PROFESSIONAL C++

INTRODUCTION ................................................................. xlvii

▸ PART I  INTRODUCTION TO PROFESSIONAL C++
CHAPTER 1  A Crash Course in C++ and the Standard Library .................. 3
CHAPTER 2  Working with Strings and String Views .............................. 57
CHAPTER 3  Coding with Style.................................................. 71

▸ PART II  PROFESSIONAL C++ SOFTWARE DESIGN
CHAPTER 4  Designing Professional C++ Programs .............................. 95
CHAPTER 5  Designing with Objects ........................................... 123
CHAPTER 6  Designing for Reuse ................................................ 143

▸ PART III  C++ CODING THE PROFESSIONAL WAY
CHAPTER 7  Memory Management ............................................. 163
CHAPTER 8  Gaining Proficiency with Classes and Objects ..................... 199
CHAPTER 9  Mastering Classes and Objects ................................... 231
CHAPTER 10 Discovering Inheritance Techniques ............................... 277
CHAPTER 11 C++ Quirks, Oddities, and Incidentals ............................ 333
CHAPTER 12 Writing Generic Code with Templates ............................. 373
CHAPTER 13 Demystifying C++ I/O ........................................... 409
CHAPTER 14 Handling Errors .................................................... 433
CHAPTER 15 Overloading C++ Operators ....................................... 473
CHAPTER 16 Overview of the C++ Standard Library ............................. 507
CHAPTER 17 Understanding Containers and Iterators .......................... 535
CHAPTER 18 Mastering Standard Library Algorithms ........................... 607
CHAPTER 19 String Localization and Regular Expressions .................... 663
CHAPTER 20 Additional Library Utilities ....................................... 691

Continues
Dedicated to my parents and my brother, who are always there for me. Their support and patience helped me in finishing this book.
ABOUT THE AUTHOR

MARC GREGOIRE is a software architect from Belgium. He graduated from the University of Leuven, Belgium, with a degree in “Burgerlijk ingenieur in de computer wetenschappen” (equivalent to master of science in engineering: computer science). The year after, he received an advanced master’s degree in artificial intelligence, cum laude, at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working with international teams stretching from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is a software architect at Nikon Metrology (www.nikonmetrology.com), a division of Nikon and a leading provider of precision optical instruments and metrology solutions for 3D geometric inspection.

His main expertise is in C/C++, and specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24/7 on Windows and Linux platforms: for example, KNX/EIB home automation software. In addition to C/C++, Marc also likes C# and uses PHP for creating web pages.

Since April 2007, he has received the annual Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (www.becpp.org), co-author of C++ Standard Library Quick Reference (Apress), technical editor for numerous books for several publishers, and a member on the CodeGuru forum (as Marc G). He maintains a blog at www.nuonsoft.com/blog/, and is passionate about traveling and gastronomic restaurants.
ABOUT THE TECHNICAL EDITOR

PETER VAN WEERT is a Belgian software engineer, whose main interests and expertise are in C++, programming languages, algorithms, and data structures.

He received his master of science in computer science from the University of Leuven, Belgium, summa cum laude, with congratulations of the Board of Examiners. In 2010, the same university awarded him a PhD for his research on the efficient compilation of rule-based programming languages (mainly Java). During his doctoral studies, he was a teaching assistant for courses on object-oriented analysis and design, Java programming, and declarative programming languages.

After his studies, Peter worked for Nikon Metrology on large-scale, industrial-application software in the area of 3D laser scanning and point cloud inspection. In 2017, he joined the software R&D unit of Nobel Biocare, which specializes in digital dentistry software. Throughout his professional career, Peter has mastered C++ software development, as well as the management, refactoring, and debugging of very large code bases. He also gained further proficiency in all aspects of the software development process, including the analysis of functional and technical requirements, and Agile- and Scrum-based project and team management.

Peter is a regular speaker at, and board member of, the Belgian C++ Users Group. He also co-authored two books: C++ Standard Library Quick Reference and Beginning C++ (5th edition), both published by Apress.
ACKNOWLEDGMENTS

I THANK THE JOHN WILEY & SONS AND WROX Press editorial and production teams for their support. Especially, thank you to Jim Minatel, executive editor at Wiley, for giving me a chance to write this new edition, Adaobi Obi Tulton, project editor, for managing this project, and Marylouise Wiack, copy editor, for improving readability and consistency and making sure the text is grammatically correct.

A very special thank you to my technical editor, Peter Van Weert, for his outstanding technical review. His many constructive comments and ideas have certainly made this book better.

