<td>Null unique_ptr that point to objects of type T that uses d, which must be an object of type D in place of delete.</td>
</tr>
<tr>
<td>u = nullptr</td>
<td>Deletes the object to which u points; makes u null.</td>
</tr>
<tr>
<td>u.release()</td>
<td>Relinquishes control of the pointer u had held; returns the pointer u had held and makes u null.</td>
</tr>
<tr>
<td>u.reset()</td>
<td>Deletes the object to which u points;</td>
</tr>
<tr>
<td>u.reset(q)</td>
<td>If the built-in pointer q is supplied, makes u point to that object.</td>
</tr>
<tr>
<td>u.reset(nullptr)</td>
<td>Otherwise makes u null.</td>
</tr>
</tbody>
</table>

Although we can’t copy or assign a unique_ptr, we can transfer ownership from one (nonconst) unique_ptr to another by calling release or reset:

```cpp
// transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes the memory to which p2 had pointed
```

The release member returns the pointer currently stored in the unique_ptr and makes that unique_ptr null. Thus, p2 is initialized from the pointer value that had been stored in p1 and p1 becomes null.
The `reset` member takes an optional pointer and repositions the `unique_ptr` to point to the given pointer. If the `unique_ptr` is not null, then the object to which the `unique_ptr` had pointed is deleted. The call to `reset` on `p2`, therefore, frees the memory used by the string initialized from "Stegosaurus", transfers `p3`'s pointer to `p2`, and makes `p3` null.

Calling `release` breaks the connection between a `unique_ptr` and the object it had been managing. Often the pointer returned by `release` is used to initialize or assign another smart pointer. In that case, responsibility for managing the memory is simply transferred from one smart pointer to another. However, if we do not use another smart pointer to hold the pointer returned from `release`, our program takes over responsibility for freeing that resource:

```c++
p2.release();  // WRONG: p2 won't free the memory and we've lost the pointer
auto p = p2.release();  // ok, but we must remember to delete(p)
```

### Passing and Returning `unique_ptr`

There is one exception to the rule that we cannot copy a `unique_ptr`: We can copy or assign a `unique_ptr` that is about to be destroyed. The most common example is when we return a `unique_ptr` from a function:

```c++
unique_ptr<int> clone(int p) {
    // ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
```

Alternatively, we can also return a copy of a local object:

```c++
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int(p));
    // ...
    return ret;
}
```

In both cases, the compiler knows that the object being returned is about to be destroyed. In such cases, the compiler does a special kind of “copy” which we’ll discuss in § 13.6.2 (p. 534).

---

**Backward Compatibility: `auto_ptr`**

Earlier versions of the library included a class named `auto_ptr` that had some, but not all, of the properties of `unique_ptr`. In particular, it was not possible to store an `auto_ptr` in a container, nor could we return one from a function.

Although `auto_ptr` is still part of the standard library, programs should use `unique_ptr` instead.

---

**Passing a Deleter to `unique_ptr`**

Like `shared_ptr`, by default, `unique_ptr` uses `delete` to free the object to which a `unique_ptr` points. As with `shared_ptr`, we can override the default
deleter in a `unique_ptr` (§ 12.1.4, p. 468). However, for reasons we’ll describe in § 16.1.6 (p. 676), the way `unique_ptr` manages its deleter is differs from the way `shared_ptr` does.

Overriding the deleter in a `unique_ptr` affects the `unique_ptr` type as well as how we construct (or reset) objects of that type. Similar to overriding the comparison operation of an associative container (§ 11.2.2, p. 425), we must supply the deleter type inside the angle brackets along with the type to which the `unique_ptr` can point. We supply a callable object of the specified type when we create or reset an object of this type:

```cpp
unique_ptr<objT, delT> p (new objT, fcn);
```

As a somewhat more concrete example, we’ll rewrite our connection program to use a `unique_ptr` in place of a `shared_ptr` as follows:

```cpp
void f(destination &d /* other needed parameters */) {
    connection c = connect(&d); // open the connection
    // when p is destroyed, the connection will be closed
    unique_ptr<connection, decltype(end_connection)*>(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

Here we use `decltype` (§ 2.5.3, p. 70) to specify the function pointer type. Because `decltype(end_connection)` returns a function type, we must remember to add a `*` to indicate that we’re using a pointer to that type (§ 6.7, p. 250).

### Exercises Section 12.1.5

**Exercise 12.16:** Compilers don’t always give easy-to-understand error messages if we attempt to copy or assign a `unique_ptr`. Write a program that contains these errors to see how your compiler diagnoses them.

**Exercise 12.17:** Which of the following `unique_ptr` declarations are illegal or likely to result in subsequent program error? Explain what the problem is with each one.

```cpp
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix); (b) IntP p1(pi);
(c) IntP p2(pi2); (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());
```

**Exercise 12.18:** Why doesn’t `shared_ptr` have a release member?
12.1.6 weak_ptr

A weak_ptr (Table 12.5) is a smart pointer that does not control the lifetime of the object to which it points. Instead, a weak_ptr points to an object that is managed by a shared_ptr. Binding a weak_ptr to a shared_ptr does not change the reference count of that shared_ptr. Once the last shared_ptr pointing to the object goes away, the object itself will be deleted. That object will be deleted even if there are weak_ptrs pointing to it—hence the name weak_ptr, which captures the idea that a weak_ptr shares its object “weakly.”

When we create a weak_ptr, we initialize it from a shared_ptr:

```cpp
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp weakly shares with p; use count in p is unchanged
```

Here both wp and p point to the same object. Because the sharing is weak, creating wp doesn’t change the reference count of p; it is possible that the object to which wp points might be deleted.

Because the object might no longer exist, we cannot use a weak_ptr to access its object directly. To access that object, we must call lock. The lock function checks whether the object to which the weak_ptr points still exists. If so, lock returns a shared_ptr to the shared object. As with any other shared_ptr, we are guaranteed that the underlying object to which that shared_ptr points continues to exist at least as long as that shared_ptr exists. For example:

```cpp
if (shared_ptr<int> np = wp.lock()) { // true if np is not null
    // inside the if, np shares its object with p
}
```

Here we enter the body of the if only if the call to lock succeeds. Inside the if, it is safe to use np to access that object.

| weak_ptr<T> w | Null weak_ptr that can point at objects of type T. |
| weak_ptr<T> w(sp) | weak_ptr that points to the same object as the shared_ptr sp. T must be convertible to the type to which sp points. |
| w = p | p can be a shared_ptr or a weak_ptr. After the assignment w shares ownership with p. |
| w.reset() | Makes w null. |
| w.use_count() | The number of shared_ptrs that share ownership with w. |
| w.expired() | Returns true if w.use_count() is zero, false otherwise. |
| w.lock() | If expired is true, returns a null shared_ptr; otherwise returns a shared_ptr to the object to which w points. |

### Checked Pointer Class

As an illustration of when a weak_ptr is useful, we’ll define a companion pointer class for our StrBlob class. Our pointer class, which we’ll name StrBlobPtr,
will store a weak_ptr to the data member of the StrBlob from which it was initialized. By using a weak_ptr, we don’t affect the lifetime of the vector to which a given StrBlob points. However, we can prevent the user from attempting to access a vector that no longer exists.

StrBlobPtr will have two data members: wptr, which is either null or points to a vector in a StrBlob; and curr, which is the index of the element that this object currently denotes. Like its companion StrBlob class, our pointer class has a check member to verify that it is safe to dereference the StrBlobPtr:

```cpp
// StrBlobPtr throws an exception on attempts to access a nonexistent element
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // prefix version
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // store a weak_ptr, which means the underlying vector might be destroyed
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // current position within the array
};
```

The default constructor generates a null StrBlobPtr. Its constructor initializer list (§ 7.1.4, p. 265) explicitly initializes curr to zero and implicitly initializes wptr as a null weak_ptr. The second constructor takes a reference to StrBlob and an optional index value. This constructor initializes wptr to point to the vector in the shared_ptr of the given StrBlob object and initializes curr to the value of sz. We use a default argument (§ 6.5.1, p. 236) to initialize curr to denote the first element by default. As we’ll see, the sz parameter will be used by the end member of StrBlob.

It is worth noting that we cannot bind a StrBlobPtr to a const StrBlob object. This restriction follows from the fact that the constructor takes a reference to a nonconst object of type StrBlob.

The check member of StrBlobPtr differs from the one in StrBlob because it must check whether the vector to which it points is still around:

```cpp
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // is the vector still around?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // otherwise, return a shared_ptr to the vector
}
```
Because a weak_ptr does not participate in the reference count of its corresponding shared_ptr, the vector to which this StrBlobPtr points might have been deleted. If the vector is gone, lock will return a null pointer. In this case, any reference to the vector will fail, so we throw an exception. Otherwise, check verifies its given index. If that value is okay, check returns the shared_ptr it obtained from lock.

**Pointer Operations**

We’ll learn how to define our own operators in Chapter 14. For now, we’ve defined functions named deref and incr to dereference and increment the StrBlobPtr, respectively. The deref member calls check to verify that it is safe to use the vector and that curr is in range:

```cpp
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) is the vector to which this object points
}
```

If check succeeds, p is a shared_ptr to the vector to which this StrBlobPtr points. The expression (*p)[curr] dereferences that shared_ptr to get the vector and uses the subscript operator to fetch and return the element at curr.

The incr member also calls check:

```cpp
// prefix: return a reference to the incremented object
StrBlobPtr& StrBlobPtr::incr()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // advance the current state
    return *this;
}
```

We’ll also give our StrBlob class begin and end operations. These members will return StrBlobPtrs pointing to the first or one past the last element in the StrBlob itself. In addition, because StrBlobPtr accesses the data member of StrBlob, we must also make StrBlobPtr a friend of StrBlob (§ 7.3.4, p. 279):

```cpp
class StrBlob {
    friend class StrBlobPtr;
    // other members as in § 12.1.1 (p. 456)
    StrBlobPtr begin(); // return StrBlobPtr to the first element
    StrBlobPtr end(); // and one past the last element
};
// these members can’t be defined until StrStrBlob and StrStrBlobPtr are defined
StrBlobPtr StrBlob::begin() { return StrBlobPtr(*this); }
StrBlobPtr StrBlob::end()
{
    return StrBlobPtr(*this, data->size());
}
```
12.2 Dynamic Arrays

The new and delete operators allocate objects one at a time. Some applications, need the ability to allocate storage for many objects at once. For example, vectors and strings store their elements in contiguous memory and must allocate several elements at once whenever the container has to be reallocated (§ 9.4, p. 355).

To support such usage, the language and library provide two ways to allocate an array of objects at once. The language defines a second kind of new expression that allocates and initializes an array of objects. The library includes a template class named allocator that lets us separate allocation from initialization. For reasons we’ll explain in § 12.2.2 (p. 481), using an allocator generally provides better performance and more flexible memory management.

Many, perhaps even most, applications have no direct need for dynamic arrays. When an application needs a varying number of objects, it is almost always easier, faster, and safer to do as we did with StrBlob: use a vector (or other library container). For reasons we’ll explain in § 13.6 (p. 531), the advantages of using a library container are even more pronounced under the new standard. Libraries that support the new standard tend to be dramatically faster than previous releases.

Most applications should use a library container rather than dynamically allocated arrays. Using a container is easier, less likely to contain memory-management bugs, and is likely to give better performance.

As we’ve seen, classes that use the containers can use the default versions of the operations for copy, assignment, and destruction (§ 7.1.5, p. 267). Classes that allocate dynamic arrays must define their own versions of these operations to manage the associated memory when objects are copied, assigned, and destroyed.

Do not allocate dynamic arrays in code inside classes until you have read Chapter 13.
12.2.1 new and Arrays

We ask new to allocate an array of objects by specifying the number of objects to allocate in a pair of square brackets after a type name. In this case, new allocates the requested number of objects and (assuming the allocation succeeds) returns a pointer to the first one:

```cpp
int *pia = new int[get_size()]; // pia points to the first of these ints
```

The size inside the brackets must have integral type but need not be a constant.

We can also allocate an array by using a type alias (§ 2.5.1, p. 67) to represent an array type. In this case, we omit the brackets:

```cpp
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT;    // allocates an array of 42 ints; p points to the first one
```

Here, new allocates an array of ints and returns a pointer to the first one. Even though there are no brackets in our code, the compiler executes this expression using new[]. That is, the compiler executes this expression as if we had written

```cpp
int *p = new int[42];
```

Allocating an Array Yields a Pointer to the Element Type

Although it is common to refer to memory allocated by new T[] as a “dynamic array,” this usage is somewhat misleading. When we use new to allocate an array, we do not get an object with an array type. Instead, we get a pointer to the element type of the array. Even if we use a type alias to define an array type, new does not allocate an object of array type. In this case, the fact that we’re allocating an array is not even visible; there is no [num]. Even so, new returns a pointer to the element type.

Because the allocated memory does not have an array type, we cannot call begin or end (§ 3.5.3, p. 118) on a dynamic array. These functions use the array dimension (which is part of an array’s type) to return pointers to the first and one past the last elements, respectively. For the same reasons, we also cannot use a range for to process the elements in a (so-called) dynamic array.

```
WARNING
It is important to remember that what we call a dynamic array does not have an array type.
```

Initializing an Array of Dynamically Allocated Objects

By default, objects allocated by new—whether allocated as a single object or in an array—are default initialized. We can value initialize (§ 3.3.1, p. 98) the elements in an array by following the size with an empty pair of parentheses.

```cpp
int *pia = new int[10];    // block of ten uninitialized ints
int *pia2 = new int[10](); // block of ten ints value initialized to 0
string *psa = new string[10]; // block of ten empty strings
string *psa2 = new string[10](); // block of ten empty strings
```
Under the new standard, we can also provide a braced list of element initializers:

```cpp
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// block of ten strings; the first four are initialized from the given initializers
// remaining elements are value initialized
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

As when we list initialize an object of built-in array type (§ 3.5.1, p. 114), the initializers are used to initialize the first elements in the array. If there are fewer initializers than elements, the remaining elements are value initialized. If there are more initializers than the given size, then the `new` expression fails and no storage is allocated. In this case, `new` throws an exception of type `bad_array_new_length`. Like `bad_alloc`, this type is defined in the `new` header.

Although we can use empty parentheses to value initialize the elements of an array, we cannot supply an element initializer inside the parentheses. The fact that we cannot supply an initial value inside the parentheses means that we cannot use `auto` to allocate an array (§ 12.1.2, p. 459).

### It Is Legal to Dynamically Allocate an Empty Array

We can use an arbitrary expression to determine the number of objects to allocate:

```cpp
size_t n = get_size(); // get_size returns the number of elements needed
int* p = new int[n]; // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
    /* process the array */;
```

An interesting question arises: What happens if `get_size` returns 0? The answer is that our code works fine. Calling `new[n]` with `n` equal to 0 is legal even though we cannot create an array variable of size 0:

```cpp
char arr[0]; // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of size zero, `new` returns a valid, nonzero pointer. That pointer is guaranteed to be distinct from any other pointer returned by `new`. This pointer acts as the off-the-end pointer (§ 3.5.3, p. 119) for a zero-element array. We can use this pointer in ways that we use an off-the-end iterator. The pointer can be compared as in the loop above. We can add zero to (or subtract zero from) such a pointer and can subtract the pointer from itself, yielding zero. The pointer cannot be dereferenced—after all, it points to no element.

In our hypothetical loop, if `get_size` returns 0, then `n` is also 0. The call to `new` will allocate zero objects. The condition in the `for` will fail (`p` is equal to `q + n` because `n` is 0). Thus, the loop body is not executed.

### Freeing Dynamic Arrays

To free a dynamic array, we use a special form of `delete` that includes an empty pair of square brackets:
Section 12.2 Dynamic Arrays

delete p;  // p must point to a dynamically allocated object or be null
delete [] pa;  // pa must point to a dynamically allocated array or be null

The second statement destroys the elements in the array to which pa points and frees the corresponding memory. Elements in an array are destroyed in reverse order. That is, the last element is destroyed first, then the second to last, and so on.

When we delete a pointer to an array, the empty bracket pair is essential: It indicates to the compiler that the pointer addresses the first element of an array of objects. If we omit the brackets when we delete a pointer to an array (or provide them when we delete a pointer to an object), the behavior is undefined.

Recall that when we use a type alias that defines an array type, we can allocate an array without using [] with new. Even so, we must use brackets when we delete a pointer to that array:

typedef int arrT[42];  // arrT names the type array of 42 ints
int *p = new arrT;  // allocates an array of 42 ints; p points to the first one
delete [] p;  // brackets are necessary because we allocated an array

Despite appearances, p points to the first element of an array of objects, not to a single object of type arrT. Thus, we must use [] when we delete p.

The compiler is unlikely to warn us if we forget the brackets when we delete a pointer to an array or if we use them when we delete a pointer to an object. Instead, our program is apt to misbehave without warning during execution.

Smart Pointers and Dynamic Arrays

The library provides a version of unique_ptr that can manage arrays allocated by new. To use a unique_ptr to manage a dynamic array, we must include a pair of empty brackets after the object type:

    // up points to an array of ten uninitialized ints
    unique_ptr<int[]> up(new int[10]);
    up.release();  // automatically uses delete[] to destroy its pointer

The brackets in the type specifier (<int[]>) say that up points not to an int but to an array of ints. Because up points to an array, when up destroys the pointer it manages, it will automatically use delete[].

unique_ptrs that point to arrays provide slightly different operations than those we used in § 12.1.5 (p. 470). These operations are described in Table 12.6 (overleaf). When a unique_ptr points to an array, we cannot use the dot and arrow member access operators. After all, the unique_ptr points to an array, not an object so these operators would be meaningless. On the other hand, when a unique_ptr points to an array, we can use the subscript operator to access the elements in the array:

    for (size_t i = 0; i != 10; ++i)
        up[i] = i;  // assign a new value to each of the elements
### Table 12.6: unique_ptr to Arrays

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique_ptr&lt;T[]&gt; u u</td>
<td>u can point to a dynamically allocated array of type T.</td>
</tr>
<tr>
<td>unique_ptr&lt;T[]&gt; u(p)</td>
<td>u points to the dynamically allocated array to which the built-in pointer p points. p must be convertible to T* (§ 4.11.2, p. 161).</td>
</tr>
<tr>
<td>u[i]</td>
<td>Returns the object at position i in the array that u owns. u must point to an array.</td>
</tr>
</tbody>
</table>

Unlike `unique_ptr`, `shared_ptr` provide no direct support for managing a dynamic array. If we want to use a `shared_ptr` to manage a dynamic array, we must provide our own deleter:

```cpp
// to use a shared_ptr we must supply a deleter
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // uses the lambda we supplied that uses delete[] to free the array
```

Here we pass a lambda (§ 10.3.2, p. 388) that uses `delete[]` as the deleter.

Had we neglected to supply a deleter, this code would be undefined. By default, `shared_ptr` uses `delete` to destroy the object to which it points. If that object is a dynamic array, using `delete` has the same kinds of problems that arise if we forget to use `[]` when we delete a pointer to a dynamic array (§ 12.2.1, p. 479).

The fact that `shared_ptr` does not directly support managing arrays affects how we access the elements in the array:

```cpp
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // use get to get a built-in pointer
```

There is no subscript operator for `shared_ptr`, and the smart pointer types do not support pointer arithmetic. As a result, to access the elements in the array, we must use `get` to obtain a built-in pointer, which we can then use in normal ways.

## Exercises Section 12.2.1

**Exercise 12.23:** Write a program to concatenate two string literals, putting the result in a dynamically allocated array of char. Write a program to concatenate two library strings that have the same value as the literals used in the first program.

**Exercise 12.24:** Write a program that reads a string from the standard input into a dynamically allocated character array. Describe how your program handles varying size inputs. Test your program by giving it a string of data that is longer than the array size you’ve allocated.

**Exercise 12.25:** Given the following new expression, how would you delete `pa`?