Of course, the support and patience of my parents and my brother were very important in finishing this book. I would also like to express my sincere gratitude toward my employer, Nikon Metrology, for supporting me during this project.

Finally, I thank you, the reader, for trying this approach to professional C++ software development.
# CONTENTS

## INTRODUCTION

---

### PART I: INTRODUCTION TO PROFESSIONAL C++

#### CHAPTER 1: A CRASH COURSE IN C++ AND THE STANDARD LIBRARY

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Basics of C++</td>
<td>4</td>
</tr>
<tr>
<td>The Obligatory Hello, World</td>
<td>4</td>
</tr>
<tr>
<td>Comments</td>
<td>4</td>
</tr>
<tr>
<td>Preprocessor Directives</td>
<td>5</td>
</tr>
<tr>
<td>The main() Function</td>
<td>6</td>
</tr>
<tr>
<td>I/O Streams</td>
<td>7</td>
</tr>
<tr>
<td>Namespaces</td>
<td>8</td>
</tr>
<tr>
<td>Literals</td>
<td>10</td>
</tr>
<tr>
<td>Variables</td>
<td>10</td>
</tr>
<tr>
<td>Operators</td>
<td>13</td>
</tr>
<tr>
<td>Types</td>
<td>15</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>15</td>
</tr>
<tr>
<td>Structs</td>
<td>16</td>
</tr>
<tr>
<td>Conditional Statements</td>
<td>17</td>
</tr>
<tr>
<td>if/else Statements</td>
<td>17</td>
</tr>
<tr>
<td>switch Statements</td>
<td>18</td>
</tr>
<tr>
<td>The Conditional Operator</td>
<td>20</td>
</tr>
<tr>
<td>Logical Evaluation Operators</td>
<td>20</td>
</tr>
<tr>
<td>Functions</td>
<td>21</td>
</tr>
<tr>
<td>Function Return Type Deduction</td>
<td>22</td>
</tr>
<tr>
<td>Current Function’s Name</td>
<td>23</td>
</tr>
<tr>
<td>C-Style Arrays</td>
<td>23</td>
</tr>
<tr>
<td>std::array</td>
<td>25</td>
</tr>
<tr>
<td>std::vector</td>
<td>25</td>
</tr>
<tr>
<td>Structured Bindings</td>
<td>26</td>
</tr>
<tr>
<td>Loops</td>
<td>26</td>
</tr>
<tr>
<td>The while Loop</td>
<td>26</td>
</tr>
<tr>
<td>The do/while Loop</td>
<td>27</td>
</tr>
<tr>
<td>The for Loop</td>
<td>27</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>The Range-Based for Loop</td>
<td>27</td>
</tr>
<tr>
<td>Initializer Lists</td>
<td>28</td>
</tr>
<tr>
<td>Those Are the Basics</td>
<td>28</td>
</tr>
<tr>
<td><strong>Diving Deeper into C++</strong></td>
<td>28</td>
</tr>
<tr>
<td>Strings in C++</td>
<td>29</td>
</tr>
<tr>
<td>Pointers and Dynamic Memory</td>
<td>29</td>
</tr>
<tr>
<td>The Stack and the Heap</td>
<td>29</td>
</tr>
<tr>
<td>Working with Pointers</td>
<td>30</td>
</tr>
<tr>
<td>Dynamically Allocated Arrays</td>
<td>31</td>
</tr>
<tr>
<td>Null Pointer Constant</td>
<td>32</td>
</tr>
<tr>
<td>Smart Pointers</td>
<td>33</td>
</tr>
<tr>
<td>The Many Uses of const</td>
<td>35</td>
</tr>
<tr>
<td>const Constants</td>
<td>35</td>
</tr>
<tr>
<td>const to Protect Parameters</td>
<td>35</td>
</tr>
<tr>
<td>References</td>
<td>35</td>
</tr>
<tr>
<td>Pass By Reference</td>
<td>36</td>
</tr>
<tr>
<td>Pass By const Reference</td>
<td>37</td>
</tr>
<tr>
<td>Exceptions</td>
<td>37</td>
</tr>
<tr>
<td>Type Inference</td>
<td>38</td>
</tr>
<tr>
<td>The auto Keyword</td>
<td>39</td>
</tr>
<tr>
<td>The decltype Keyword</td>
<td>40</td>
</tr>
<tr>
<td><strong>C++ as an Object-Oriented Language</strong></td>
<td>40</td>
</tr>
<tr>
<td>Defining Classes</td>
<td>40</td>
</tr>
<tr>
<td>Using Classes</td>
<td>43</td>
</tr>
<tr>
<td><strong>Uniform Initialization</strong></td>
<td>43</td>
</tr>
<tr>
<td>Direct List Initialization versus Copy List Initialization</td>
<td>45</td>
</tr>
<tr>
<td><strong>The Standard Library</strong></td>
<td>46</td>
</tr>
<tr>
<td><strong>Your First Useful C++ Program</strong></td>
<td>46</td>
</tr>
<tr>
<td>An Employee Records System</td>
<td>46</td>
</tr>
<tr>
<td>The Employee Class</td>
<td>47</td>
</tr>
<tr>
<td>Employee.h</td>
<td>47</td>
</tr>
<tr>
<td>Employee.cpp</td>
<td>48</td>
</tr>
<tr>
<td>EmployeeTest.cpp</td>
<td>50</td>
</tr>
<tr>
<td>The Database Class</td>
<td>50</td>
</tr>
<tr>
<td>Database.h</td>
<td>50</td>
</tr>
<tr>
<td>Database.cpp</td>
<td>51</td>
</tr>
<tr>
<td>DatabaseTest.cpp</td>
<td>52</td>
</tr>
<tr>
<td>The User Interface</td>
<td>53</td>
</tr>
<tr>
<td>Evaluating the Program</td>
<td>55</td>
</tr>
<tr>
<td>Summary</td>
<td>56</td>
</tr>
</tbody>
</table>
CHAPTER 2: WORKING WITH STRINGS AND STRING VIEWS  57

Dynamic Strings  58
  C-Style Strings  58
  String Literals  60
    Raw String Literals  60
  The C++ std::string Class  62
    What Is Wrong with C-Style Strings?  62
    Using the string Class  62
    std::string Literals  64
  High-Level Numeric Conversions  64
  Low-Level Numeric Conversions  65
The std::string_view Class  67
  std::string_view Literals  69
Nonstandard Strings  69
Summary  69

CHAPTER 3: CODING WITH STYLE  71

The Importance of Looking Good  71
  Thinking Ahead  72
  Elements of Good Style  72
Documenting Your Code  72
  Reasons to Write Comments  72
    Commenting to Explain Usage  72
    Commenting to Explain Complicated Code  74
    Commenting to Convey Meta-information  75
  Commenting Styles  77
    Commenting Every Line  77
    Prefix Comments  78
    Fixed-Format Comments  79
    Ad Hoc Comments  80
    Self-Documenting Code  81
Decomposition  81
  Decomposition through Refactoring  82
  Decomposition by Design  83
  Decomposition in This Book  83
Naming  83
  Choosing a Good Name  83
  Naming Conventions  84
    Counters  84
    Prefixes  84
<table>
<thead>
<tr>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hungarian Notation</td>
<td>85</td>
</tr>
<tr>
<td>Getters and Setters</td>
<td>86</td>
</tr>
<tr>
<td>Capitalization</td>
<td>86</td>
</tr>
<tr>
<td>Namespaced Constants</td>
<td>86</td>
</tr>
<tr>
<td><strong>Using Language Features with Style</strong></td>
<td>86</td>
</tr>
<tr>
<td>Use Constants</td>
<td>87</td>
</tr>
<tr>
<td>Use References Instead of Pointers</td>
<td>87</td>
</tr>
<tr>
<td>Use Custom Exceptions</td>
<td>88</td>
</tr>
<tr>
<td><strong>Formatting</strong></td>
<td>88</td>
</tr>
<tr>
<td>The Curly Brace Alignment Debate</td>
<td>88</td>
</tr>
<tr>
<td>Coming to Blows over Spaces and Parentheses</td>
<td>89</td>
</tr>
<tr>
<td>Spaces and Tabs</td>
<td>90</td>
</tr>
<tr>
<td><strong>Stylistic Challenges</strong></td>
<td>90</td>
</tr>
<tr>
<td>Summary</td>
<td>91</td>
</tr>
<tr>
<td><strong>PART II: PROFESSIONAL C++ SOFTWARE DESIGN</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CHAPTER 4: DESIGNING PROFESSIONAL C++ PROGRAMS</strong></td>
<td>95</td>
</tr>
<tr>
<td>What Is Programming Design</td>
<td>96</td>
</tr>
<tr>
<td>The Importance of Programming Design</td>
<td>97</td>
</tr>
<tr>
<td>Designing for C++</td>
<td>99</td>
</tr>
<tr>
<td><strong>Two Rules for C++ Design</strong></td>
<td>100</td>
</tr>
<tr>
<td>Abstraction</td>
<td>100</td>
</tr>
<tr>
<td>Benefiting from Abstraction</td>
<td>100</td>
</tr>
<tr>
<td>Incorporating Abstraction in Your Design</td>
<td>101</td>
</tr>
<tr>
<td>Reuse</td>
<td>101</td>
</tr>
<tr>
<td>Writing Reusable Code</td>
<td>102</td>
</tr>
<tr>
<td>Reusing Designs</td>
<td>103</td>
</tr>
<tr>
<td><strong>Reusing Existing Code</strong></td>
<td>103</td>
</tr>
<tr>
<td>A Note on Terminology</td>
<td>104</td>
</tr>
<tr>
<td>Deciding Whether or Not to Reuse Code</td>
<td>105</td>
</tr>
<tr>
<td>Advantages to Reusing Code</td>
<td>105</td>
</tr>
<tr>