```cpp
int *pa = new int[10];
```
### 12.2.2 The allocator Class

An aspect of `new` that limits its flexibility is that `new` combines allocating memory with constructing object(s) in that memory. Similarly, `delete` combines destruction with deallocation. Combining initialization with allocation is usually what we want when we allocate a single object. In that case, we almost certainly know the value the object should have.

When we allocate a block of memory, we often plan to construct objects in that memory as needed. In this case, we’d like to decouple memory allocation from object construction. Decoupling construction from allocation means that we can allocate memory in large chunks and pay the overhead of constructing the objects only when we actually need to create them.

In general, coupling allocation and construction can be wasteful. For example:

```cpp
string *const p = new string[n]; // construct n empty strings
string s;
string *q = p; // q points to the first string
while (cin >> s && q != p + n)
    *q++ = s; // assign a new value to *q
const size_t size = q - p; // remember how many strings we read
// use the array
delete[] p; // p points to an array; must remember to use delete[]
```

This `new` expression allocates and initializes `n` strings. However, we might not need `n` strings; a smaller number might suffice. As a result, we may have created objects that are never used. Moreover, for those objects we do use, we immediately assign new values over the previously initialized strings. The elements that are used are written twice: first when the elements are default initialized, and subsequently when we assign to them.

More importantly, classes that do not have default constructors cannot be dynamically allocated as an array.

#### The allocator Class

The library `allocator` class, which is defined in the `memory` header, lets us separate allocation from construction. It provides type-aware allocation of raw, unconstructed, memory. Table 12.7 (overleaf) outlines the operations that `allocator` supports. In this section, we’ll describe the `allocator` operations. In § 13.5 (p. 524), we’ll see an example of how this class is typically used.

Like `vector`, `allocator` is a template (§ 3.3, p. 96). To define an `allocator` we must specify the type of objects that a particular `allocator` can allocate. When an `allocator` object allocates memory, it allocates memory that is appropriately sized and aligned to hold objects of the given type:

```cpp
allocator<string> alloc; // object that can allocate strings
auto const p = alloc.allocate(n); // allocate n unconstructed strings
```

This call to `allocate` allocates memory for `n` strings.
Table 12.7: Standard allocator Class and Customized Algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocator&lt;T&gt; a</td>
<td>Defines an allocator object named a that can allocate memory for objects of type T.</td>
</tr>
<tr>
<td>a.allocate(n)</td>
<td>Allocates raw, unconstructed memory to hold n objects of type T.</td>
</tr>
<tr>
<td>a.deallocate(p, n)</td>
<td>Deallocates memory that held n objects of type T starting at the address in the T* pointer p; p must be a pointer previously returned by allocate, and n must be the size requested when p was created. The user must run destroy on any objects that were constructed in this memory before calling deallocate.</td>
</tr>
<tr>
<td>a.construct(p, args)</td>
<td>p must be a pointer to type T that points to raw memory; args are passed to a constructor for type T, which is used to construct an object in the memory pointed to by p.</td>
</tr>
<tr>
<td>a.destroy(p)</td>
<td>Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the T* pointer p.</td>
</tr>
</tbody>
</table>

 allocators Allocate Unconstructed Memory

The memory an allocator allocates is unconstructed. We use this memory by constructing objects in that memory. In the new library the `construct` member takes a pointer and zero or more additional arguments; it constructs an element at the given location. The additional arguments are used to initialize the object being constructed. Like the arguments to `make_shared` (§ 12.1.1, p. 451), these additional arguments must be valid initializers for an object of the type being constructed. In particular, if the object is a class type, these arguments must match a constructor for that class:

```cpp
auto q = p; // q will point to one past the last constructed element
alloc.construct(q++); // *q is the empty string
alloc.construct(q++, 10, 'c'); // *q is cccccccccc
alloc.construct(q++, "hi"); // *q is hi!
```

In earlier versions of the library, `construct` took only two arguments: the pointer at which to construct an object and a value of the element type. As a result, we could only copy an element into unconstructed space, we could not use any other constructor for the element type. It is an error to use raw memory in which an object has not been constructed:

```cpp
cout << *p << endl; // ok: uses the string output operator
cout << *q << endl; // disaster: q points to unconstructed memory!
```

**WARNING** We must construct objects in order to use memory returned by allocate. Using unconstructed memory in other ways is undefined.

When we’re finished using the objects, we must destroy the elements we constructed, which we do by calling `destroy` on each constructed element. The `destroy` function takes a pointer and runs the destructor (§ 12.1.1, p. 452) on the pointed-to object:
while (q != p)
    alloc.destroy(--q);  // free the strings we actually allocated

At the beginning of our loop, q points one past the last constructed element. We decrement q before calling destroy. Thus, on the first call to destroy, q points to the last constructed element. We destroy the first element in the last iteration, after which q will equal p and the loop ends.

**WARNING**

We may destroy only elements that are actually constructed.

Once the elements have been destroyed, we can either reuse the memory to hold other strings or return the memory to the system. We free the memory by calling deallocate:

    alloc.deallocate(p, n);

The pointer we pass to deallocate cannot be null; it must point to memory allocated by allocate. Moreover, the size argument passed to deallocate must be the same size as used in the call to allocate that obtained the memory to which the pointer points.

**Algorithms to Copy and Fill Uninitialized Memory**

As a companion to the allocator class, the library also defines two algorithms that can construct objects in uninitialized memory. These functions, described in Table 12.8, are defined in the memory header.

<table>
<thead>
<tr>
<th>Table 12.8: allocator Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>These functions construct elements in the destination, rather than assigning to them.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uninitialized_copy(b, e, b2)</td>
<td>Copies elements from the input range denoted by iterators b and e into unconstructed, raw memory denoted by the iterator b2. The memory denoted by b2 must be large enough to hold a copy of the elements in the input range.</td>
</tr>
<tr>
<td>uninitialized_copy_n(b, n, b2)</td>
<td>Copies n elements starting from the one denoted by the iterator b into raw memory starting at b2.</td>
</tr>
<tr>
<td>uninitialized_fill(b, e, t)</td>
<td>Constructs objects in the range of raw memory denoted by iterators b and e as a copy of t.</td>
</tr>
<tr>
<td>uninitialized_fill_n(b, n, t)</td>
<td>Constructs an unsigned number n objects starting at b. b must denote unconstructed, raw memory large enough to hold the given number of objects.</td>
</tr>
</tbody>
</table>

As an example, assume we have a vector of ints that we want to copy into dynamic memory. We’ll allocate memory for twice as many ints as are in the vector. We’ll construct the first half of the newly allocated memory by copying elements from the original vector. We’ll construct elements in the second half by filling them with a given value:
484 Dynamic Memory

```c++
// allocate twice as many elements as vi holds
auto p = alloc.allocate(vi.size() * 2);
// construct elements starting at p as copies of elements in vi
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// initialize the remaining elements to 42
uninitialized_fill_n(q, vi.size(), 42);
```

Like the copy algorithm (§ 10.2.2, p. 382), uninitialized_copy takes three iterators. The first two denote an input sequence and the third denotes the destination into which those elements will be copied. The destination iterator passed to uninitialized_copy must denote unconstructed memory. Unlike copy, uninitialized_copy constructs elements in its destination.

Like copy, uninitialized_copy returns its (incremented) destination iterator. Thus, a call to uninitialized_copy returns a pointer positioned one element past the last constructed element. In this example, we store that pointer in q, which we pass to uninitialized_fill_n. This function, like fill_n (§ 10.2.2, p. 380), takes a pointer to a destination, a count, and a value. It will construct the given number of objects from the given value at locations starting at the given destination.

**EXERCISES SECTION 12.2.2**

**Exercise 12.26:** Rewrite the program on page 481 using an allocator.

### 12.3 Using the Library: A Text-Query Program

To conclude our discussion of the library, we’ll implement a simple text-query program. Our program will let a user search a given file for words that might occur in it. The result of a query will be the number of times the word occurs and a list of lines on which that word appears. If a word occurs more than once on the same line, we’ll display that line only once. Lines will be displayed in ascending order—that is, line 7 should be displayed before line 9, and so on.

For example, we might read the file that contains the input for this chapter and look for the word `element`. The first few lines of the output would be

```
element occurs 112 times
(line 36) A set element contains only a key;
(line 158) operator creates a new element
(line 160) Regardless of whether the element
(line 168) When we fetch an element from a map, we
(line 214) If the element is not found, find returns
```

followed by the remaining 100 or so lines in which the word `element` occurs.
12.3.1 Design of the Query Program

A good way to start the design of a program is to list the program’s operations. Knowing what operations we need can help us see what data structures we’ll need. Starting from requirements, the tasks our program must do include the following:

- When it reads the input, the program must remember the line(s) in which each word appears. Hence, the program will need to read the input a line at a time and break up the lines from the input file into its separate words
- When it generates output,
  - The program must be able to fetch the line numbers associated with a given word
  - The line numbers must appear in ascending order with no duplicates
  - The program must be able to print the text appearing in the input file at a given line number.

These requirements can be met quite neatly by using various library facilities:

- We’ll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. When we want to print a line, we can fetch the line using its line number as the index.
- We’ll use an `istringstream` (§ 8.3, p. 321) to break each line into words.
- We’ll use a `set` to hold the line numbers on which each word in the input appears. Using a `set` guarantees that each line will appear only once and that the line numbers will be stored in ascending order.
- We’ll use a `map` to associate each word with the `set` of line numbers on which the word appears. Using a `map` will let us fetch the `set` for any given word.

For reasons we’ll explain shortly, our solution will also use `shared_ptr`s.

Data Structures

Although we could write our program using `vector`, `set`, and `map` directly, it will be more useful if we define a more abstract solution. We’ll start by designing a class to hold the input file in a way that makes querying the file easy. This class, which we’ll name `TextQuery`, will hold a `vector` and a `map`. The `vector` will hold the text of the input file; the `map` will associate each word in that file to the `set` of line numbers on which that word appears. This class will have a constructor that reads a given input file and an operation to perform the queries.

The work of the query operation is pretty simple: It will look inside its `map` to see whether the given word is present. The hard part in designing this function is deciding what the query function should return. Once we know that a word was found, we need to know how often it occurred, the line numbers on which it occurred, and the corresponding text for each of those line numbers.

The easiest way to return all those data is to define a second class, which we’ll name `QueryResult`, to hold the results of a query. This class will have a `print` function to print the results in a `QueryResult`. 
Sharing Data between Classes

Our QueryResult class is intended to represent the results of a query. Those results include the set of line numbers associated with the given word and the corresponding lines of text from the input file. These data are stored in objects of type TextQuery.

Because the data that a QueryResult needs are stored in a TextQuery object, we have to decide how to access them. We could copy the set of line numbers, but that might be an expensive operation. Moreover, we certainly wouldn’t want to copy the vector, because that would entail copying the entire file in order to print (what will usually be) a small subset of the file.

We could avoid making copies by returning iterators (or pointers) into the TextQuery object. However, this approach opens up a pitfall: What happens if the TextQuery object is destroyed before a corresponding QueryResult? In that case, the QueryResult would refer to data in an object that no longer exists.

This last observation about synchronizing the lifetime of a QueryResult with the TextQuery object whose results it represents suggests a solution to our design problem. Given that these two classes conceptually “share” data, we’ll use shared_ptrs (§ 12.1.1, p. 450) to reflect that sharing in our data structures.

Using the TextQuery Class

When we design a class, it can be helpful to write programs using the class before actually implementing the members. That way, we can see whether the class has the operations we need. For example, the following program uses our proposed TextQuery and QueryResult classes. This function takes an ifstream that points to the file we want to process, and interacts with a user, printing the results for the given words:

```cpp
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile); // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
}
```

We start by initializing a TextQuery object named tq from a given ifstream. The TextQuery constructor reads that file into its vector and builds the map that associates the words in the input with the line numbers on which they appear.

The while loop iterates (indefinitely) with the user asking for a word to query and printing the related results. The loop condition tests the literal true (§ 2.1.3, p. 41), so it always succeeds. We exit the loop through the break (§ 5.5.1, p. 190)
after the first if. That if checks that the read succeeded. If so, it also checks whether the user entered a q to quit. Once we have a word to look for, we ask tq to find that word and then call print to print the results of the search.

### Exercises Section 12.3.1

**Exercise 12.27:** The TextQuery and QueryResult classes use only capabilities that we have already covered. Without looking ahead, write your own versions of these classes.

**Exercise 12.28:** Write a program to implement text queries without defining classes to manage the data. Your program should take a file and interact with a user to query for words in that file. Use vector, map, and set containers to hold the data for the file and to generate the results for the queries.

**Exercise 12.29:** We could have written the loop to manage the interaction with the user as a do while (§ 5.4.4, p. 189) loop. Rewrite the loop to use a do while. Explain which version you prefer and why.

### 12.3.2 Defining the Query Program Classes

We'll start by defining our TextQuery class. The user will create objects of this class by supplying an istream from which to read the input file. This class also provides the query operation that will take a string and return a QueryResult representing the lines on which that string appears.

The data members of the class have to take into account the intended sharing with QueryResult objects. The QueryResult class will share the vector representing the input file and the sets that hold the line numbers associated with each word in the input. Hence, our class has two data members: a shared_ptr to a dynamically allocated vector that holds the input file, and a map from string to shared_ptr<set>. The map associates each word in the file with a dynamically allocated set that holds the line numbers on which that word appears.

To make our code a bit easier to read, we'll also define a type member (§ 7.3.1, p. 271) to refer to line numbers, which are indices into a vector of strings:

```cpp
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // input file
    std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
};
```
The hardest part about this class is untangling the class names. As usual, for code that will go in a header file, we use `std::` when we use a library name (§ 3.1, p. 83). In this case, the repeated use of `std::` makes the code a bit hard to read at first. For example,

```cpp
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

is easier to understand when rewritten as

```cpp
map<string, shared_ptr<set<line_no>>> wm;
```

### The TextQuery Constructor

The TextQuery constructor takes an `ifstream`, which it reads a line at a time:

```cpp
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
  string text;
  while (getline(is, text)) { // for each line in the file
    file->push_back(text); // remember this line of text
    int n = file->size() - 1; // the current line number
    istringstream line(text); // separate the line into words
    string word;
    while (line >> word) { // for each word in that line
      // if word isn't already in wm, subscripting adds a new entry
      auto &lines = wm[word]; // lines is a shared_ptr
      if (!lines) // that pointer is null the first time we see word
        lines.reset(new set<line_no>); // allocate a new set
      lines->insert(n); // insert this line number
    }
  }
}
```

The constructor initializer allocates a new `vector` to hold the text from the input file. We use `getline` to read the file a line at a time and push each line onto the vector. Because `file` is a `shared_ptr`, we use the `->` operator to dereference `file` to fetch the `push_back` member of the `vector` to which `file` points.

Next we use an `istringstream` (§ 8.3, p. 321) to process each word in the line we just read. The inner `while` uses the `istringstream` input operator to read each word from the current line into `word`. Inside the `while`, we use the `map` subscript operator to fetch the `shared_ptr<set>` associated with `word` and bind `lines` to that pointer. Note that `lines` is a reference, so changes made to `lines` will be made to the element in `wm`.

If `word` wasn’t in the map, the subscript operator adds `word` to `wm` (§ 11.3.4, p. 435). The element associated with `word` is value initialized, which means that `lines` will be a null pointer if the subscript operator added `word` to `wm`. If `lines` is null, we allocate a new `set` and call `reset` to update the `shared_ptr` to which `lines` refers to point to this newly allocated `set`.

Regardless of whether we created a new `set`, we call `insert` to add the current line number. Because `lines` is a reference, the call to `insert` adds an element
to the set in \( \text{wm} \). If a given word occurs more than once in the same line, the call to \text{insert} does nothing.

**The QueryResult Class**

The \text{QueryResult} class has three data members: a \text{string} that is the word whose results it represents; a \text{shared_ptr} to the \text{vector} containing the input file; and a \text{shared_ptr} to the set of line numbers on which this word appears. Its only member function is a constructor that initializes these three members:

```cpp
class QueryResult {
friend std::ostream& print(std::ostream&, const QueryResult&);
public:
  QueryResult(std::string s,
               std::shared_ptr<std::set<line_no>> p,
               std::shared_ptr<std::vector<std::string>> f):
    sought(s), lines(p), file(f) { }
private:
  std::string sought; // word this query represents
  std::shared_ptr<std::set<line_no>> lines; // lines it's on
  std::shared_ptr<std::vector<std::string>> file; // input file
};
```

The constructor’s only job is to store its arguments in the corresponding data members, which it does in the constructor initializer list (§7.1.4, p. 265).

**The query Function**

The \text{query} function takes a \text{string}, which it uses to locate the corresponding set of line numbers in the map. If the string is found, the \text{query} function constructs a \text{QueryResult} from the given \text{string}, the \text{TextQuery} \text{file} member, and the set that was fetched from \text{wm}.

The only question is: What should we return if the given \text{string} is not found? In this case, there is no set to return. We’ll solve this problem by defining a local \text{static} object that is a \text{shared_ptr} to an empty set of line numbers. When the word is not found, we’ll return a copy of this \text{shared_ptr}:

```cpp
QueryResult
TextQuery::query(const string &sought) const
{
  // we'll return a pointer to this set if we don't find sought
  static shared_ptr<set<line_no>> nodata(new set<line_no>);
  // use find and not a subscript to avoid adding words to wm!
  auto loc = wm.find(sought);
  if (loc == wm.end())
    return QueryResult(sought, nodata, file); // not found
  else
    return QueryResult(sought, loc->second, file);
}
```
Printing the Results

The print function prints its given QueryResult object on its given stream:

```cpp
ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " ", ";
    // make plural (qr.lines->size(), "time", "s") << endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
    // don’t confound the user with text lines starting at 0
    os << \t(line " << num + 1 << ")
    << *(qr.file->begin() + num) << endl;
    return os;
}
```

We use the size of the set to which the qr.lines points to report how many matches were found. Because that set is in a shared_ptr, we have to remember to dereference lines. We call make_plural (§ 6.3.2, p. 224) to print time or times, depending on whether that size is equal to 1.

In the for we iterate through the set to which lines points. The body of the for prints the line number, adjusted to use human-friendly counting. The numbers in the set are indices of elements in the vector, which are numbered from zero. However, most users think of the first line as line number 1, so we systematically add 1 to the line numbers to convert to this more common notation.

We use the line number to fetch a line from the vector to which file points. Recall that when we add a number to an iterator, we get the element that many elements further into the vector (§ 3.4.2, p. 111). Thus, file->begin() + num is the numth element after the start of the vector to which file points.

Note that this function correctly handles the case that the word is not found. In this case, the set will be empty. The first output statement will note that the word occurred 0 times. Because *res.lines is empty, the for loop won’t be executed.

---

**Exercises Section 12.3.2**

**Exercise 12.30:** Define your own versions of the TextQuery and QueryResult classes and execute the runQueries function from § 12.3.1 (p. 486).

**Exercise 12.31:** What difference(s) would it make if we used a vector instead of a set to hold the line numbers? Which approach is better? Why?

**Exercise 12.32:** Rewrite the TextQuery and QueryResult classes to use a StrBlob instead of a vector<string> to hold the input file.

**Exercise 12.33:** In Chapter 15 we’ll extend our query system and will need some additional members in the QueryResult class. Add members named begin and end that return iterators into the set of line numbers returned by a given query, and a member named get_file that returns a shared_ptr to the file in the QueryResult object.
**Chapter Summary**

In C++, memory is allocated through `new` expressions and freed through `delete` expressions. The library also defines an `allocator` class for allocating blocks of dynamic memory.

Programs that allocate dynamic memory are responsible for freeing the memory they allocate. Properly freeing dynamic memory is a rich source of bugs: Either the memory is never freed, or it is freed while there are still pointers referring to the memory. The new library defines smart pointers—`shared_ptr`, `unique_ptr`, and `weak_ptr`—that make managing dynamic memory much safer. A smart pointer automatically frees the memory once there are no other users of that memory. When possible, modern C++ programs ought to use smart pointers.