<td>Disadvantages to Reusing Code</td>
<td>105</td>
</tr>
<tr>
<td>Putting It Together to Make a Decision</td>
<td>106</td>
</tr>
<tr>
<td><strong>Strategies for Reusing Code</strong></td>
<td>107</td>
</tr>
<tr>
<td>Understand the Capabilities and Limitations</td>
<td>107</td>
</tr>
<tr>
<td>Understand the Performance</td>
<td>108</td>
</tr>
<tr>
<td>Understand Platform Limitations</td>
<td>110</td>
</tr>
<tr>
<td>Understand Licensing and Support</td>
<td>110</td>
</tr>
<tr>
<td>Know Where to Find Help</td>
<td>111</td>
</tr>
<tr>
<td>Prototype</td>
<td>111</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Bundling Third-Party Applications</td>
<td>112</td>
</tr>
<tr>
<td>Open-Source Libraries</td>
<td>112</td>
</tr>
<tr>
<td>The Open-Source Movements</td>
<td>112</td>
</tr>
<tr>
<td>Finding and Using Open-Source Libraries</td>
<td>113</td>
</tr>
<tr>
<td>Guidelines for Using Open-Source Code</td>
<td>113</td>
</tr>
<tr>
<td>The C++ Standard Library</td>
<td>114</td>
</tr>
<tr>
<td>C Standard Library</td>
<td>114</td>
</tr>
<tr>
<td>Deciding Whether or Not to Use the Standard Library</td>
<td>114</td>
</tr>
<tr>
<td>Designing a Chess Program</td>
<td>114</td>
</tr>
<tr>
<td>Requirements</td>
<td>115</td>
</tr>
<tr>
<td>Design Steps</td>
<td>115</td>
</tr>
<tr>
<td>Divide the Program into Subsystems</td>
<td>115</td>
</tr>
<tr>
<td>Choose Threading Models</td>
<td>117</td>
</tr>
<tr>
<td>Specify Class Hierarchies for Each Subsystem</td>
<td>118</td>
</tr>
<tr>
<td>Specify Classes, Data Structures, Algorithms, and Patterns for Each Subsystem</td>
<td>118</td>
</tr>
<tr>
<td>Specify Error Handling for Each Subsystem</td>
<td>120</td>
</tr>
<tr>
<td>Summary</td>
<td>121</td>
</tr>
<tr>
<td><strong>CHAPTER 5: DESIGNING WITH OBJECTS</strong></td>
<td>123</td>
</tr>
<tr>
<td>Am I Thinking Procedurally?</td>
<td>124</td>
</tr>
<tr>
<td>The Object-Oriented Philosophy</td>
<td>124</td>
</tr>
<tr>
<td>Classes</td>
<td>124</td>
</tr>
<tr>
<td>Components</td>
<td>125</td>
</tr>
<tr>
<td>Properties</td>
<td>125</td>
</tr>
<tr>
<td>Behaviors</td>
<td>126</td>
</tr>
<tr>
<td>Bringing It All Together</td>
<td>126</td>
</tr>
<tr>
<td>Living in a World of Objects</td>
<td>127</td>
</tr>
<tr>
<td>Over-Objectification</td>
<td>127</td>
</tr>
<tr>
<td>Overly General Objects</td>
<td>128</td>
</tr>
<tr>
<td>Object Relationships</td>
<td>129</td>
</tr>
<tr>
<td>The Has-A Relationship</td>
<td>129</td>
</tr>
<tr>
<td>The Is-A Relationship (Inheritance)</td>
<td>130</td>
</tr>
<tr>
<td>Inheritance Techniques</td>
<td>130</td>
</tr>
<tr>
<td>Polymorphism versus Code Reuse</td>
<td>131</td>
</tr>
<tr>
<td>The Fine Line between Has-A and Is-A</td>
<td>132</td>
</tr>
<tr>
<td>The Not-A Relationship</td>
<td>135</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>136</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>137</td>
</tr>
<tr>
<td>Mixin Classes</td>
<td>138</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>Abstraction 138</td>
<td></td>
</tr>
<tr>