**Defined Terms**

- **allocator** Library class that allocates unconstructed memory.

- **dangling pointer** A pointer that refers to memory that once had an object but no longer does. Program errors due to dangling pointers are notoriously difficult to debug.

- **delete** Frees memory allocated by `new`. `delete p` frees the object and `delete [] p` frees the array to which `p` points. `p` may be null or point to memory allocated by `new`.

- **deleter** Function passed to a smart pointer to use in place of `delete` when destroying the object to which the pointer is bound.

- **destructor** Special member function that cleans up an object when the object goes out of scope or is deleted.

- **dynamically allocated** Object that is allocated on the free store. Objects allocated on the free store exist until they are explicitly deleted or the program terminates.

- **free store** Memory pool available to a program to hold dynamically allocated objects.

- **heap** Synonym for free store.

- **new** Allocates memory from the free store. `new T` allocates an object of type `T` and returns a pointer to that object; if `T` is an array type, `new` returns a pointer to the first element in the array. Similarly, `new [n] T` allocates `n` objects of type `T` and returns a pointer to the first element in the array. By default, the allocated object is default initialized. We may also provide optional initializers.

- **placement new** Form of `new` that takes additional arguments passed in parentheses following the keyword `new`; for example, `new (nothrow) int` tells `new` that it should not throw an exception.

- **reference count** Counter that tracks how many users share a common object. Used by smart pointers to know when it is safe to delete memory to which the pointers point.

- **shared_ptr** Smart pointer that provides shared ownership: The object is deleted when the last `shared_ptr` pointing to that object is destroyed.

- **smart pointer** Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

- **unique_ptr** Smart pointer that provides single ownership: The object is deleted when the last `unique_ptr` pointing to that object is destroyed. `unique_ptr`s cannot be directly copied or assigned.

- **weak_ptr** Smart pointer that points to an object managed by a `shared_ptr`. The `shared_ptr` does not count `weak_ptr`s when deciding whether to delete its object.
This page intentionally left blank
Index

**Bold face** numbers refer to the page on which the term was first defined. Numbers in *italic* refer to the “Defined Terms” section in which the term is defined.

### What’s new in C++11

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>= default</td>
<td>265, 506</td>
</tr>
<tr>
<td>= delete</td>
<td>507</td>
</tr>
<tr>
<td>allocator, construct forwards to any constructor</td>
<td>482</td>
</tr>
<tr>
<td>array container</td>
<td>327</td>
</tr>
<tr>
<td>auto, 68</td>
<td></td>
</tr>
<tr>
<td>for type abbreviation</td>
<td>88, 129</td>
</tr>
<tr>
<td>not with dynamic array</td>
<td>478</td>
</tr>
<tr>
<td>with dynamic object</td>
<td>459</td>
</tr>
<tr>
<td>begin function</td>
<td>118</td>
</tr>
<tr>
<td>bind function</td>
<td>397</td>
</tr>
<tr>
<td>bitset enhancements</td>
<td>726</td>
</tr>
<tr>
<td>constexpr constructor, function, variable</td>
<td>299, 239, 66</td>
</tr>
<tr>
<td>container</td>
<td></td>
</tr>
<tr>
<td>cbegin and cend</td>
<td>109, 334</td>
</tr>
<tr>
<td>emplace members</td>
<td>345</td>
</tr>
<tr>
<td>insert return type</td>
<td>344</td>
</tr>
<tr>
<td>nonmember swap</td>
<td>339</td>
</tr>
<tr>
<td>of container</td>
<td>97, 329</td>
</tr>
<tr>
<td>shrink_to_fit</td>
<td>357</td>
</tr>
<tr>
<td>decltype, 70</td>
<td></td>
</tr>
<tr>
<td>function return type</td>
<td>250</td>
</tr>
<tr>
<td>delegating constructor</td>
<td>291</td>
</tr>
<tr>
<td>deleted copy-control</td>
<td>624</td>
</tr>
<tr>
<td>division rounding</td>
<td>141</td>
</tr>
<tr>
<td>end function</td>
<td>118</td>
</tr>
<tr>
<td>enumeration</td>
<td></td>
</tr>
<tr>
<td>controlling representation</td>
<td>834</td>
</tr>
<tr>
<td>forward declaration</td>
<td>834</td>
</tr>
<tr>
<td>scoped</td>
<td>832</td>
</tr>
<tr>
<td>explicit conversion operator</td>
<td>582</td>
</tr>
<tr>
<td>explicit instantiation</td>
<td>675</td>
</tr>
<tr>
<td>final class</td>
<td>600</td>
</tr>
<tr>
<td>format control for floating-point</td>
<td>757</td>
</tr>
<tr>
<td>forward function</td>
<td>694</td>
</tr>
<tr>
<td>forward_list container</td>
<td>327</td>
</tr>
<tr>
<td>function interface to callable objects</td>
<td>577</td>
</tr>
<tr>
<td>in-class initializer</td>
<td>73, 274</td>
</tr>
<tr>
<td>inherited constructor</td>
<td>628, 804</td>
</tr>
<tr>
<td>initializer_list</td>
<td>220</td>
</tr>
<tr>
<td>inline namespace</td>
<td>790</td>
</tr>
<tr>
<td>lambda expression</td>
<td>388</td>
</tr>
<tr>
<td>list initialization</td>
<td></td>
</tr>
<tr>
<td>= (assignment)</td>
<td>145</td>
</tr>
<tr>
<td>container</td>
<td>336, 423</td>
</tr>
<tr>
<td>dynamic array</td>
<td>478</td>
</tr>
<tr>
<td>dynamic object</td>
<td>459</td>
</tr>
<tr>
<td>pair</td>
<td>431</td>
</tr>
<tr>
<td>return value</td>
<td>226, 427</td>
</tr>
<tr>
<td>variable</td>
<td>43</td>
</tr>
<tr>
<td>vector</td>
<td>98</td>
</tr>
<tr>
<td>long long</td>
<td>33</td>
</tr>
<tr>
<td>mem_fn function</td>
<td>843</td>
</tr>
<tr>
<td>move function</td>
<td>533</td>
</tr>
<tr>
<td>move avoids copies</td>
<td>529</td>
</tr>
<tr>
<td>move constructor</td>
<td>534</td>
</tr>
<tr>
<td>move iterator</td>
<td>543</td>
</tr>
<tr>
<td>move-enabled this pointer</td>
<td>546</td>
</tr>
<tr>
<td>noexcept</td>
<td></td>
</tr>
<tr>
<td>exception specification</td>
<td>535, 779</td>
</tr>
<tr>
<td>operator</td>
<td>780</td>
</tr>
<tr>
<td>nullptr, 54</td>
<td></td>
</tr>
<tr>
<td>random-number library</td>
<td>745</td>
</tr>
<tr>
<td>range for statement</td>
<td>91, 187</td>
</tr>
<tr>
<td>not with dynamic array</td>
<td>477</td>
</tr>
<tr>
<td>regular expression-library</td>
<td>728</td>
</tr>
<tr>
<td>rvalue reference</td>
<td>532</td>
</tr>
<tr>
<td>cast from lvalue</td>
<td>691</td>
</tr>
<tr>
<td>reference collapsing</td>
<td>688</td>
</tr>
<tr>
<td>sizeof data member</td>
<td>157</td>
</tr>
<tr>
<td>sizeof... operator</td>
<td>700</td>
</tr>
</tbody>
</table>
smart pointer, 450
  shared_ptr, 450
  unique_ptr, 470
  weak_ptr, 473
string
  numeric conversions, 367
  parameter with IO types, 317
template
  function template default template argument, 670
  type alias, 666
  type parameter as friend, 666
  variadic, 699
  variadics and forwarding, 704
trailing return type, 229
  in function template, 684
  in lambda expression, 396
tuple, 718
  type alias declaration, 68
  union member of class type, 848
unordered containers, 443
virtual function
  final, 606
  override, 596, 606

Symbols
  ... (ellipsis parameter), 222
  /\* / (block comment), 9, 26
  // (single-line comment), 9, 26
  = default, 265, 306
  copy-control members, 506
  default constructor, 265
  = delete, 507
  copy control, 507–508
  default constructor, 507
  function matching, 508
  move operations, 538
__DATE__, 242
__FILE__, 242
__LINE__, 242
__TIME__, 242
__cplusplus, 860
\0 (null character), 39
\Xnnn (hexadecimal escape sequence), 39
\n (newline character), 39
\t (tab character), 39
\nnnn (octal escape sequence), 39
[ ] (curly brace), 2, 26
#include, 6, 28
  standard header, 6
  user-defined header, 21
#define, 77, 80
#undef, 77, 80
#ifdef, 77, 80
#undefif, 77, 80
~classname, see destructor
; (semicolon), 3
  class definition, 73
  null statement, 172
++ (increment), 12, 28, 147–149, 170
  iterator, 107, 132
  overloaded operator, 566–568
  pointer, 118
  precedence and associativity, 148
  reverse iterator, 407
StrBlobPtr, 566
-- (decrement), 13, 28, 147–149, 170
  iterator, 107
  overloaded operator, 566–568
  pointer, 118
  precedence and associativity, 148
  reverse iterator, 407, 408
StrBlobPtr, 566
* (dereference), 53, 80, 448
  iterator, 107
  map iterators, 429
  overloaded operator, 569
  pointer, 53
  precedence and associativity, 148
  smart pointer, 451
StrBlobPtr, 569
& (address-of), 52, 80
  overloaded operator, 554
-> (arrow operator), 110, 132, 150
  overloaded operator, 569
StrBlobPtr, 569
. (dot), 23, 28, 150
-** (pointer to member arrow), 837
. * (pointer to member dot), 837
[ ] (subscript), 93
  array, 116, 132
  array, 347
  bitset, 727
  deque, 347
  does not add elements, 104
  map, and unordered_map, 435, 448
    adds element, 435
  multidimensional array, 127
  out-of-range index, 93
  overloaded operator, 564
  pointer, 121
Index

string, 93, 132, 347
StrVec, 565
subscription range, 95
vector, 103, 132, 347
() (call operator), 23, 28, 202, 252
absInt, 571
const member function, 573
execution flow, 203
overloaded operator, 571
PrintString, 571
size_t, 573
string, 89
- (subtraction), 140
  iterator, 111
  pointer, 119
* (multiplication), 140
/ (division), 140
  rounding, 141
% (modulus), 141
  grading program, 176
== (equality), 18, 28
  arithmetic conversion, 144
  container, 88, 102, 340, 341
  iterator, 106, 107
  overloaded operator, 561, 562
  pointer, 55, 120
Sales_data, 561
  string, 88
tuple, 720
  unordered container key_type, 443
  used in algorithms, 377, 385, 413
vs. == (assignment), 146
!= (inequality), 28
  arithmetic conversion, 144
  container, 88, 102, 340, 341
  iterator, 106, 107
  overloaded operator, 562
  pointer, 55, 120
Sales_data, 561
  string, 88
tuple, 720
  used in algorithms, 378, 385, 413
< (less-than), 28, 143
  container, 88, 340
  ordered container key_type, 425
  overloaded operator, 562
  strict weak ordering, 562
  string, 88
tuple, 720
  used in algorithms, 378, 385, 413
<= (less-than-or-equal), 12, 28, 143
  container, 88, 340
  string, 88
> (greater-than), 28, 143
  container, 88, 340
  string, 88
>= (greater-than-or-equal), 28, 143
  container, 88, 340
  string, 88
>> (input operator), 8, 28
  as condition, 15, 86, 312
  chained-input, 8
istream, 8
istream_iterator, 403
overloaded operator, 558–559
precedence and associativity, 155
Sales_data, 558
Sales_item, 21
string, 85, 132
<< (output operator), 7, 28
bitset, 727
chained output, 7
ostream, 7
ostream_iterator, 405
overloaded operator, 557–558
precedence and associativity, 155
Query, 641
Sales_data, 557
Sales_item, 21
string, 85, 132
>> (right-shift), 153, 170
<< (left-shift), 153, 170
&& (logical AND), 94, 132, 142, 169
order of evaluation, 138
overloaded operator, 554
short-circuit evaluation, 142
|| (logical OR), 142
order of evaluation, 138
overloaded operator, 554
short-circuit evaluation, 142
& (bitwise AND), 154, 169
Query, 638, 644
!(logical NOT), 87, 132, 143, 170
|| (logical OR), 132, 170
| (bitwise OR), 154, 170
Query, 638, 644
^ (bitwise XOR), 154, 170
~ (bitwise NOT), 154, 170
Query, 638, 643
, (comma operator), 157, 169
order of evaluation, 138
overloaded operator, 554
?: (conditional operator), 151, 169
order of evaluation, 138
precedence and associativity, 151
+ (unary plus), 140
- (unary minus), 140
L’c’ (wchar_t literal), 38
ddd.dddL or ddd.ddd1 (long double literal), 41
numE or numE (double literal), 39
numF or numF (float literal), 41

numL or numl (long literal), 41
numLL or numll (long long literal), 41
numU or numu (unsigned literal), 41
class member: constant expression, see bit-field

A
absInt, 571
() (call operator), 571
abstract base class, 610, 649
BinaryQuery, 643
Disc_quote, 610
Query_base, 636
abstract data type, 254, 305
access control, 611–616
class derivation list, 596
default inheritance access, 616
default member access, 268
derived class, 613
derived-to-base conversion, 613
design, 614
inherited members, 612
local class, 853
nested class, 844
private, 268
protected, 595, 611
public, 268
using declaration, 615
access specifier, 268, 305
accessible, 611, 649
derived-to-base conversion, 613
Account, 301
accumulate, 379, 882
bookstore program, 406
Action, 839
adaptor, 372
back_inserter, 402
ccontainer, 368, 368–371
front_inserter, 402
inserter, 402
make_move_iterator, 543
add, Sales_data, 261
add_item, Basket, 633
add_to_Folder, Message, 522
address, 33, 78
adjacent_difference, 882
adjacent_find, 871
advice
always initialize a pointer, 54
avoid casts, 165
avoid undefined behavior, 36
choosing a built-in type, 34
define small utility functions, 277
define variables near first use, 48
don’t create unnecessary regex objects, 733
forwarding parameter pattern, 706
keep lambda captures simple, 394
managing iterators, 331, 354
prefix vs. postfix operators, 148
rule of five, 541
use move sparingly, 544
use constructor initializer lists, 289
when to use overloading, 233
writing compound expressions, 139
aggregate class, 298, 305
initialization, 298
algorithm header, 376
algorithms, 376, 418
see also Appendix A
architecture
_copy versions, 383, 414
_if versions, 414
operate on iterators not containers, 378
overloading pattern, 414
parameter pattern, 412–413
read-only, 379–380
reorder elements, 383–385, 414
write elements, 380–383
associative container and, 430
bind as argument, 397
can’t change container size, 385
element type requirements, 377
function object arguments, 572
iterator category, 410–412
iterator range, 376
lambda as argument, 391, 396
library function object, 575
ostream_iterator, 404
sort comparison, requires strict weak ordering, 425
supplying comparison operation, 386, 413
function, 386
lambda, 389, 390
type independence, 377
use element’s == (equality), 385, 413
use element’s < (less-than), 385, 413
accumulate, 379
bookstore program, 406
copy, 382
count, 378
equal_range, 722
equal, 380
find_if, 388, 397, 414
find, 376
for_each, 391
replace_copy, 383
replace, 383
set_intersection, 647
sort, 384
stable_sort, 387
transform, 396
unique, 384
alias declaration
namespace, 792, 817
template type, 666
type, 68
all_of, 871
alloc_n_copy, StrVec, 527
allocate, allocator, 481
allocator, 481, 481–483, 491, 524–531
allocate, 481, 527
compared to operator new, 823
construct, 482
forwards to constructor, 527
deallocate, 483, 528
compared to operator delete, 823
destroy, 482, 528
alternative operator name, 46
alternative_sum, program, 682
ambiguous
conversion, 583–589
multiple inheritance, 806
function call, 234, 245, 251
multiple inheritance, 808
overloaded operator, 588
AndQuery, 637
class definition, 644
eval function, 646
anonymous union, 848, 862
any, bitset, 726
any_of, 871
app (file mode), 319
append, string, 362
argc, 219
argument, 23, 26, 202, 251
array, 214–219
  buffer overflow, 215
to pointer conversion, 214
C-style string, 216
cconversion, function matching, 234
default, 236
forwarding, 704
initializes parameter, 203
iterator, 216
low-level const, 213
main function, 218
multidimensional array, 218
nonreference parameter, 209
pass by reference, 210, 252
pass by value, 209, 252
  uses copy constructor, 498
  uses move constructor, 539, 541
passing, 208–212
pointer, 214
reference parameter, 210, 214
reference to const, 211
top-level const, 212
argument list, 202
argument-dependent lookup, 797
  move and forward, 798
argv, 219
arithmetic
  conversion, 35, 159, 168
    in equality and relational operators, 144
    integral promotion, 160, 169
    signed to unsigned, 34
to bool, 162
operators, 139
  compound assignment (e.g., +=), 147
function object, 574
  overloaded, 560
type, 32, 78
  machine-dependent, 32
arithmetic (addition and subtraction)
  iterators, 111, 131
pointers, 119, 132
array, 113–130
  [ ] (subscript), 116, 132
  argument and parameter, 214–219
  argument conversion, 214
auto returns pointer, 117
begin function, 118
compound type, 113
conversion to pointer, 117, 161
  function arguments, 214
template arguments, 214
decltype argument deduction, 679
decltype returns array type, 118
definition, 113
dimension, constant expression, 113
dynamically allocated, 476, 476–484
  allocator, 481
can’t use begin and end, 477
can’t use range for statement, 477
delete [], 478
empty array, 478
new [], 477
shared_ptr, 480
unique_ptr, 479
elements and destructor, 502
end function, 118
initialization, 114
initializer of vector, 125
multidimensional, 125–130
no copy or assign, 114
of char initialization, 114
parameter
  buffer overflow, 215
  converted to pointer, 117
  function template, 654
  pointer to, 218
  reference to, 217
return type, 204
  trailing, 229
type alias, 229
decltype, 230
sizeof, 157
subscript range, 116
subscript type, 116
understanding complicated declarations, 115
array
see also container
see also sequential container
[ ] (subscript), 347
= (assignment), 337
assign, 338
copy initialization, 337
default initialization, 336
definition, 336
header, 329
initialization, 334–337
list initialization, 337
overview, 327
random-access iterator, 412
swap, 339
assert preprocessor macro, 241, 251
assign
array, 338
invalidates iterator, 338
sequential container, 338
string, 362
assignment, vs. initialization, 42, 288
assignment operators, 144–147
associative array, see map
associative container, 420, 447
and library algorithms, 430
initialization, 423, 424
key_type requirements, 425, 445
members
begin, 430
count, 437, 438
emplace, 432
end, 430
equal_range, 439
erase, 434
find, 437, 438
insert, 432
key_type, 428, 447
mapped_type, 428, 448
value_type, 428, 448
override default comparison, 425
override default hash, 446
overview, 423
associativity, 134, 136–137, 168
= (assignment), 145
?: (conditional operator), 151
dot and dereference, 150
increment and dereference, 148
IO operator, 155
overloaded operator, 553
at
deque, 348
map, 435
string, 348
unordered_map, 435
vector, 348
ate (file mode), 319
auto, 68, 78
cbegin, 109, 379
cend, 109, 379
for type abbreviation, 88, 129
of array, 117
of reference, 69
pointer to function, 249
with new, 459
auto_ptr deprecated, 471
automatic object, 205, 251
see also local variable
see also parameter
and destructor, 502
avg_price, Sales_data, 259

B
back
queue, 371
sequential container, 346
StrBlob, 457
back_inserter, 382, 402, 417
requires push_back, 382, 402
bad, 313
bad_alloc, 197, 460
bad_cast, 197, 826
bad_typeid, 828
badbit, 312
base, reverse iterator, 409
base class, 592, 649
see also virtual function
abstract, 610, 649
base-to-derived conversion, not automatic, 602
can be a derived class, 600
definition, 594
derived-to-base conversion, 597
accessibility, 613
key concepts, 604
multiple inheritance, 805
final, 600
friendship not inherited, 614
initialized or assigned from derived, 603
member hidden by derived, 619
member new and delete, 822
multiple, see multiple inheritance
must be complete type, 600
protected member, 611
scope, 617
inheritance, 617–621
multiple inheritance, 807
virtual function, 620
static members, 599
user of, 614
virtual, see virtual base class
virtual destructor, 622
Basket, 631
add_item, 633
total, 632
Bear, 803
virtual base class, 812
before_begin, forward_list, 351
begin
  associative container, 430
  container, 106, 131, 333, 372
  function, 118, 131
    not with dynamic array, 477
  multidimensional array, 129
StrBlob, 475
StrVec, 526
bernoulli_distribution, 752
best match, 234, 251
  see also function matching
bidirectional iterator, 412, 417
bigness program, 391
binary (file mode), 319
binary operators, 134, 168
  overloaded operator, 552
binary predicate, 386, 417
binary_function deprecated, 579
binary_search, 873
BinaryQuery, 637
  abstract base class, 643
bind, 397, 417
  check_size, 398
  generates callable object, 397
    from pointer to member, 843
  placeholders, 399
  reference parameter, 400
bind1st deprecated, 401
bind2nd deprecated, 401
binops desk calculator, 577
bit-field, 854, 862
  access to, 855
  constant expression, 854
bitset, 723, 723–728, 769
  [ ] (subscript), 727
  << (output operator), 727
  any, 726
  count, 727
  flip, 727
  grading program, 728
  header, 723
  initialization, 723–725
    from string, 724
    from unsigned, 723
  none, 726
  reset, 727
  set, 727
  test, 727
  to_ulong, 727
bitwise, bitset, operators, 725
bitwise operators, 152–156
  += (compound assignment), 155
  compound assignment (e.g., +=), 147
  grading program, 154
  operand requirements, 152
Blob
class template, 659
  constructor, 662
    initializer_list, 662
    iterator parameters, 673
    instantiation, 660
    member functions, 661–662
block, 2, 12, 26, 173, 199
  function, 204
  scope, 48, 80, 173
  try, 193, 194, 200, 818
block (/* */), comment, 9, 26
book from author program, 438–440
bookstore program
  Sales_data, 255
    using algorithms, 406
    Sales_item, 24
bool, 32
  conversion, 35
  literal, 41
    in condition, 143
boolalpha, manipulator, 754
brace, curly, 2, 26
braced list, see list initialization
break statement, 190, 199
  in switch, 179–181
bucket management, unordered container, 444
buffer, 7, 26
  flushing, 314
buffer overflow, 105, 116, 131
  array parameter, 215
  C-style string, 123
buildMap program, 442
built-in type, 2, 26, 32–34
  default initialization, 43
Bulk_quote
class definition, 596
  constructor, 598, 610
derived from Disc_quote, 610
design, 592
synthesized copy control, 623
byte, 33, 78
Index 895

C
.c file, 4
.cc file, 4
.cpp file, 4
.cp file, 4
C library header, 91
C-style cast, 164
C-style string, 114, 122, 122–123, 131
buffer overflow, 123
initialization, 122
parameter, 216
string, 124
c_str, 124
call by reference, 208, 210, 251
call by value, 209, 251
uses copy constructor, 498
uses move constructor, 539
call signature, 576, 590
callable object, 388, 417, 571–572
absInt, 571
bind, 397
call signature, 576
function and function pointers, 388
function objects, 572
pointer to member
and bind, 843
and function, 842
and mem_fn, 843
not callable, 842
PrintString, 571
ShorterString, 573
SizeComp, 573
with function, 576–579
with algorithms, 390
candidate function, 243, 251
see also function matching
function template, 695
namespace, 800
overloaded operator, 587
capacity
string, 356
StrVec, 526
vector, 356
capture list, see lambda expression
case label, 179, 179–182, 199
default, 181
constant expression, 179
case sensitive, string, 365
cassert header, 241
cast, see also named cast, 168
checked, see dynamic_cast
old-style, 164
to rvalue reference, 691
catch, 193, 195, 199, 775, 816
catch(...), 777, 816
exception declaration, 195, 200, 775, 816
exception object, 775
matching, 776
ordering of, 776
runtime_error, 195
catch all (catch(...)), 777, 816
cautions
ambiguous conversion operator, 581
conversions to unsigned, 37
dynamic memory pitfalls, 462
exception safety, 196
IO buffers, 315
overflow, 140
overloaded operator misuse, 555
overloaded operators and conversion operators, 586
smart pointer, pitfalls, 469
uninitialized variables, 45
using directives cause pollution, 795
cbegin
auto, 109, 379
decltype, 109, 379
container, 109, 333, 334, 372
cctype
functions, 91–93
header, 91
cend
auto, 109, 379
decltype, 109, 379
container, 109, 333, 334, 372
cerr, 6, 26
chained input, 8
chained output, 7
char, 32
signed, 34
unsigned, 34
array initialization, 114
literal, 39
representation, 34
char16_t, 33
char32_t, 33
character
newline (\n), 39
nonprintable, 39, 79
null (\0), 39
tab (\t), 39
character string literal, see string literal
check
    StrBlob, 457
    StrBlobPtr, 474
check_size, 398
bind, 398
checked cast, see dynamic_cast
children’s story program, 383–391
chk_n_alloc, StrVec, 526
cin, 6, 26
tied to cout, 315
cl, 5
class, 19, 26, 72, 305
    see also constructor
    see also destructor
    see also member function
    see also static member
access specifier, 268
default, 268
private, 268, 306
public, 268, 306
aggregate, 298, 305
assignment operator
    see copy assignment
    see move assignment
base, see base class, 649
data member, 73, 78
    const vs. mutable, 274
    const, initialization, 289
in-class initializer, 274
initialization, 263, 274
must be complete type, 279
mutable, 274, 306
order of destruction, 502
order of initialization, 289
pointer, not deleted, 503
reference, initialization, 289
sizeof, 157
declaration, 278, 305
default inheritance specifier, 616
definition, 72, 256–267
    ends with semicolon, 73
derived, see derived class, 649
exception, 193, 200
final specifier, 600
forward declaration, 279, 306
friend, 269, 280
class, 280
function, 269
member function, 280
overloaded function, 281
scope, 270, 281
    template class or function, 664
implementation, 254
interface, 254
literal, 299
local, see local class
member, 73, 78
member access, 282
member new and delete, 822
member : constant expression, see bit-field
multiple base classes, see multiple inheritance
name lookup, 284
nested, see nested class
pointer to member, see pointer to member
preventing copies, 507
scope, 73, 282, 282–287, 305
synthesized, copy control, 267, 497, 500, 503, 537
template member, see member template
type member, 271
    : : (scope operator), 282
user of, 255
valuelike, 512
without move constructor, 540
class
    compared to typename, 654
default access specifier, 268
default inheritance specifier, 616
template parameter, 654
class derivation list, 596
    access control, 612
default access specifier, 616
direct base class, 600
indirect base class, 600
multiple inheritance, 803
virtual base class, 812
class template, 96, 131, 658, 659, 658–667, 713
    see also template parameter
    see also instantiation
    B1ob, 659
declaration, 669
default template argument, 671
definition, 659
error detection, 657
explicit instantiation, 675, 675–676
explicit template argument, 660
friend, 664
   all instantiations, 665
declaration dependencies, 665
same instantiation, 664
specific instantiation, 665
instantiation, 660
member function
   defined outside class body, 661
   instantiation, 663
member template, see member template
specialization, 707, 709–712, 714
   hash<key_type>, 709, 788
member, 711
namespace, 788
partial, 711, 714
static member
   accessed through an instantiation, 667
definition, 667
template argument, 660
template parameter, used in definition, 660
type parameter as friend, 666
type-dependent code, 658
class type, 19, 26
conversion, 162, 305, 590
   ambiguities, 587
conversion operator, 579
converting constructor, 294
impact on function matching, 584
overloaded function, 586
with standard conversion, 581
default initialization, 44
initialization, 73, 84, 262
union member of, 848
variable vs. function declaration, 294
clear
   sequential container, 350
   stream, 313
clog, 6, 26
close, file stream, 318
cmatch, 733
cmath header, 751, 757
collapsing rule, reference, 688
combine, Sales_data, 259
comma (,), operator, 157
comment, 9, 26
   block (/* */), 9, 26
   single-line (//), 9, 26
compare
default template argument, 670
   function template, 652
default template argument, 670
   explicit template argument, 683
   specialization, 706
   string literal version, 654
template argument deduction, 680
   string, 366
compareIsbn
   and associative container, 426
   Sales_data, 387
compilation
   common errors, 16
   compiler options, 207
   conditional, 240
declaration vs. definition, 44
   mixing C and C++, 860
needed when class changes, 270
templates, 656
   error detection, 657
   explicit instantiation, 675–676
compiler
   extension, 114, 131
   GNU, 5
   Microsoft, 5
   options for separate compilation, 207
composition vs. inheritance, 637
compound assignment (e.g., +=)
   arithmetic operators, 147
   bitwise operators, 147
compound expression, see expression
compound statement, 173, 199
compound type, 50, 50–58, 78
   array, 113
   declaration style, 57
   understanding complicated declarations, 115
concatenation
   string, 89
   string literal, 39
condition, 12, 26
   = (assignment) in, 146
conversion, 159
do while statement, 189
for statement, 13, 185
if statement, 18, 175
in IO expression, 156
logical operators, 141
smart pointer as, 451
stream type as, 15, 162, 312
while statement, 12, 183
condition state, IO classes, 312, 324
conditional compilation, 240
conditional operator (? :), 151
connection, 468
console window, 6
const, 59, 78
and typedef, 68
conversion, 162
template argument deduction, 679
dynamically allocated
destruction, 461
initialization, 460
initialization, 59
class type object, 262
low-level const, 64
argument and parameter, 213
conversion from, 163
conversion to, 162
overloaded function, 232
template argument deduction, 693
member function, 258, 305
() (call operator), 573
not constructors, 262
overloaded function, 276
reference return, 276
parameter, 212
function matching, 246
overloaded function, 232
pointer, 63, 78
pointer to, 62, 79
conversion from nonconst, 162
initialization from nonconst, 62
overloaded parameter, 232
reference, see reference to const
top-level const, 64
and auto, 69
argument and parameter, 212
decltype, 71
parameter, 232
template argument deduction, 679
variable, 59
declared in header files, 76
extern, 60
local to file, 60
const_cast, 163, 163
const_iterator, container, 108, 332
const_reference, container, 333
const_reverse_iterator, container, 332, 407
constant expression, 65, 78
array dimension, 113
bit-field, 854
case label, 179
enumerator, 833
integral, 65
nontype template parameter, 655
sizeof, 156
static data member, 303
constexpr, 66, 78
constructor, 299
declared in header files, 76
function, 239, 251
nonconstant return value, 239
function template, 655
pointer, 67
variable, 66
construct
allocator, 482
forwards to constructor, 527
constructor, 262, 264, 262–266, 305
see also default constructor
see also copy constructor
see also move constructor
calls to virtual function, 627
constexpr, 299
converting, 294, 305
function matching, 585
Sales_data, 295
with standard conversion, 580
default argument, 290
delegating, 291, 306
derived class, 598
initializes direct base class, 610
initializes virtual base, 813
explicit, 296, 306
function try block, 778, 817
inherited, 628
initializer list, 265, 288–292, 305
class member initialization, 274
compared to assignment, 288
derived class, 598
function try block, 778, 817
sometimes required, 288
virtual base class, 814
initializer_list parameter, 662
not const, 262
order of initialization, 289
derived class object, 598, 623
multiple inheritance, 804
virtual base classes, 814
overloaded, 262
Index

StrBlob, 456
StrBlobPtr, 474
TextQuery, 488
Blob, 662
  initializer_list, 662
  iterator parameters, 673
Bulk_quote, 598, 610
Disc_quote, 609
Sales_data, 264–266
container, 96, 131, 326, 372
see also sequential container
see also associative container
adaptor, 368, 368–371
  equality and relational operators, 370
  initialization, 369
  requirements on container, 369
and inheritance, 630
as element type, 97, 329
associative, 420, 447
copy initialization, 334
element type constraints, 329, 341
elements and destructor, 502
elements are copies, 342
initialization from iterator range, 335
list initialization, 336
members
  see also iterator
= (assignment), 337
== (equality), 341
!= (inequality), 341
begin, 106, 333, 372
cbegin, 109, 333, 334, 372
cend, 109, 333, 334, 372
const_iterator, 108, 332
const_reference, 333
const_reverse_iterator, 332, 407
crbegin, 333
crend, 333
difference_type, 131, 332
empty, 87, 102, 131, 340
end, 106, 131, 333, 373
equality and relational operators,
  88, 102, 340
iterator, 108, 332
rbegin, 333, 407
reference, 333
relational operators, 341
rend, 333, 407
reverse_iterator, 332, 407
size, 88, 102, 132, 340
size_type, 88, 102, 132, 332
swap, 339
move operations, 529
  moved-from object is valid but un-
specified, 537
nonmember swap, 339
of container, 97, 329
overview, 328
sequential, 326, 373
type members, :: (scope operator), 333
continue statement, 191, 199
control, flow of, 11, 172, 200
conversion, 78, 159, 168
= (assignment), 145, 159
ambiguous, 583–589
argument, 203
arithmetic, 35, 159, 168
array to pointer, 117
  argument, 214
  exception object, 774
  multidimensional array, 128
  template argument deduction, 679
base-to-derived, not automatic, 602
bool, 35
class type, 162, 294, 305, 590
  ambiguities, 587
  conversion operator, 579
  function matching, 584, 586
  with standard conversion, 581
condition, 159
derived-to-base, 597, 649
  accessibility, 613
  key concepts, 604
  shared_ptr, 630
floating-point, 35
function to pointer, 248
  exception object, 774
  template argument deduction, 679
integral promotion, 160, 169
istream, 162
multiple inheritance, 805
  ambiguous, 806
narrowing, 43
operand, 159
pointer to bool, 162
rank, 245
return value, 223
Sales_data, 295
signed type, 160
signed to unsigned, 34
to const, 162
  from pointer to nonconst, 62
  from reference to nonconst, 61
template argument deduction, 679
unscoped enumeration to integer, 834
unsigned, 36
virtual base class, 812
conversion operator, 580, 580–587, 590
design, 581
explicit, 582, 590
bool, 583
function matching, 585, 586
SmallInt, 580
used implicitly, 580
with standard conversion, 580
converting constructor, 294, 305
function matching, 585
with standard conversion, 580
_copy algorithms, 383, 414
copy, 382, 874
copy and swap assignment, 518
  move assignment, 540
  self-assignment, 519
copy assignment, 500–501, 549
  = default, 506
  = delete, 507
base from derived, 603
copy and swap, 518, 549
derived class, 626
HasPtr
  reference counted, 516
  valuelike, 512
memberwise, 500
Message, 523
preventing copies, 507
private, 509
reference count, 514
rule of three/five, 505
  virtual destructor exception, 622
self-assignment, 512
StrVec, 528
synthesized, 500, 550
  deleted function, 508, 624
derived class, 623
  multiple inheritance, 805
union with class type member, 851
  used for copy-initialization, 498
copy control, 267, 496, 549
  = delete, 507–508
inheritance, 623–629
memberwise, 267, 550
copy assignment, 500
  copy constructor, 497
  move assignment, 538
  move constructor, 538
multiple inheritance, 805
synthesized, 267
  as deleted function, 508
  as deleted in derived class, 624
  move operations as deleted function, 538
unions, 849
virtual base class, synthesized, 815
copy initialization, 84, 131, 497, 497–499, 549
array, 337
  container, 334
  container elements, 342
  explicit constructor, 498
invalid for arrays, 114
move vs. copy, 539
parameter and return value, 498
uses copy constructor, 497
uses move constructor, 541
copy_backward, 875
copy_if, 874
copy_n, 874
copyUnion, Token, 851
count
  algorithm, 378, 871
  associative container, 437, 438
  bitset, 727
count_calls, program, 206
count_if, 871
cout, 6, 26
tied to cin, 315
cplusplus_primer, namespace, 787
crbegin, container, 333
cref, binds reference parameter, 400, 417
cregex_iterator, 733, 769
crend, container, 333
cstddef header, 116, 120
cstdio header, 762
cstdlib header, 54, 227, 778, 823
cstring
  functions, 122–123
  header, 122
csub_match, 733, 769
ctime header, 749
curly brace, 2, 26
dangling else, 177, 199
dangling pointer, 225, 463, 491
  undefined behavior, 463
data abstraction, 254, 306
data hiding, 270
data member, see class data member
data structure, 19, 26
deallocate, allocator, 483, 528
debug_rep program
  additional nontemplate versions, 698
  general template version, 695
  nontemplate version, 697
  pointer template version, 696
DebugDelete, member template, 673
dec, manipulator, 754
decimal, literal, 38
declaration, 45, 78
  class, 278, 305
  class template, 669
  class type, variable, 294
  compound type, 57
  dependencies
    member function as friend, 281
    overloaded templates, 698
    template friends, 665
    template instantiation, 657
template specializations, 708
  variadic templates, 702
derived class, 600
explicit instantiation, 675
friend, 269
function template, 669
instantiation, 713
member template, 673
template, 669
template specialization, 708
type alias, 68
using, 82, 132
  access control, 615
  overloaded inherited functions, 621
variable, 45
  const, 60
declarator, 50, 79
decltype, 70, 79
array return type, 230
cbegin, 109, 379
cend, 109, 379
depends on form, 71
for type abbreviation, 88, 106, 129
of array, 118
of function, 250
pointer to function, 249
top-level const, 71
yields lvalue, 71, 135
decrement operators, 147–149
default argument, 236, 251
adding default arguments, 237
and header file, 238
constructor, 290
default constructor, 291
function call, 236
function matching, 243
initializer, 238
static member, 304
virtual function, 607
default case label, 181, 199
default constructor, 263, 306
  = default, 265
  = delete, 507
default argument, 291
Sales_data, 262
StrVec, 526
synthesized, 263, 306
  deleted function, 508, 624
  derived class, 623
Token, 850
used implicitly
default initialization, 293
value initialization, 293
default initialization, 43
array, 336
built-in type, 43
class type, 44
string, 44, 84
uses default constructor, 293
vector, 97
default template argument, 670
class template, 671
compare, 670
function template, 670
template<>, 671
default_random_engine, 745, 769
default float manipulator, 757
definition, 79
array, 113
associative container, 423
base class, 594
class, 72, 256–267
class template, 659
member function, 661
static member, 667
class template partial specialization, 711
derived class, 596
dynamically allocated object, 459
explicit instantiation, 675
function, 577
in if condition, 175
in while condition, 183
instantiation, 713
member function, 256–260
multidimensional array, 126
namespace, 785
can be discontiguous, 786
member, 788
overloaded operator, 500, 552
pair, 426
pointer, 52
pointer to function, 247
pointer to member, 836
reference, 51
sequential container, 334
shared_ptr, 450
static member, 302
string, 84
template specialization, 706–712
unique_ptr, 470, 472
variable, 41, 45

const, 60
variable after case label, 182
vector, 97
weak_ptr, 473
delegating constructor, 291, 306
delete, 460–463, 491
const object, 461
execution flow, 820
memory leak, 462
null pointer, 461
pointer, 460
runs destructor, 502
delete[], dynamically allocated array, 478
deleted function, 507, 549
deleter, 469, 491
shared_ptr, 469, 480, 491
unique_ptr, 472, 491
deprecated, 401
auto_ptr, 471
binary_function, 579
bind1st, 401
bind2nd, 401
generalized exception specification, 780
ptr_fun, 401
unary_function, 579
deque, 372
see also container, container member
see also sequential container
[] (subscript), 347
at, 348
header, 329
initialization, 334–337
list initialization, 336
overview, 327
push_back, invalidates iterator, 354
push_front, invalidates iterator, 354
random-access iterator, 412
value initialization, 336
deref, StrBlobPtr, 475
derived class, 592, 649
see also virtual function:
:: (scope operator) to access base class member, 607
= (assignment), 626
access control, 613
as base class, 600
assigned or copied to base object, 603
base-to-derived conversion, not automatic, 602
constructor, 598
  initializer list, 598
  initializes direct base class, 610
  initializes virtual base, 813
copy assignment, 626
copy constructor, 626
declaration, 600
default derivation specifier, 616
definition, 596
derivation list, 596, 649
  access control, 612
derived object
  contains base part, 597
  multiple inheritance, 803
derived-to-base conversion, 597
  accessibility, 613
  key concepts, 604
  multiple inheritance, 805
destructor, 627
direct base class, 600, 649
final, 600
friendship not inherited, 615
indirect base class, 600, 650
is user of base class, 614
member new and delete, 822
move assignment, 626
move constructor, 626
multiple inheritance, 803
name lookup, 617
order of destruction, 627
  multiple inheritance, 805
order of initialization, 598, 623
  multiple inheritance, 804
  virtual base classes, 814
scope, 617
  hidden base members, 619
  inheritance, 617–621
  multiple inheritance, 807
  name lookup, 618
  virtual function, 620
static members, 599
synthesized
  copy control members, 623
  deleted copy control members, 624
using declaration
  access control, 615
  overloaded inherited functions, 621
  virtual function, 596
derived-to-base conversion, 597, 649
  accessible, 613
  key concepts, 604
multiple inheritance, 805
not base-to-derived, 602
shared_ptr, 630
design
  access control, 614
  Bulk_quote, 592
  conversion operator, 581
  Disc_quote, 608
equality and relational operators, 562
generic programs, 655
inheritance, 637
Message class, 520
namespace, 786
overloaded operator, 554–556
Query classes, 636–639
Quote, 592
reference count, 514
StrVec, 525
destination sequence, 381, 413
destroy, allocator, 482, 528
destructor, 452, 491, 501, 501–503, 549
  = default, 506
called during exception handling, 773
calls to virtual function, 627
container elements, 502
derived class, 627
doesn’t delete pointer members, 503
explicit call to, 824
HasPtr
  reference counted, 515
  valuelike, 512
local variables, 502
Message, 522
not deleted function, 508
not private, 509
order of destruction, 502
  derived class, 627
  multiple inheritance, 805
  virtual base classes, 815
reference count, 514
rule of three/five, 505
  virtual destructor, exception, 622
run by delete, 502
shared_ptr, 453
should not throw exception, 774
StrVec, 528
synthesized, 503, 550
  deleted function, 508, 624
  derived class, 623
  multiple inheritance, 805
Token, 850
valuelike class, 512
virtual function, 622
virtual in base class, 622
development environment, integrated, 3
difference_type, 112
    vector, 112
    container, 131, 332
    string, 112
direct base class, 600
direct initialization, 84, 131
    emplace members use, 345
Disc_quote
    abstract base class, 610
    class definition, 609
    constructor, 609
    design, 608
discriminant, 849, 862
    Token, 850
distribution types
    bernoulli_distribution, 752
    default template argument, 750
    normal_distribution, 751
    random-number library, 745
    uniform_int_distribution, 746
    uniform_real_distribution, 750
divides<T>, 575
division rounding, 141
do while statement, 189, 200
domain_error, 197
double, 33
    literal (нумЕн or нумен), 38
    output format, 755
    output notation, 757
dynamic binding, 593, 650
    requirements for, 603
    static vs. dynamic type, 605
dynamic type, 601, 650
dynamic_cast, 163, 825, 825, 862
    bad_cast, 826
    to pointer, 825
    to reference, 826
dynamically allocated, 450, 491
array, 476, 476–484
    allocator, 481
    can’t use begin and end, 477
    can’t use range for statement, 477
    delete [], 478
    empty array, 478
    new [], 477
    returns pointer to an element, 477
    shared_ptr, 480
unique_ptr, 479
    delete runs destructor, 502
    lifetime, 450
    new runs constructor, 458
    object, 458–463
    const object, 460
    delete, 460
    factory program, 461
    initialization, 459
    make_shared, 451
    new, 458
    shared objects, 455, 486
    shared_ptr, 455, 486
    unique_ptr, 470
E
echo command, 4
ECMAScript, 730, 739
    regular expression library, 730
eedit-compile-debug, 16, 26
    errors at link time, 657
element type constraints, container, 329, 341
elimDups program, 383–391
ellipsis, parameter, 222
else, see if statement
emplace
    associative container, 432
    priority_queue, 371
    queue, 371
    sequential container, 345
    stack, 371
emplace_back
    sequential container, 345
    StrVec, 704
emplace_front, sequential container, 345
empty
    container, 87, 102, 131, 340
    priority_queue, 371
    queue, 371
    stack, 371
encapsulation, 254, 306
    benefits of, 270
end
    associative container, 430
    container, 106, 131, 333, 373
    function, 118, 131
    multidimensional array, 129
    StrBlob, 475
    StrVec, 526
Index

end-of-file, 15, 26, 762
character, 15
Endangered, 803
endl, 7
  manipulator, 314
ends, manipulator, 315
engine, random-number library, 745, 770
  default_random_engine, 745
  max, min, 747
  retain state, 747
  seed, 748, 770
enum, unscoped enumeration, 832
enum class, scoped enumeration, 832
enumeration, 832, 863
  as union discriminant, 850
  function matching, 835
  scoped, 832, 864
  unscoped, 832, 864
  conversion to integer, 834
  unnamed, 832
enumerator, 832, 863
  constant expression, 833
  conversion to integer, 834
eof, 313
eofbit, 312
equal, 380, 872
equal virtual function, 829
equal_range
  algorithm, 722, 873
  associative container, 439
equal_to<T>, 575
equality operators, 141
  arithmetic conversion, 144
  container adaptor, 370
  container member, 340
  iterator, 106
  overloaded operator, 561
  pointer, 120
  Sales_data, 561
string, 88
vector, 102
erase
  associative container, 434
  changes container size, 385
  invalidates iterator, 349
  sequential container, 349
  string, 362
error, standard, 6
error_type, 732
error_msg program, 221
ERRORLEVEL, 4
escape sequence, 39, 79
  hexadecimal (\xnnn), 39
  octal (\nnn), 39
eval function
  AndQuery, 646
  NotQuery, 647
  OrQuery, 645
exception
  class, 193, 200
  class hierarchy, 783
  deriving from, 782
    Sales_data, 783
  header, 197
  initialization, 197
  what, 195, 782
exception handling, 193–198, 772, 817
  see also throw
  see also catch
exception declaration, 195, 775, 816
  and inheritance, 775
    must be complete type, 775
    exception in destructor, 773
    exception object, 774, 817
    finding a catch, 776
    function try block, 778, 817
    handler, see catch
    local variables destroyed, 773
    noexcept specification, 535, 779, 817
    nonthrowing function, 779, 818
    safe resource allocation, 467
    stack unwinding, 773, 818
    terminate function, 196, 200
    try block, 194, 773
    uncaught exception, 773
    unhandled exception, 196
exception object, 774, 817
  catch, 775
  conversion to pointer, 774
  initializes catch parameter, 775
  pointer to local object, 774
  rethrow, 777
exception safety, 196, 200
  smart pointers, 467
exception specification
  argument, 780
  generalized, deprecated, 780
  noexcept, 779
  nonthrowing, 779
  pointer to function, 779, 818
  throw(), 780
  violation, 779
virtual function, 781
executable file, 5, 251
execution flow
   () (call operator), 203
delete, 820
   for statement, 186
new, 820
   switch statement, 180
   throw, 196, 773
EXIT_FAILURE, 227
EXIT_SUCCESS, 227
expansion
   forward, 705
   parameter pack, 702, 702–704, 714
      function parameter pack, 703
      template parameter pack, 703
   pattern, 702
explicit
   constructor, 296, 306
      copy initialization, 498
   conversion operator, 582, 590
   conversion to bool, 583
explicit call to
   destructor, 824
   overloaded operator, 553
   postfix operators, 568
explicit instantiation, 675, 713
explicit template argument, 660, 713
   class template, 660
   forward, 694
   function template, 682
      function pointer, 686
   template argument deduction, 682
exporting C++ to C, 860
expression, 7, 27, 134, 168
   callable, see callable object
   constant, 65, 78
   lambda, see lambda expression
   operand conversion, 159
   order of evaluation, 137
   parenthesized, 136
   precedence and associativity, 136–137
   regular, see regular expression
expression statement, 172, 200
extension, compiler, 114, 131
extern
   and const variables, 60
   explicit instantiation, 675
   variable declaration, 45
extern ’C’, see linkage directive

F
fact program, 202
factorial program, 227
factory program
   new, 461
      shared_ptr, 453
fail, 313
failbit, 312
failure, new, 460
file, source, 4
file extension, program, 730
   version 2, 738
file marker, stream, 765
file mode, 319, 324
file redirection, 22
file static, 792, 817
file stream, see fstream
fill, 380, 874
fill_n, 381, 874
final specifier, 600
   class, 600
      virtual function, 607
find
   algorithm, 376, 871
      associative container, 437, 438
      string, 364
find last word program, 408
find_char program, 211
find_first_of, 872
find_first_not_of, string, 365
find_first_of, 872
   string, 365
find_if, 388, 397, 414, 871
find_if_not, 871
find_if_not_of, 871
find_last_not_of, string, 366
find_last_of, string, 366
findBook, program, 721
fixed manipulator, 757
flip
   bitset, 727
   program, 694
flip1, program, 692
flip2, program, 693
float, 33
   literal (numF or numf), 41
floating-point, 32
   conversion, 35
   literal, 38
   output format, 755
output notation, 757
flow of control, 11, 172, 200
flush, manipulator, 315
Folder, see Message
for statement, 13, 27, 185, 185–187, 200
condition, 13
execution flow, 186
for header, 185
range, 91, 187, 187–189, 200
can’t add elements, 101, 188
multidimensional array, 128
for_each, 391, 872
format state, stream, 753
formatted IO, 761, 769
forward, 694
argument-dependent lookup, 798
explicit template argument, 694
pack expansion, 705
passes argument type unchanged, 694, 705
usage pattern, 706
forward declaration, class, 279, 306
forward iterator, 411, 417
forward_list
see also container
see also sequential container
before_begin, 351
forward iterator, 411
header, 329
initialization, 334–337
list initialization, 336
merge, 415
overview, 327
remove, 415
remove_if, 415
reverse, 415
splice_after, 416
unique, 415
value initialization, 336
forwarding, 692–694
passes argument type unchanged, 694
preserving type information, 692
rvalue reference parameters, 693, 705
typical implementation, 706
variadic template, 704
free, StrVec, 528
free library function, 823, 863
free store, 450, 491
friend, 269, 306
class, 280
class template type parameter, 666
declaration, 269
declaration dependencies
member function as friend, 281
template friends, 665
function, 269
inheritance, 614
member function, 280, 281
overloaded function, 281
scope, 270, 281
namespace, 799
template as, 664
front
queue, 371
sequential container, 346
StrBlob, 457
front_insert, 402, 417
compared to inserter, 402
requires push_front, 402
fstream, 316–320
close, 318
file marker, 765
file mode, 319
header, 310, 316
initialization, 317
off_type, 766
open, 318
pos_type, 766
random access, 765
random IO program, 766
seek and tell, 763–768
function, 2, 27, 202, 251
see also return type
see also return value
block, 204
body, 2, 27, 202, 251
callable object, 388
candidate, 251
candidate function, 243
constexpr, 239, 251
nonconstant return value, 239
declaration, 206
declaration and header file, 207
decltype returns function type, 250
default argument, 236, 251
adding default arguments, 237
and header file, 238
initializer, 238
deleted, 507, 549
function matching, 508
exception specification
noexcept, 779
throw(), 780
friend, 269
function to pointer conversion, 248
inline, 238, 252
and header, 240
linkage directive, 859
member, see member function
name, 2, 27
nonthrowing, 779, 818
overloaded
  compared to redeclaration, 231
  friend declaration, 281
  scope, 234
parameter, see parameter
parameter list, 2, 27, 202, 204
prototype, 207, 251
recursive, 227
  variadic template, 701
  scope, 204
  viable, 252
  viable function, 243
  virtual, see virtual function
function, 577, 576–579, 590
  and pointer to member, 842
  definition, 577
  desk calculator, 577
function call
  ambiguous, 234, 245, 251
  default argument, 236
  execution flow, 203
  overhead, 238
  through pointer to function, 248
  through pointer to member, 839
  to overloaded operator, 553
  to overloaded postfix operator, 568
function matching, 233, 251
  = delete, 508
  argument, conversion, 234
  candidate function, 243
  overloaded operator, 587
  const arguments, 246
  conversion, class type, 583–587
  conversion operator, 585, 586
  conversion rank, 245
  class type conversions, 586
  default argument, 243
  enumeration, 835
  function template, 694–699
  specialization, 708
  integral promotions, 246
  member function, 273
  multiple parameters, 244
  namespace, 800
  overloaded operator, 587–589
  prefers more specialized function, 695
  rvalue reference, 539
  variadic template, 702
  viable function, 243
function object, 571, 590
  argument to algorithms, 572
  arithmetic operators, 574
  is callable object, 571
function parameter, see parameter
function parameter pack, 700
  expansion, 703
  pattern, 704
function pointer, 247–250
  callable object, 388
  definition, 247
  exception specification, 779, 781
  function template instantiation, 686
  overloaded function, 248
  parameter, 249
  return type, 204, 249
    using decltype, 250
  template argument deduction, 686
  type alias declaration, 249
typedef, 249
function table, 577, 577, 590, 840
function template, 652, 713
see also template parameter
see also template argument deduction
see also instantiation
argument conversion, 680
array function parameters, 654
candidate function, 695
callable object, 388
declaration, 669
default template argument, 670
template argument deduction, 683
type alias declaration, 249
string literal version, 654
constexpr, 655
declaration, 669
default template argument, 670
error detection, 657
explicit instantiation, 675, 675–676
explicit template argument, 682
  compare, 683
function matching, 694–699
inline function, 655
nontype parameter, 654
overloaded function, 694–699
parameter pack, 713
specialization, 707, 714
  compare, 706
function matching, 708  
is an instantiation, 708  
namespace, 788  
scope, 708  
trailing return type, 684  
type-dependent code, 658  
function try block, 778, 817  
functional header, 397, 399, 400, 575, 577, 843  

G  
g++, 5  
gcount, istream, 763  
generate, 874  
generate_n, 874  
generic algorithms, see algorithms  
generic programming, 108  
type-independent code, 655  
get  
istream, 761  
multi-byte version, istream, 762  
returns int, istream, 762, 764  
get<n>, 719, 770  
getline, 87, 131  
istream, 762  
istringstream, 321  
TextQuery constructor, 488  
global function  
operator delete, 863  
operator new, 863  
global namespace, 789, 817  
:: (scope operator), 789, 818  
global scope, 48, 80  
global variable, lifetime, 204  
GNU compiler, 5  
good, 313  
goto statement, 192, 200  
grade clusters program, 103  
greater<T>, 575  
greater_equal<T>, 575  

H  
.h file header, 19  
handler, see catch  
has-a relationship, 637  
hash<key_type>, 445, 447  
override, 446  
specialization, 709, 788  
compatible with == (equality), 710  
hash function, 443, 447  
HasPtr  
reference counted, 514–516  
copy assignment, 515  
destructor, 515  
valuelike, 512  
copy assignment, 512  
move assignment, 540  
move constructor, 540  
swap, 516  
header, 6, 27  
iostream, 27  
C library, 91  
const and constexpr, 76  
default argument, 238  
function declaration, 207  
.h file, 19  
#include, 6, 21  
inline function, 240  
inline member function definition, 273  
namespace members, 786  
standard, 6  
table of library names, 866  
template definition, 656  
template specialization, 708  
user-defined, 21, 76–77, 207, 240  
using declaration, 83  
Sales_data.h, 76  
Sales_item.h, 19  
algorithm, 376  
array, 329  
bitset, 723  
cassert, 241  
cctype, 91  
cmath, 751, 757  
cstddef, 116, 120  
cstdio, 762  
cstdlib, 54, 227, 778, 823  
cstring, 122  
ctime, 749  
deque, 329  
exception, 197  
forward_list, 329  
fstream, 310, 316  
functional, 397, 399, 400, 575, 577, 843  
initializer_list, 220  
iomanip, 756  
iostream, 6, 310  
iterator, 119, 382, 401
index, 329
map, 420
memory, 450, 451, 481, 483
new, 197, 460, 478, 821
numeric, 376, 881
queue, 371
random, 745
regex, 728
set, 420
sstream, 310, 321
stack, 370
stdexcept, 194, 197
string, 74, 76, 84
tuple, 718
type_info, 197
type_traits, 684
typeinfo, 826, 827, 831
unordered_map, 420
unordered_set, 420
utility, 426, 530, 533, 694
vector, 96, 329
header guard, 77, 79
preprocessor, 77
heap, 450, 491
hex, manipulator, 754
hexadecimal
escape sequence (\Xnnn), 39
literal (0Xnum or 0xnum), 38
hexfloat manipulator, 757
high-order bits, 723, 770

I
i before e, program, 729
version 2, 734
IDE, 3
identifier, 46, 79
reserved, 46
_if algorithms, 414
_if statement, 17, 27, 175, 175–178, 200
compared to switch, 178
condition, 18, 175
dangling else, 177
else branch, 18, 175, 200
ifstream, 311, 316–320, 324
see also istream
close, 318
file marker, 765
file mode, 319
initialization, 317
off_type, 766
open, 318
pos_type, 766
random access, 765
random IO program, 766
seek and tell, 763–768
ignore, istream, 763
implementation, 254, 254, 306
in (file mode), 319
in scope, 49, 79
in-class initializer, 73, 73, 79, 263, 265, 274
#include
standard header, 6, 21
user-defined header, 21
includes, 880
incomplete type, 279, 306
can’t be base class, 600
not in exception declaration, 775
restrictions on use, 279
incr, StrBlobPtr, 475
increment operators, 147–149
indentation, 19, 177
index, 94, 131
see also [ ] (subscript)
indirect base class, 600, 650
inferred return type, lambda expression, 396
inheritance, 650
and container, 630
copy control, 623–629
friend, 614
hierarchy, 592, 600
interface class, 637
IO classes, 311, 324
name collisions, 618
private, 612, 650
protected, 612, 650
public, 612, 650
vs. composition, 637
inherited, constructor, 628
initialization
aggregate class, 298
array, 114
associative container, 423, 424
bitset, 723–725
C-style string, 122
class type objects, 73, 262
class
static data member, 302
class type object, 262
data member, 289
multiple key container, 433
dynamic allocation object, 459
insert iterator, 382, 401, 402, 418
backward inserter, 402
front inserter, 402
inserter, 402
insertion, 402, 418
compared to front inserter, 402
instantiation, 96, 131, 653, 656, 713
Blob, 660
class template, 660
member function, 663
declaration, 713
definition, 713
error detection, 657
explicit, 675–676
function template from function pointer, 686
member template, 674
static member, 667
int, 33
literal, 38
integral
constant expression, 65
promotion, 134, 160, 169
function matching, 246
type, 32, 79
integrated development environment, 3
interface, 254, 306
internal, manipulator, 759
interval, left-inclusive, 373
invalid pointer, 52
invalidated iterator
and container operations, 354
undefined behavior, 353
invalidates iterator
assign, 338
erase, 349
resize, 352
IO
formatted, 761, 769
unformatted, 761, 770
IO classes
condition state, 312, 324
inheritance, 324
IO stream, see stream
iomanip header, 756
iostate, 312
machine-dependent, 313
object, 59
copy, 84, 131, 497, 497–499, 549
default, 43, 293
direct, 84, 131
dynamically allocated object, 459
exception, 197
istream_iterator, 405
list, see list initialization
lvalue reference, 532
multidimensional array, 126
new [], 477
ostream_iterator, 405
pair, 426
parameter, 203, 208
pointer, 52–54
to const, 62
queue, 369
reference, 51
data member, 289
to const, 61
return value, 224
rvalue reference, 532
sequential container, 334–337
shared_ptr, 464
stack, 369
string, 84–85, 360–361
string streams, 321
tuple, 718
unique_ptr, 470
value, 98, 132, 293
variable, 42, 43, 79
vector, 97–101
vs. assignment, 42, 288
weak_ptr, 473
initializer_list, 220, 220–222, 252
= (assignment), 563
constructor, 662
header, 220
inline function, 238, 252
and header, 240
function template, 655
member function, 257, 273
and header, 273
inline namespace, 790, 817
inner scope, 48, 79
inner_product, 882
inplace_merge, 875
input, standard, 6
input iterator, 411, 418
insert
associative container, 432
iostream, 5
  file marker, 765
  header, 6, 27, 310
  off_type, 766
  pos_type, 766
  random access, 765
  random IO program, 766
  seek and tell, 763–768
  virtual base class, 810
iota, 882
is-a relationship, 637
is_partitioned, 876
is_permutation, 879
is_sorted, 877
is_sorted_until, 877
isalnum, 92
isalpha, 92
isbn
  Sales_data, 257
  Sales_item, 23
ISBN, 2
isbn_mismatch, 783
iscntrl, 92
isdigit, 92
isgraph, 92
islower, 92
isprint, 92
ispunct, 92
isShorter program, 211
isspace, 92
istream, 5, 27, 311
  see also manipulator
    >> (input operator), 8
      precedence and associativity, 155
    as condition, 15
    chained input, 8
    condition state, 312
    conversion, 162
    explicit conversion to bool, 583
    file marker, 765
    flushing input buffer, 314
    format state, 753
    gcount, 763
    get, 761
      multi-byte version, 762
      returns int, 762, 764
    getline, 87, 321, 762
    ignore, 763
    no copy or assign, 311
    off_type, 766
    peek, 761
pos_type, 766
put, 761
putback, 761
random access, 765
random IO program, 766
read, 763
seek and tell, 763–768
unformatted IO, 761
  multi-byte, 763
  single-byte, 761
unget, 761
istream_iterator, 403, 418
  >> (input operator), 403
    algorithms, 404
    initialization, 405
    off-the-end iterator, 403
    operations, 404
    type requirements, 406
istringstream, 311, 321, 321–323
  see also istream
    word per line processing, 442
    file marker, 765
    getline, 321
    initialization, 321
    off_type, 766
    phone number program, 321
    pos_type, 766
    random access, 765
    random IO program, 766
    seek and tell, 763–768
    TextQuery constructor, 488
isupper, 92
isxdigit, 92
iter_swap, 875
iterator, 106, 106–112, 131
  ++ (increment), 107, 132
  -- (decrement), 107
  * (dereference), 107
  += (compound assignment), 111
  + (addition), 111
  - (subtraction), 111
  == (equality), 106, 107
  != (inequality), 106, 107
  algorithm type independence, 377
  arithmetic, 111, 131
    compared to reverse iterator, 409
    destination, 413
    insert, 401, 418
    move, 401, 418, 543
      uninitialized_copy, 543
    off-the-beginning
before_begin, 351
forward_list, 351
off-the-end, 106, 132, 373
istream_iterator, 403
parameter, 216
regex, 734
relational operators, 111
reverse, 401, 407–409, 418
stream, 401, 403–406, 418
used as destination, 382

iterator
compared to reverse_iterator, 408
container, 108, 332
header, 119, 382, 401
set iterators are const, 429
iterator category, 410, 410–412, 418
bidirectional iterator, 412, 417
forward iterator, 411, 417
input iterator, 411, 418
output iterator, 411, 418
random-access iterator, 412, 418
iterator range, 331, 331–332, 373
algorithms, 376
as initializer of container, 335
container erase member, 349
container insert member, 344
left-inclusive, 331
off-the-end, 331

type checking, 46
types define behavior, 3
use concise expressions, 149
key_type
associative container, 428, 447
requirements
ordered container, 425
unordered container, 445
keyword table, 47
Koenig lookup, 797

L
L’c’ (wchar_t literal), 38
label
case, 179, 199
statement, 192
labeled statement, 192, 200
lambda expression, 388, 418
arguments, 389
biggies program, 391
reference capture, 393
capture list, 388, 417
capture by reference, 393
capture by value, 390, 392
implicit capture, 394
inferred return type, 389, 396
mutable, 395
parameters, 389
passed to find_if, 390
passed to stable_sort, 389
synthesized class type, 572–574
trailing return type, 396
left, manipulated, 758
left-inclusive interval, 331, 373
length_error, 197
less<T>, 575
less_equal<T>, 575
letter grade, program, 175
lexicographical_compare, 881
library function objects, 574
as arguments to algorithms, 575
library names to header table, 866
library type, 5, 27, 82
lifetime, 204, 252
compared to scope, 204
dynamically allocated objects, 450, 461
global variable, 204
local variable, 204
parameter, 205
linkage directive, 858, 863
C++ to C, 860
compound, 858
overloaded function, 860
parameter or return type, 859
pointer to function, 859
return type, 859
single, 858
linker, 208, 252
template errors at link time, 657
list, 373
see also container
see also sequential container
bidirectional iterator, 412
header, 329
initialization, 334–337
list initialization, 336
merge, 415
overview, 327
remove, 415
remove_if, 415
reverse, 415
splice, 416
unique, 415
value initialization, 336
list initialization, 43, 79
= (assignment), 145
array, 337
associative container, 423
container, 336
dynamically allocated, object, 459
pair, 427, 431, 527
preferred, 99
prevents narrowing, 43
return value, 226, 427, 527
sequential container, 336
vector, 98
literal, 38, 38–41, 79
bool, 41
  in condition, 143
char, 39
decimal, 38
double (numEnum or numenum), 38
float (numF or numf), 41
floating-point, 38
hexadecimal (0Xnum or 0xnum), 38
int, 38
long (numL or num1), 38
long double (ddd.dddL or ddd.ddd1), 41
long long (numLL or num11), 38
octal (0num), 38
string, 7, 28, 39
unsigned (numU or numu), 41
wchar_t, 40
literal type, 66
class type, 299
local class, 852, 863
  access control, 853
  name lookup, 853
  nested class in, 854
  restrictions, 852
local scope, see block scope
local static object, 205, 252
local variable, 204, 252
  destroyed during exception handling, 467, 773
  destructor, 502
  lifetime, 204
  pointer, return value, 225
  reference, return value, 225
  return statement, 224
lock, weak_ptr, 473
logic_error, 197
logical operators, 141, 142
  condition, 141
  function object, 574
logical_and<T>, 575
logical_not<T>, 575
logical_or<T>, 575
long, 33
  literal (numL or num1), 38
long double, 33
  literal (ddd.dddL or ddd.ddd1), 41
long long, 33
  literal (numLL or num11), 38
lookup, name, see name lookup
low-level const, 64, 79
  argument and parameter, 213
  conversion from, 163
  conversion to, 162
  overloaded function, 232
  template argument deduction, 693
low-order bits, 723, 770
lower_bound
algorithm, 873
ordered container, 438
lround, 751
lvalue, 135, 169
cast to rvalue reference, 691
copy initialization, uses copy constructor, 539
dcltype, 135
reference collapsing rule, 688
result
  - (arrow operator), 150
  ++ (increment) prefix, 148
  -- (decrement) prefix, 148
  * (dereference), 135
  [ ] (subscript), 135
  = (assignment), 145
  , (comma operator), 158
  ?: (conditional operator), 151
  cast, 163
dcltype, 71
  function reference return type, 226
variable, 533
lvalue reference, see also reference, 532, 549
collapsing rule, 688
  compared to rvalue reference, 533
  function matching, 539
  initialization, 532
  member function, 546
    overloaded, 547
move, 533
template argument deduction, 687

M
machine-dependent
  bit-field layout, 854
  char representation, 34
  end-of-file character, 15
  enum representation, 835
  iostate, 313
  linkage directive language, 861
  nonzero return from main, 227
  random IO, 763
  reinterpret_cast, 164
  return from exception what, 198
  signed out-of-range value, 35
  signed types and bitwise operators, 153
  size of arithmetic types, 32
  terminate function, 196
  type_info members, 831
  vector, memory management, 355
  volatile implementation, 856
main, 2, 27
  not recursive, 228
  parameters, 218
  return type, 2
  return value, 2–4, 227
make_move_iterator, 543
make_pair, 428
make_plural program, 224
make_shared, 451
make_tuple, 718
malloc library function, 823, 863
manipulator, 7, 27, 753, 770
  boolalpha, 754
  change format state, 753
dec, 754
defaultfloat, 757
endl, 314
ends, 315
fixed, 757
flush, 315
hex, 754
hexfloat, 757
internal, 759
left, 758
nboolalpha, 754
nshowbase, 755
nshowpoint, 758
noskipws, 760
nouppercase, 755
oct, 754
right, 758
scientific, 757
setfill, 759
setprecision, 756
setw, 758
showbase, 755
showpoint, 758
skipws, 760
unitbuf, 315
uppercase, 755
map, 420, 447
  see also ordered container
  * (dereference), 429
  [ ] (subscript), 435, 448
    adds element, 435
  at, 435
definition, 423
header, 420
insert, 431
key_type requirements, 425
list initialization, 423
lower_bound, 438
map, initialization, 424
TextQuery class, 485
upper_bound, 438
word_count program, 421
mapped_type, associative container, 428, 448

match
  best, 251
  no, 252
match_flag_type, regex_constants, 743

max, 881
max_element, 881
mem_fn, 843, 863
  generates callable, 843
member, see class data member
member access operators, 150
member function, 23, 27, 306
  as friend, 281
  base member hidden by derived, 619
class template
  defined outside class body, 661
  instantiation, 663
cost, 258, 305
  () (call operator), 573
  reference return, 276
declared but not defined, 509
defined outside class, 259
definition, 256–260
  :: (scope operator), 259
  name lookup, 285
  parameter list, 282
  return type, 283
explicitly inline, 273
function matching, 273
implicit this parameter, 257
implicitly inline, 257
inline and header, 273
move-enabled, 545
name lookup, 287
overloaded, 273
  on const, 276
  on lvalue or rvalue reference, 547
overloaded operator, 500, 552
reference qualified, 546, 550
returning *this, 260, 275
rvalue reference parameters, 544
scope, 282
template, see member template
member template, 672, 714
Blob, iterator constructor, 673
DebugDelete, 673
declaration, 673
defined outside class body, 674
instantiation, 674
template parameters, 673, 674
memberwise
  copy assignment, 500
copy constructor, 497
copy control, 267, 550
destruction is implicit, 503
move assignment, 538
move constructor, 538
memory
  see also dynamically allocated
  exhaustion, 460
  leak, 462
memory header, 450, 451, 481, 483
merge, 874
  list and forward_list, 415
Message, 519–524
  add_to_Folder, 522
  class definition, 521
copy assignment, 523
copy constructor, 522
design, 520
destructor, 522
move assignment, 542
move constructor, 542
move_Folders, 542
  remove_from_Folders, 523
method, see member function
Microsoft compiler, 5
min, 881
min_element, 881
minmax, 881
minus<T>, 575
mismatch, 872
mode, file, 324
modulus<T>, 575
move, 530, 533, 874
  argument-dependent lookup, 798
  binds rvalue reference to lvalue, 533
  explained, 690–692
  inherently dangerous, 544
  Message, move operations, 541
  moved from object has unspecified
  value, 533
  reference collapsing rule, 691
  StrVec reallocate, 530
  remove_reference, 691
move assignment, 536, 550
  copy and swap, 540
derived class, 626
HasPtr, valuelike, 540
memberwise, 538
Message, 542
moved-from object destructible, 537
noexcept, 535
rule of three/five, virtual destructor exception, 622
self-assignment, 537
StrVec, 536
synthesized
  deleted function, 538, 624
derived class, 623
multiple inheritance, 805
sometimes omitted, 538
move constructor, 529, 534, 534–536, 550
and copy initialization, 541
derived class, 626
HasPtr, valuelike, 540
memberwise, 538
Message, 542
moved-from object destructible, 534, 537
noexcept, 535
rule of three/five, virtual destructor exception, 622
string, 529
StrVec, 535
synthesized
  deleted function, 624
derived class, 623
multiple inheritance, 805
sometimes omitted, 538
move iterator, 401, 418, 543, 550
make_move_iterator, 543
StrVec, reallocate, 543
uninitialized_copy, 543
move operations, 531–548
  function matching, 539
move, 533
noexcept, 535
rvalue references, 532
valid but unspecified, 537
move_backward, 875
move_Folders, Message, 542
multidimensional array, 125–130
  [ ] (subscript), 127
argument and parameter, 218
begin, 129
conversion to pointer, 128
definition, 126
data member, 274
definition, 129
end, 129
initialization, 126
pointer, 128
range for statement and, 128
multimap, 448
  see also ordered container
* (dereference), 429
definition, 423
has no subscript operator, 435
insert, 431, 433
key_type requirements, 425
list initialization, 423
lower_bound, 438
map, initialization, 424
upper_bound, 438
multiple inheritance, 802, 817
  see also virtual base class
= (assignment), 805
ambiguous conversion, 806
ambiguous names, 808
avoiding ambiguities, 809
class derivation list, 803
conversion, 805
copy control, 805
name lookup, 807
object composition, 803
order of initialization, 804
scope, 807
virtual function, 807
multiplies<T>, 575
multiset, 448
  see also ordered container
insert, 433
iterator, 429
key_type requirements, 425
list initialization, 423
lower_bound, 438
override comparison
  Basket class, 631
  using compareIsbn, 426
upper_bound, 438
used in Basket, 632
mutable
mutable
mutable
N
N

\n (newline character), 39
name lookup, 283, 306
  :: (scope operator), overrides, 286
argument-dependent lookup, 797
before type checking, 619
multiple inheritance, 809
block scope, 48
class, 284
class member
declaration, 284
definition, 285, 287
function, 284
depends on static type, 617, 619
multiple inheritance, 806
derived class, 617
name collisions, 618
local class, 853
multiple inheritance, 807
ambiguous names, 808
namespace, 796
nested class, 846
overloaded virtual functions, 621
templates, 657
type checking, 235
virtual base class, 812
named cast, 162
const_cast, 163, 163
dynamic_cast, 163, 825
reinterpret_cast, 163, 164
static_cast, 163, 163
namespace, 7, 27, 785, 817
alias, 792, 817
argument-dependent lookup, 797
candidate function, 800
cplusplus_primer, 787
definition, 785
design, 786
discontiguous definition, 786
friend declaration scope, 799
function matching, 800
global, 789, 817
inline, 790, 817
member, 786
member definition, 788
outside namespace, 788
name lookup, 796
nested, 789
overloaded function, 800
placeholders, 399
scope, 785–790
std, 7
template specialization, 709, 788
unnamed, 791, 818
local to file, 791
replace file static, 791
namespace pollution, 785, 817
narrowing conversion, 43
NDEBUG, 241
negate<T>, 575
nested class, 843, 863
access control, 844
class defined outside enclosing class, 845
constructor, QueryResult, 845
in local class, 854
member defined outside class body, 845
name lookup, 846
QueryResult, 844
relationship to enclosing class, 844, 846
scope, 844
static member, 845
nested namespace, 789
nested type, see nested class
new, 458, 458–460, 491
execution flow, 820
failure, 460
header, 197, 460, 478, 821
initialization, 458
placement, 460, 491, 824, 863
union with class type member, 851
shared_ptr, 464
unique_ptr, 470
with auto, 459
new[], 477, 477–478
initialization, 477
returns pointer to an element, 477
value initialization, 478
newline (\n), character, 39
next_permutation, 879
no match, 234, 252
see also function matching
nboolalpha, manipulator, 754
NoDefault, 293
noexcept
exception specification, 779, 817
argument, 779–781
violation, 779
move operations, 535
operator, 780, 817
nonconst reference, see reference
none, bitset, 726
none_of, 871
nonportable, 36, 863
nonprintable character, 39, 79
nonthrowing function, 779, 818
nontype parameter, 654, 714
Index

compare, 654
must be constant expression, 655
type requirements, 655
normal_distribution, 751
noshowbase, manipulator, 755
noshowpoint, manipulator, 758
nospitws, manipulator, 760
not_equal_to<T>, 575
NotQuery, 637
class definition, 642
eval function, 647
nouppercase, manipulator, 755
nth_element, 877
NULL, 54
null (\0), character, 39
null pointer, 53, 79
delete of, 461
null statement, 172, 200
null-terminated character string, see C-style string
nullptr, 54, 79
numeric header, 376, 881
numeric conversion, to and from string, 367
numeric literal
float (numF or numf), 41
long (numL or numl), 41
long double (ddd.dddL or ddd.ddd1), 41
long long(numLL or numll), 41
unsigned (numU or numu), 41

class type object, 289
derived class object, 598, 623
multiple inheritance, 804
virtual base classes, 814
object code, 252
object file, 208, 252
object-oriented programming, 650
oct, manipulator, 754
octal, literal (onum), 38
octal escape sequence (\num), 39
off-the-beginning iterator, 351, 373
before_begin, 351
forward_list, 351
off-the-end iterator, 106, 132, 373
iterator range, 331
pointer, 118
ofstream, 311, 316–320, 324
see also ostream
close, 318
file marker, 765
file mode, 319
initialization, 317
off_type, 766
open, 318
pos_type, 766
random access, 765
random IO program, 766
seek and tell, 763–768
old-style, cast, 164
open, file stream, 318
operand, 134, 169
conversion, 159
operator, 134, 169
operator alternative name, 46
operator delete
class member, 822
global function, 820, 863
operator delete[]
class member, 822
compared to deallocate, 823
global function, 820
operator new
class member, 822
global function, 820, 863
operator new[]
class member, 822
compared to allocate, 823
global function, 820
operator overloading, see overloaded operator
operators
arithmetic, 139
assignment, 12, 144–147
binary, 134, 168
bitwise, 152–156
    bitset, 725
comma (,), 157
compound assignment, 12
conditional (? :), 151
decrement, 147–149
equality, 18, 141
increment, 12, 147–149
input, 8
iterator
    addition and subtraction, 111
    arrow, 110
dereference, 107
equality, 106, 108
    increment and decrement, 107
    relational, 111
logical, 141
member access, 150
noexcept, 780
output, 7
overloaded, arithmetic, 560
pointer
    addition and subtraction, 119
    equality, 120
    increment and decrement, 118
    relational, 120, 123
    subscript, 121
    relational, 12, 141, 143
Sales_data
    += (compound assignment), 564
    + (addition), 560
    == (equality), 561
    != (inequality), 561
    >> (input operator), 558
    <= (output operator), 557
    Sales_item, 20
    scope, 82
    sizeof, 156
    sizeof..., 700
    string
        addition, 89
        equality and relational, 88
        IO, 85
        subscript, 93–95
    subscript, 116
typeid, 826, 864
unary, 134, 169

vector
    equality and relational, 102
    subscript, 103–105
options to main, 218
order of destruction
    class type object, 502
derived class object, 627
multiple inheritance, 805
virtual base classes, 815
order of evaluation, 134, 169
    && (logical AND), 138
    || (logical OR), 138
    , (comma operator), 138
    ?: (conditional operator), 138
expression, 137
pitfalls, 149
order of initialization
    class type object, 289
derived class object, 598
multiple base classes, 816
multiple inheritance, 804
virtual base classes, 814
ordered container
    see also container
    see also associative container
    key_type requirements, 425
    lower_bound, 438
    override default comparison, 425
    upper_bound, 438
ordering, strict weak, 425, 448
OrQuery, 637
class definition, 644
eval function, 645
ostream, 5, 27, 311
    see also manipulator
    << (output operator), 7
        precedence and associativity, 155
chained output, 7
condition state, 312
explicit conversion to bool, 583
file marker, 765
floating-point notation, 757
flushing output buffer, 314
format state, 753
no copy or assign, 311
not flushed if program crashes, 315
off_type, 766
output format, floating-point, 755
pos_type, 766
precision member, 766
random access member, 765
random IO program, 766
seek and tell, 763–768
tie member, 315
virtual base class, 810
write, 763

ostream_iterator, 403, 418
<< (output operator), 405
algorithms, 404
initialization, 405
operations, 405
type requirements, 406

ostringstream, 311, 321, 321–323

ostream, 403, 418
file marker, 765
initialization, 321
off_type, 766
phone number program, 323
pos_type, 766
random access, 765
random IO program, 766
seek and tell, 763–768
str, 323

out (file mode), 319
out-of-range value, signed, 35
out_of_range, 197
at function, 348
out_of__stock, 783
outer scope, 48, 79
output, standard, 6
output iterator, 411, 418
overflow, 140
overflow_error, 197
overhead, function call, 238
overload resolution, see function matching
overloaded function, 230, 230–235, 252
see also function matching
as friend, 281
generated to redeclaration, 231
generated to template specialization, 708
const parameters, 232
constructor, 262
definition, function template, 694–699
linkage directive, 860
member function, 273
candidate function, 587
const, 276
move-enabled, 545
reference qualified, 547
virtual, 621
move-enabled, 545
namespace, 800
pointer to, 248
scope, 234
derived hides base, 619
using declaration, 800
in derived class, 621
using directive, 801
overloaded operator, 135, 169, 500, 550, 552, 590
++ (increment), 566–568
- - (decrement), 566–568
* (dereference), 569

StrBlobPtr, 569
& (address-of), 554
-> (arrow operator), 569
StrBlobPtr, 569
[ ] (subscript), 564
StrVec, 565
() (call operator), 571
absInt, 571
PrintString, 571
= (assignment), 500, 563

StrVec initializer_list, 563
+= (compound assignment), 555, 560
Sales_data, 564
+ (addition), Sales_data, 560
== (equality), 561
Sales_data, 561
!= (inequality), 562
Sales_data, 561
< (less-than), strict weak ordering, 562
>> (input operator), 558–559
Sales_data, 558
<< (output operator), 557–558
Sales_data, 557
&& (logical AND), 554
|| (logical OR), 554
& (bitwise AND), Query, 644
| (bitwise OR), Query, 644
~ (bitwise NOT), Query, 643
, (comma operator), 554
ambiguous, 588
arithmetic operators, 560
associativity, 553
binary operators, 552
consistency between relational and equality operators, 562
definition, 500, 552
design, 554–556
equality operators, 561
explicit call to, 553
postfix operators, 568
function matching, 587–589
member function, 500, 552
member vs. nonmember function, 552, 555
precedence, 553
relational operators, 562
requires class-type parameter, 552
short-circuit evaluation lost, 553
unary operators, 552
override, virtual function, 595, 650
override specifier, 593, 596, 606

P
pair, 426, 448
default initialization, 427
definition, 426
initialization, 426
list initialization, 427, 431, 527
make_pair, 428
map, * (dereference), 429
operations, 427
public data members, 427
return value, 527
Panda, 803
parameter, 202, 208, 252
array, 214–219
  buffer overflow, 215
to pointer conversion, 214
C-style string, 216
cstr, 212
copy constructor, 496
ellipsis, 222
forwarding, 693
function pointer, linkage directive, 859
implicit this, 257
initialization, 203, 208
iterator, 216
lifetime, 205
low-level cstr, 213
main function, 218
multidimensional array, 218
nonreference, 209
  uses copy constructor, 498
  uses move constructor, 539
pass by reference, 210, 252
pass by value, 209, 252
passing, 208–212
pointer, 209, 214
pointer to cstr, 246
pointer to array, 218
pointer to function, 249
  linkage directive, 859
reference, 210–214
to cstr, 213, 246
to array, 217
reference to cstr, 246
template, see template parameter
top-level cstr, 212
parameter list
  function, 2, 27, 202
template, 653, 714
parameter pack, 714
  expansion, 702, 702–704, 714
  function template, 713
  sizeof..., 700
  variadic template, 699
parentheses, override precedence, 136
partial_sort, 877
partial_sort_copy, 877
partial_sum, 882
partition, 876
partition_copy, 876
partition_point, 876
pass by reference, 208, 210, 252
pass by value, 209, 252
  uses copy constructor, 498
  uses move constructor, 539
pattern, 702, 714
  function parameter pack, 704
  regular expression, phone number, 739
  template parameter pack, 703
peek, istream, 761
PersonInfo, 321
phone number, regular expression
  program, 738
  reformat program, 742
  valid, 740
pitfalls
dynamic memory, 462
order of evaluation, 149
self-assignment, 512
smart pointer, 469
using directive, 795
placeholders, 399
placement new, 460, 491, 824, 863
  union, class type member, 851
plus<T>, 575
pointer, 52, 52–58, 79
  ++ (increment), 118
  -- (decrement), 118
  * (dereference), 53
  [ ] (subscript), 121
  = (assignment), 55
  + (addition), 119
  - (subtraction), 119
  == (equality), 55, 120
  != (inequality), 55, 120
  arithmetic, 119, 132
  const, 63, 78
  const pointer to const, 63
  constexpr, 67
  conversion
    from array, 161
    to bool, 162
    to const, 62, 162
    to void*, 161
  dangling, 463, 491
  declaration style, 57
  definition, 52
  delete, 460
  derived-to-base conversion, 597
    under multiple inheritance, 805
  dynamic_cast, 825
  implicit this, 257, 306
  initialization, 52–54
  invalid, 52
  multidimensional array, 128
  null, 53, 79
  off-the-end, 118
  parameter, 209, 214
  relational operators, 123
  return type, 204
  return value, local variable, 225
  smart, 450, 491
  to const, 62
    and typedef, 68
  to array
    parameter, 218
    return type, 204
    return type declaration, 229
  to const, 79
    overloaded parameter, 232, 246
  to pointer, 58
  typeid operator, 828
  valid, 52
  volatile, 856
  pointer to function, 247–250

auto, 249
callable object, 388
decltype, 249
exception specification, 779, 781
explicit template argument, 686
function template instantiation, 686
linkage directive, 859
overloaded function, 248
parameter, 249
return type, 204, 249
  using decltype, 250
template argument deduction, 686
trailing return type, 250
type alias, 249
typedef, 249
pointer to member, 835, 863
  arrow (->*), 837
definition, 836
dot (.*), 837
function, 838
  and bind, 843
  and function, 842
  and mem_fn, 843
  not callable object, 842
  function call, 839
  function table, 840
  precedence, 838
polymorphism, 605, 650
pop
  priority_queue, 371
  queue, 371
  stack, 371
pop_back
  sequential container, 348
  StrBlob, 457
pop_front, sequential container, 348
portable, 854
precedence, 134, 136–137, 169
  = (assignment), 146
  ?: (conditional operator), 151
  assignment and relational operators, 146
dot and dereference, 150
increment and dereference, 148
of IO operator, 156
overloaded operator, 553
parentheses overrides, 136
pointer to member and call operator, 838
precedence table, 166
precision member, ostream, 756
predicate, 386, 418
   binary, 386, 417
   unary, 386, 418
prefix, smatch, 736
preprocessor, 76, 79
   #include, 7
   assert macro, 241, 251
   header guard, 77
   variable, 54, 79
prev_permutation, 879
print, Sales_data, 261
print program
   array parameter, 215
   array reference parameter, 217
   pointer and size parameters, 217
   pointer parameter, 216
   two pointer parameters, 216
   variadic template, 701
print_total
   explained, 604
   program, 593
PrintString, 571
   () (call operator), 571
priority_queue, 371, 373
   emplace, 371
   empty, 371
   equality and relational operators, 370
   initialization, 369
   pop, 371
   push, 371
   sequential container, 371
   size, 371
   swap, 371
   top, 371
private
   access specifier, 268, 306
   copy constructor and assignment, 509
   inheritance, 612, 650
program
   addition
      Sales_data, 74
      Sales_item, 24
      alternative_sum, 682
biggies, 391
binops desk calculator, 577
book from author version 1, 438
book from author version 2, 439
book from author version 3, 440
bookstore
   Sales_data, 255
   Sales_data using algorithms, 406
   Sales_item, 24
buildMap, 442
children’s story, 383–391
compare, 652
count_calls, 206
debug_rep
   additional nontemplate versions, 698
   general template version, 695
   nontemplate version, 697
   pointer template version, 696
elimDups, 383–391
error_msg, 221
fact, 202
factorial, 227
factory
   new, 461
   shared_ptr, 453
file extension, 730
   version 2, 738
find last word, 408
find_char, 211
findBook, 721
flip, 694
flip1, 692
flip2, 693
grade clusters, 103
grading
   bitset, 728
   bitwise operators, 154
i before e, 729
   version 2, 734
isShorter, 211
letter grade, 175
make_plural, 224
message handling, 519
phone number
   istringstream, 321
   ostringstream, 323
   reformat, 742
   regular expression version, 738
   valid, 740
print
   array parameter, 215
   array reference parameter, 217
   pointer and size parameters, 217
   pointer parameter, 216
   two pointer parameters, 216
   variadic template, 701
print_total, 593
Query, 635
class design, 636–639
random IO, 766
reset
  pointer parameters, 209
  reference parameters, 210
restricted word_count, 422
sum, 682
swap, 223
TextQuery, 486
design, 485
transform, 442
valid, 740
vector capacity, 357
vowel counting, 179
word_count
  map, 421
  unordered_map, 444
word_transform, 441
ZooAnimal, 802
promotion, see integral promotion
protected
  access specifier, 595, 611, 650
  inheritance, 612, 650
  member, 611
ptr_fun deprecated, 401
ptrdiff_t, 120, 132
public
  access specifier, 268, 306
  inheritance, 612, 650
pure virtual function, 609, 650
  Disc_quote, 609
  Query_base, 636
push
  priority_queue, 371
queue, 371
  empty, 371
match, 371
pop, 371
push, 371
Reversal, 371
push_back
  back_inserter, 382, 402
  sequential container, 100, 132, 342
  move-enabled, 545
StrVec, 527
  move-enabled, 545
push_front
  front_inserter, 402
  sequential container, 342
put, istream, 761
pushback, istream, 761
Q
Query, 638
<< (output operator), 641
& (bitwise AND), 638
definition, 644
| (bitwise OR), 638
definition, 644
~ (bitwise NOT), 638
definition, 643
classes, 636–639
definition, 640
interface class, 637
operations, 635
program, 635
recap, 640
Query_base, 636
  abstract base class, 636
definition, 639
  member function, 636
QueryResult, 485
class definition, 489
nested class, 849
  constructor, 845
print, 490
queue, 371, 373
  back, 371
equality and relational operators, 370
emplace, 371
empty, 371
header, 371
initialization, 369
pop, 371
push, 371
sequential container, 371
size, 371
swap, 371
Quote
class definition, 594
design, 592
R
Raccoon, virtual base class, 812
raise exception, see throw
rand function, drawbacks, 745
random header, 745
random IO, 765
  machine-dependent, 763
  program, 766
random-access iterator, 412, 418
random-number library, 745
  compared to rand function, 745
distribution types, 745, 770  
engine, 745, 770  
default_random_engine, 745  
max, min, 747  
retain state, 747  
seed, 748, 770  
generator, 746, 770  
range, 747  
random_shuffle, 878  
range for statement, 91, 132, 187, 187–189, 200  
can’t add elements, 101, 188  
multidimensional array, 128  
not with dynamic array, 477  
range_error, 197  
rbegin, container, 333, 407  
rdstate, stream, 313  
read  
istream, 763  
Sales_data, 261  
reallocate, StrVec, 530  
move iterator version, 543  
recursion loop, 228, 252, 608  
recursive function, 227, 252  
variadic template, 701  
ref, binds reference parameter, 400, 418  
refactoring, 611, 650  
reference, 50, 79  
seeing also lvalue reference  
seeing also rvalue reference  
auto deduces referred to type, 69  
collapsing rule, 688  
forward, 694  
lvalue arguments, 688  
move, 691  
rvalue reference parameters, 693  
const, seeing reference to const  
conversion  
not from const, 61  
to reference to const, 162  
data member, initialization, 289  
declaration style, 57  
decltype yields reference type, 71  
definition, 51  
derived-to-base conversion, 597  
under multiple inheritance, 805  
dynamic_cast operator, 826  
initialization, 51  
member function, 546  
parameter, 210–214  
bind, 400  
limitations, 214  
template argument deduction, 687–689  
remove_reference, 684  
return type, 224  
assignment operator, 500  
is lvalue, 226  
return value, local variable, 225  
to array parameter, 217  
reference, container, 333  
reference count, 452, 491, 514, 550  
copy assignment, 514  
copy constructor, 514  
design, 514  
destructor, 514  
HasPtr class, 514–516  
reference to const, 61, 80  
argument, 211  
initialization, 61  
parameter, 211, 213  
overloaded, 232, 246  
return type, 226  
regex, 728, 770  
error_type, 732  
header, 728  
regex_error, 732, 770  
syntax_option_type, 730  
regex_constants, 743  
match_flag_type, 743  
regex_error, 732, 770  
regex_match, 729, 770  
regex_replace, 742, 770  
format flags, 744  
format string, 742  
regex_search, 729, 730, 770  
regular expression library, 728, 770  
case sensitive, 730  
compiled at run time, 732  
ECMAScript, 730  
file extension program, 730  
i before e program, 729  
version 2, 734  
match data, 735–737  
pattern, 729  
phone number, valid, 740  
phone number pattern, 739  
phone number program, 738  
phone number reformat, program, 742  
regex iterators, 734  
search functions, 729
Index

smatch, provides context for a match, 735
subexpression, 738
  file extension program version 2, 738
types, 733
valid, program, 740
reinterpret_cast, 163, 164
  machine-dependent, 164
relational operators, 141, 143
  arithmetic conversion, 144
  container adaptor, 370
  container member, 340
  function object, 574
  iterator, 111
  overloaded operator, 562
  pointer, 120, 123
  Sales_data, 563
  string, 88
  tuple, 720
  vector, 102
release, unique_ptr, 470
remove, 878
  list and forward_list, 415
remove_copy, 878
remove_copy_if, 878
remove_from_Folders, Message, 523
remove_if, 878
  list and forward_list, 415
remove_pointer, 685
remove_reference, 684
  move, 691
rend, container, 333, 407
replace, 383, 875
  string, 362
  replace_copy, 383, 874
  replace_copy_if, 874
  replace_if, 875
reserve
  string, 356
  vector, 356
reserved identifiers, 46
reset
  bitset, 727
  shared_ptr, 466
  unique_ptr, 470
reset program
  pointer parameters, 209
  reference parameters, 210
resize
  invalidates iterator, 352
sequential container, 352
  value initialization, 352
restricted word_count program, 422
result, 134, 169
  * (dereference), lvalue, 135
  [ ] (subscript), lvalue, 135
  , (comma operator), lvalue, 158
  ?: (conditional operator), lvalue, 151
  cast, lvalue, 163
rethrow, 776
  exception object, 777
throw, 776, 818
return statement, 222, 222–228
  from main, 227
  implicit return from main, 223
  local variable, 224, 225
return type, 2, 27, 202, 204, 252
  array, 204
  array using decltype, 230
  function, 204
  function pointer, 249
  using decltype, 250
linkage directive, 859
main, 2
  member function, 283
  nonreference, 224
  copy initialized, 498
  pointer, 204
  pointer to function, 204
  reference, 224
  reference to const, 226
  reference yields lvalue, 226
trailing, 229, 252, 396, 684
virtual function, 606
void, 223
return value
  conversion, 223
  copy initialized, 498
  initialization, 224
list initialization, 226, 427, 527
local variable, pointer, 225
main, 2–4, 227
pair, 427, 527
reference, local variable, 225
*this, 260, 275
tuple, 721
type checking, 223
unique_ptr, 471
reverse, 878
  list and forward_list, 415
reverse iterator, 401, 407–409, 418
++ (increment), 407
-- (decrement), 407, 408
base, 409
  compared to iterator, 409
reverse_copy, 414, 878
reverse_copy_if, 414
reverse_iterator
  compared to iterator, 408
  container, 332, 407
rfind, string, 366
right, manipulator, 758
rotate, 878
rotate_copy, 878
rule of three/five, 505, 541
  virtual destructor exception, 622
run-time type identification, 825–831, 864
  compared to virtual functions, 829
dynamic_cast, 825, 825
  bad_cast, 826
  to pointer, 825
  to reference, 826
type-sensitive equality, 829
typeid, 826, 827
  returns type_info, 827
runtime binding, 594, 650
runtime_error, 194, 197
  initialization from string, 196
rvalue, 135, 169
  copy initialization, uses move constructor, 539
result
  ++ (increment) postfix, 148
  -- (decrement) postfix, 148
  function nonreference return type, 224
rvalue reference, 532, 550
  cast from lvalue, 691
collapsing rule, 688
compared to lvalue reference, 533
function matching, 539
initialization, 532
member function, 546
  overloaded, 547
move, 533
parameter
  forwarding, 693, 705
  member function, 544
preserves argument type information, 693
template argument deduction, 687
variable, 533

S
Sales_data
  compareIsbn, 387
  += (compound assignment), 564
  + (addition), 560
  == (equality), 561
  != (inequality), 561
  >> (input operator), 558
  << (output operator), 557
add, 261
addition program, 74
avg_price, 259
bookstore program, 255
  using algorithms, 406
class definition, 72, 268
combine, 259
compareIsbn, 425
  with associative container, 426
constructors, 264–266
converting constructor, 295
default constructor, 262
exception classes, 783
exception version
  += (compound assignment), 784
  + (addition), 784
explicit constructor, 296
isbn, 257
operations, 254
print, 261
read, 261
relational operators, 563
Sales_data.h header, 76
Sales_item, 20
  + (addition), 22
  >> (input operator), 21
  << (output operator), 21
addition program, 21, 23
bookstore program, 24
isbn, 23
operations, 20
Sales_item.h header, 19
scientific manipulator, 757
scope, 48, 80
  base class, 617
  block, 48, 80, 173
class, 73, 282, 282–287, 305
  static member, 302
  compared to object lifetime, 204
derived class, 617
friend, 270, 281
function, 204
global, 48, 80
inheritance, 617–621
member function, 282
parameters and return type, 283
multiple inheritance, 807
name collisions, using directive, 795
namespace, 785–790
nested class, 844
overloaded function, 234
statement, 174
template parameter, 668
template specialization, 708
using directive, 794
virtual function, 620
scoped enumeration, 832, 864
enum class, 832
Screen, 271
pos member, 272
concatenating operations, 275
do_display, 276
friends, 279
get, 273, 282
get_cursor, 283
Menu function table, 840
move, 841
move members, 275
set, 275
search, 872
search_n, 871
seed, random-number engine, 748
seekp, seekg, 763–768
self-assignment
copy and swap assignment, 519
copy assignment, 512
explicit check, 542
HasPtr
reference counted, 515
valuelike, 512
Message, 523
move assignment, 537
pitfalls, 512
StrVec, 528
semicolon (;), 3
class definition, 73
null statement, 172
separate compilation, 44, 80, 252
compiler options, 207
declaration vs. definition, 44
templates, 656
sequential container, 326, 373
array, 326
deque, 326
forward_list, 326
initialization, 334–337
list, 326
list initialization, 336
members
assign, 338
back, 346
clear, 350
duplicate, 283
erase, 349
front, 346
insert, 343
pop_back, 348
pop_front, 348
push_back, 132
push_back, 100, 342, 545
push_front, 342
resize, 352
value_type, 333
performance characteristics, 327
priority_queue, 371
queue, 371
stack, 370
value initialization, 336
vector, 326
set, 420, 448
see also ordered container
bitset, 727
header, 420
insert, 431
iterator, 429
key_type requirements, 425
list initialization, 423
lower_bound, 438
TextQuery class, 485
upper_bound, 438
word_count program, 422
set_difference, 880
set_intersection, 647, 880
set_symmetric_difference, 880
set_union, 880
setfill, manipulator, 759
setprecision, manipulator, 756
setstate, stream, 313
setw, manipulator, 758
shared_ptr, 450, 450–457, 464–469, 491
* (dereference), 451
copy and assignment, 451
definition, 450
deleter, 469, 491
    bound at run time, 677
derived-to-base conversion, 630
destructor, 453
dynamically allocated array, 480
exception safety, 467
factory program, 453
initialization, 464
make_shared, 451
pitfalls, 469
reset, 466
StrBlob, 455
TextQuery class, 485
with new, 464
short, 33
short-circuit evaluation, 142, 169
    && (logical AND), 142
    || (logical OR), 142
    not in overloaded operator, 553
ShorterString, 573
    () (call operator), 573
shorterString, 224
showbase, manipulator, 755
showpoint, manipulator, 758
shrink_to_fit
    deque, 357
    string, 357
    vector, 357
shuffle, 878
signed, 34, 80
    char, 34
    conversion to unsigned, 34, 160
    out-of-range value, 35
signed type, 34
single-line (//), comment, 9, 26
size
    container, 88, 102, 132, 340
    priority_queue, 371
    queue, 371
    returns unsigned, 88
    stack, 371
    StrVec, 526
size_t, 116, 132, 727
    array subscript, 116
size_type, container, 88, 102, 132, 332
SizeComp, 573
    () (call operator), 573
sizeof, 156, 169
    array, 157
data member, 157
sizeof..., parameter pack, 700
skipws, manipulator, 760
sliced, 603, 650
SmallInt
    + (addition), 588
    conversion operator, 580
smart pointer, 450, 491
    exception safety, 467
    pitfalls, 469
smatch, 729, 733, 769, 770
    prefix, 736
    provide context for a match, 735
    suffix, 736
sort, 384, 876
source file, 4, 27
specialization, see template specialization
splice, list, 416
splice_after, forward_list, 416
sregex_iterator, 733, 770
    i before e program, 734
sstream
    file marker, 765
    header, 310, 321
    off_type, 766
    pos_type, 766
    random access, 765
    random IO program, 766
    seek and tell, 763–768
ssub_match, 733, 736, 770
    example, 740
stable_partition, 876
stable_sort, 387, 876
stack, 370, 373
    emplace, 371
    empty, 371
    equality and relational operators, 370
    header, 370
    initialization, 369
    pop, 371
    push, 371
    sequential container, 370
    size, 371
    swap, 371
    top, 371
stack unwinding, exception handling, 773, 818
standard error, 6, 27
standard header, #include, 6, 21
standard input, 6, 27
standard library, 6, 27
Index

standard output, 6, 27
statement, 2, 27  
  block, see block
  break, 190, 199
  compound, 173, 199
  continue, 191, 199
  do while, 189, 200
  expression, 172, 200
  for, 13, 27, 185, 185–187, 200
  goto, 192, 200
  if, 17, 27, 175, 175–178, 200
  labeled, 192, 200
  null, 172, 200
  range for, 91, 187, 187–189, 200
  return, 222, 222–228
  scope, 174
  switch, 178, 178–182, 200
  while, 11, 28, 183, 183–185, 200
statement label, 192
static (file static), 792, 817
static member  
  Account, 301
  class template, 667
    accessed through an instantiation, 667
    definition, 667
  const data member, initialization, 302
  data member, 300
    definition, 302
  default argument, 304
  definition, 302
  inheritance, 599
  instantiation, 667
  member function, 301
  nested class, 845
  scope, 302
static object, local, 205, 252
static type, 601, 650  
  determines name lookup, 617, 619
    multiple inheritance, 806
static type checking, 46
static_cast, 163, 163  
  lvalue to rvalue, 691
std, 7, 28
std::forward, see forward
std::move, see move
stdexcept header, 194, 197
stod, 368
stof, 368
stoi, 368
stol, 368
stold, 368
stoll, 368
store, free, 450, 491
stoull, 368
str, string streams, 323
StrBlob, 456
  back, 457
  begin, 475
  check, 457
  constructor, 456
  end, 475
  front, 457
  pop_back, 457
  shared_ptr, 455
StrBlobPtr, 474
  ++ (increment), 566
  -- (decrement), 566
  * (dereference), 569
  -> (arrow operator), 569
  check, 474
  constructor, 474
  deref, 475
  incr, 475
  weak_ptr, 474
strcat, 123
strcmp, 123
strncpy, 123
stream
  as condition, 15, 162, 312
  clear, 313
  explicit conversion to bool, 583
  file marker, 765
  flushing buffer, 314
  format state, 753
  istream_iterator, 403
  iterator, 401, 403–406, 418
  type requirements, 406
  not flushed if program crashes, 315
  ostream_iterator, 403
  random IO, 765
  rdstate, 313
  setstate, 313
strict weak ordering, 425, 448
string, 80, 84–93, 132
  see also container
  see also sequential container
  see also iterator
[ ] (subscript), 93, 132, 347
+= (compound assignment), 89
+ (addition), 89
>> (input operator), 85, 132
>> (input operator) as condition, 86
<< (output operator), 85, 132
and string literal, 89–90
append, 362
assign, 362
at, 348
C-style string, 124
c_str, 124
capacity, 356
case sensitive, 365
compare, 366
concatenation, 89
default initialization, 44
difference_type, 112
equality and relational operators, 88
erase, 362
find, 364
find_first_not_of, 365
find_last_not_of, 366
find_last_of, 366
getline, 87, 321
header, 74, 76, 84
initialization, 84–85, 360–361
initialization from string literal, 84
insert, 362
move constructor, 529
numeric conversions, 367
random-access iterator, 412
replace, 362
reserve, 356
rfind, 366
substring range, 95
substr, 361
TextQuery class, 485
string literal, 7, 28, 39
see also C-style string
and string, 89–90
concatenation, 39
stringstream, 321, 321–323, 324
initialization, 321
strlen, 122
struct
see also class
default access specifier, 268
default inheritance specifier, 616
StrVec, 525
[ ] (subscript), 565
= (assignment), initializer_list, 563
alloc_n_copy, 527
begin, 526
capacity, 526
chk_n_alloc, 526
copy assignment, 528
copy constructor, 528
default constructor, 526
design, 525
destructor, 528
emplace_back, 704
end, 526
free, 528
memory allocation strategy, 525
move assignment, 536
move constructor, 535
push_back, 527
move-enabled, 545
reallocate, 530
move iterator version, 543
size, 526
subexpression, 770
substring range, 93
array, 116
string, 95
validating, 104
vector, 105
substr, string, 361
suffix, smatch, 736
sum, program, 682
swap, 516
array, 339
container, 339
container nonmember version, 339
copy and swap assignment operator, 518
priority_queue, 371
queue, 371
stack, 371
typical implementation, 517–518
swap program, 223
swap_ranges, 875
switch statement, 178, 178–182, 200
default case label, 181
break, 179–181, 190
compared to if, 178
execution flow, 180
variable definition, 182
syntax_option_type, regex, 730
synthesized
copy assignment, 500, 550
copy constructor, 497, 550
Index

copy control, 267
    as deleted function, 508
    as deleted in derived class, 624
Bulk_quote, 623
multiple inheritance, 805
virtual base class, 815
virtual base classes, 815
volatile, 857
default constructor, 263, 306
derived class, 623
    members of built-in type, 263
destructor, 503, 550
move operations
    deleted function, 538
    not always defined, 538

template
    see also class template
    see also function template
    see also instantiation
declaration, 669
link time errors, 657
overview, 652
parameter, see template parameter
parameter list, 714
template argument, 653, 714
    explicit, 660, 713
template member, see member template
type alias, 666
type transformation templates, 684, 714
type-dependencies, 658
variadic, see variadic template
template argument deduction, 678, 714
    compare, 680
    explicit template argument, 682
    function pointer, 686
    limited conversions, 679
    low-level const, 693
    lvalue reference parameter, 687
    multiple function parameters, 680
    parameter with nontemplate type, 680
    reference parameters, 687–689
    rvalue reference parameter, 687
    top-level const, 679
template class, see class template
template function, see function template
template parameter, 653, 714
    default template argument, 670
    class template, 671
    function template, 671
    name, 668
    restrictions on use, 669
    nontype parameter, 654, 714
    must be constant expression, 655
type requirements, 655
    scope, 668
template argument deduction, 680
type parameter, 654, 654, 714
    as friend, 666
    used in template class, 660
template parameter pack, 699, 714
    expansion, 703
    pattern, 703
template specialization, 707, 706–712, 714
    class template, 709–712
    class template member, 711
calculate function template, 707
compared to overloading, 708
declaration dependencies, 708
function template, 707
hash<key_type>, 709, 788
headers, 708
of namespace member, 709, 788
partial, class template, 711, 714
    scope, 708
    template<>, 707
template<>
    default template argument, 671
template specialization, 707
temporary, 62, 80
terminate function, 773, 818
    exception handling, 196, 200
    machine-dependent, 196
terminology
    const reference, 61
    iterator, 109
    object, 42
    overloaded new and delete, 822
test, bitset, 727
TextQuery, 485
    class definition, 487
    constructor, 488
    main program, 486
    program design, 485
    query, 489
    revisited, 635
this pointer, 257, 306
  static members, 301
  as argument, 266
  in return, 260
  overloaded
    on const, 276
    on lvalue or rvalue reference, 546
throw, 193, 193, 200, 772, 818
  execution flow, 196, 773
  pointer to local object, 774
  rethrow, 776, 818
runtime_error, 194
  throw(), exception specification, 780
  tie member, ostream, 315
  to_string, 368
Token, 849
  assignment operators, 850
  copy control, 851
  copyUnion, 851
  default constructor, 850
  discriminant, 850
tolower, 92
top
  priority_queue, 371
  stack, 371
top-level const, 64, 80
  and auto, 69
  argument and parameter, 212
decltype, 71
  parameter, 232
  template argument deduction, 679
toupper, 92
ToyAnimal, virtual base class, 815
trailing return type, 229, 252
  function template, 684
  lambda expression, 396
  pointer to array, 229
  pointer to function, 250
transform
  algorithm, 396, 874
  program, 442
translation unit, 4
trunc (file mode), 319
try block, 193, 194, 200, 773, 818
tuple, 718, 770
  findBook, program, 721
  equality and relational operators, 720
  header, 718
  initialization, 718
  make_tuple, 718
  return value, 721
  value initialization, 718
value initialization, 718
type
type alias, 67, 80
  template, 666
  alias declaration, 68
  arithmetic, 32, 78
  built-in, 2, 26, 32–34
  checking, 46, 80
    argument and parameter, 203
    array reference parameter, 217
    function return value, 223
    name lookup, 235
  class, 19, 26
    compound, 50, 50–58, 78
    conversion, see conversion
dynamic, 601, 650
  incomplete, 279, 306
  integral, 32, 79
  literal, 66
    class type, 299
    specifier, 41, 80
static, 601, 650
type alias declaration, 68, 78, 80
  pointer to array, 229
  pointer to function, 249
  pointer to member, 839
template type, 666
type independence, algorithms, 377
type member, class, 271
type parameter, see template parameter
type transformation templates, 684, 714
type_traits, 685
type_info, 864
  header, 197
  name, 831
  no copy or assign, 831
  operations, 831
  returned from typeid, 827
type_traits
  header, 684
  remove_pointer, 685
  remove_reference, 684
    and move, 691
type transformation templates, 685
typedef, 67, 80
  const, 68
  and pointer, to const, 68
  pointer to array, 229
  pointer to function, 249
typeid operator, 826, 827, 864
  returns type_info, 827
Index 935

**typeinfo header**, 826, 827, 831

**typename**
- compared to class, 654
- required for type member, 670
- template parameter, 654

**U**

unary operators, 134, 169
- overloaded operator, 552

unary predicate, 386, 418

unary_function deprecated, 579

uncaught exception, 773

undefined behavior, 35, 80
- base class destructor not virtual, 622
- bitwise operators and signed values, 153
- caching end() iterator, 355
- cstring functions, 122
- dangling pointer, 463
- default initialized members of built-in type, 263
- delete of invalid pointer, 460
- destination sequence too small, 382
- element access empty container, 346
- invalidated iterator, 107, 353
- missing return statement, 224
- misuse of smart pointer get, 466
- omitting [] when deleting array, 479
- operand order of evaluation, 138, 149
- out-of-range subscript, 93
- out-of-range value assigned to signed type, 35
- overflow and underflow, 140
- pointer casts, 163
- pointer comparisons, 123
- return reference or pointer to local variable, 225
- string invalid initializer, 361
- uninitialized
  - dynamic object, 458
  - local variable, 205
  - pointer, 54
  - variable, 45
- using unconstructed memory, 482
- using unmatched match object, 737
- writing to a const object, 163
- wrong deleter with smart pointer, 480

**underflow_error**, 197

unformatted IO, 761, 770

**istream**, 761

**multi-byte, istream**, 763

**single-byte, istream**, 761

**unget, istream**, 761

**uniform_int_distribution**, 746

**uniform_real_distribution**, 750

**uninitialized**, 8, 28, 44, 80
- pointer, undefined behavior, 54
- variable, undefined behavior, 45

**uninitialized_copy**, 483

**move iterator**, 543

**uninitialized_fill**, 483

**union**, 847, 864
- anonymous, 848, 862
- class type member, 848
  - assignment operators, 850
  - copy control, 851
  - default constructor, 850
  - deleted copy control, 849
  - placement new, 851
  - definition, 848
  - discriminant, 850
  - restrictions, 847

**unique**, 384, 878
- list and forward_list, 415
- unique_copy, 403, 878

**unique_ptr**, 450, 470–472, 491
- *(dereference), 451
- copy and assignment, 470
- definition, 470, 472
- deleter, 472, 491
  - bound at compile time, 678
- dynamically allocated array, 479
- initialization, 470
- pitfalls, 469
- release, 470
- reset, 470
- return value, 471
- transfer ownership, 470
- with new, 470

**unitbuf**, manipulator, 315

unnamed namespace, 791, 818
- local to file, 791
- replace file static, 792

unordered container, 443, 448
- see also container
- see also associative container
- bucket management, 444
- hash<key_type> specialization, 709, 788
  - compatible with == (equality), 710
  - key_type requirements, 445
override default hash, 446
unordered_map, 448
  see also unordered container
* (dereference), 429
[ ] (subscript), 435, 448
  adds element, 435
at, 435
definition, 423
header, 420
list initialization, 423
word_count program, 444
unordered_multimap, 448
  see also unordered container
* (dereference), 429
definition, 423
has no subscript operator, 435
insert, 433
list initialization, 423
unordered_multiset, 448
  see also unordered container
insert, 433
iterator, 429
list initialization, 423
override default hash, 446
unordered_set, 448
  see also unordered container
header, 420
iterator, 429
list initialization, 423
unordered_multiset, 448
  see also unordered container
insert, 433
iterator, 429
list initialization, 423
override default hash, 446
unsigned, 34, 80
  char, 34
  conversion, 36
  conversion from signed, 34
  conversion to signed, 160
  literal (numU or numu), 41
  size return type, 88
unsigned type, 34
unwinding, stack, 773, 818
upper_bound
  algorithm, 873
  ordered container, 438
  used in Basket, 632
uppercase, manipulator, 755
use count, see reference count
user-defined conversion, see class type conversion
user-defined header, 76–77
const and constexpr, 76
default argument, 238
definition, 423
definition, 423
definition, 423

definition, 423
definition, 423

definition, 423
definition, 423
海量文本内容
using =, see type alias declaration
using declaration, 82, 132, 793, 818
  access control, 615
  not in header files, 83
  overloaded function, 800
  overloaded inherited functions, 621
  scope, 793
using directive, 793, 818
  overloaded function, 801
pitfalls, 795
scope, 793, 794
  name collisions, 795
utility header, 426, 530, 533, 694
valid, program, 740
valid but unspecified, 537
valid pointer, 52
value initialization, 98, 132
dynamically allocated, object, 459
map subscript operator, 435
new[], 478
resize, 352
sequential container, 336
tuple, 718
uses default constructor, 293
vector, 98
value_type
  associative container, 428, 448
  sequential container, 333
valuelike class, copy control, 512
varargs, 222
variable, 8, 28, 41, 41–49, 80
cost, 59
costexpr, 66
declaration, 45
class type, 294
define before use, 46
defined after label, 182, 192
definition, 41, 45
extern, 45
extern and const, 60
initialization, 42, 43, 79
is lvalue, 533
lifetime, 204
local, 204, 252
preprocessor, 79
variadic template, 699, 714
declaration dependencies, 702
forwarding, 704
usage pattern, 706
function matching, 702
pack expansion, 702–704
parameter pack, 699
print program, 701
recursive function, 701
sizeof . . . , 700
vector, 96–105, 132, 373
see also container
see also sequential container
see also iterator
[ ] (subscript), 103, 132, 347
= (assignment), list initialization, 145
at, 348
capacity, 356
capacity program, 357
definition, 97
difference_type, 112
erase, changes container size, 385
header, 96, 329
initialization, 97–101, 334–337
initialization from array, 125
list initialization, 98, 336
memory management, 355
overview, 326
push_back, invalidates iterator, 354
random-access iterator, 412
reserve, 356
subscript range, 105
TextQuery class, 485
value initialization, 98, 336
viable function, 243, 252
see also function matching
virtual base class, 811, 818
ambiguities, 812
Bear, 812
class derivation list, 812
conversion, 812
derived class constructor, 813
iostream, 810
name lookup, 812
order of destruction, 815
order of initialization, 814
ostream, 810
Raccoon, 812
ToyAnimal, 815
ZooAnimal, 811
virtual function, 592, 595, 603–610, 650
compared to run-time type identification, 829
default argument, 607
derived class, 596
destructor, 622
exception specification, 781
final specifier, 607
in constructor, destructor, 627
multiple inheritance, 807
overloaded function, 621
override, 595, 650
override specifier, 593, 596, 606
overriding run-time binding, 607
overview, 595
pure, 609
resolved at run time, 604, 605
return type, 606
scope, 620
type-sensitive equality, 829
virtual inheritance, see virtual base class
Visual Studio, 5
void, 32, 80
return type, 223
void*, 56, 80
conversion from pointer, 161
volatile, 856, 864
pointer, 856
synthesized copy-control members, 857
vowel counting, program, 179

W
wcerr, 311
wchar_t, 33
liter, 40
wchar_t streams, 311
wcin, 311
wcout, 311
weak ordering, strict, 448
weak_ptr, 450, 473–475, 491
definition, 473
initialization, 473
lock, 473
StrBlobPtr, 474
wfstream, 311
what, exception, 195, 782
while statement, 11, 28, 183, 183–185, 200
    condition, 12, 183
wide character streams, 311
wifstream, 311
window, console, 6
Window_mgr, 279
wiostream, 311
wistream, 311
wstringstream, 311
wofstream, 311
word, 33, 80
word_count program
    map, 421
    set, 422
    unordered_map, 444
word_transform program, 441
WordQuery, 637, 642
wostream, 311
wostringstream, 311
wregex, 733
write, ostream, 763
wstringstream, 311

X
\Xnnn (hexadecimal escape sequence), 39

Z
ZooAnimal
    program, 802
    virtual base class, 811