<td>Interface versus Implementation 138</td>
<td></td>
</tr>
<tr>
<td>Deciding on an Exposed Interface 139</td>
<td></td>
</tr>
<tr>
<td>Consider the Audience 139</td>
<td></td>
</tr>
<tr>
<td>Consider the Purpose 139</td>
<td></td>
</tr>
<tr>
<td>Consider the Future 141</td>
<td></td>
</tr>
<tr>
<td>Designing a Successful Abstraction 141</td>
<td></td>
</tr>
<tr>
<td>Summary 142</td>
<td></td>
</tr>
<tr>
<td>CHAPTER 6: DESIGNING FOR REUSE 143</td>
<td></td>
</tr>
<tr>
<td>The Reuse Philosophy 144</td>
<td></td>
</tr>
<tr>
<td>How to Design Reusable Code 144</td>
<td></td>
</tr>
<tr>
<td>Use Abstraction 145</td>
<td></td>
</tr>
<tr>
<td>Structure Your Code for Optimal Reuse 146</td>
<td></td>
</tr>
<tr>
<td>Avoid Combining Unrelated or Logically Separate Concepts 146</td>
<td></td>
</tr>
<tr>
<td>Use Templates for Generic Data Structures and Algorithms 148</td>
<td></td>
</tr>
<tr>
<td>Provide Appropriate Checks and Safeguards 150</td>
<td></td>
</tr>
<tr>
<td>Design for Extensibility 151</td>
<td></td>
</tr>
<tr>
<td>Design Usable Interfaces 153</td>
<td></td>
</tr>
<tr>
<td>Design Interfaces That Are Easy to Use 153</td>
<td></td>
</tr>
<tr>
<td>Design General-Purpose Interfaces 157</td>
<td></td>
</tr>
<tr>
<td>Reconciling Generality and Ease of Use 157</td>
<td></td>
</tr>
<tr>
<td>The SOLID Principles 158</td>
<td></td>
</tr>
<tr>
<td>Summary 159</td>
<td></td>
</tr>
<tr>
<td>PART III: C++ CODING THE PROFESSIONAL WAY</td>
<td></td>
</tr>
<tr>
<td>CHAPTER 7: MEMORY MANAGEMENT 163</td>
<td></td>
</tr>
<tr>
<td>Working with Dynamic Memory 164</td>
<td></td>
</tr>
<tr>
<td>How to Picture Memory 164</td>
<td></td>
</tr>
<tr>
<td>Allocation and Deallocation 166</td>
<td></td>
</tr>
<tr>
<td>Using new and delete 166</td>
<td></td>
</tr>
<tr>
<td>What about My Good Friend malloc? 167</td>
<td></td>
</tr>
<tr>
<td>When Memory Allocation Fails 167</td>
<td></td>
</tr>
<tr>
<td>Arrays 168</td>
<td></td>
</tr>
<tr>
<td>Arrays of Basic Types 168</td>
<td></td>
</tr>
<tr>
<td>Arrays of Objects 170</td>
<td></td>
</tr>
<tr>
<td>Deleting Arrays 171</td>
<td></td>
</tr>
<tr>
<td>Multi-dimensional Arrays 172</td>
<td></td>
</tr>
<tr>
<td>Working with Pointers 175</td>
<td></td>
</tr>
</tbody>
</table>
CONTENTS

A Mental Model for Pointers 175
Casting with Pointers 176
Array-Pointer Duality 177
Arrays Are Pointers! 177
Not All Pointers Are Arrays! 179
Low-Level Memory Operations 179
Pointer Arithmetic 179
Custom Memory Management 180
Garbage Collection 181
Object Pools 182
Smart Pointers 182
unique_ptr 183
Creating unique_ptrs 183
Using unique_ptrs 185
unique_ptr and C-Style Arrays 186
Custom Deleters 186
shared_ptr 186
Casting a shared_ptr 187
The Need for Reference Counting 188
Aliasing 189
weak_ptr 189
Move Semantics 190
enable_shared_from_this 191
The Old Deprecated/Removed auto_ptr 192
Common Memory Pitfalls 192
Underallocating Strings 192
Accessing Out-of-Bounds Memory 193
Memory Leaks 194
Finding and Fixing Memory Leaks in Windows with Visual C++ 195
Finding and Fixing Memory Leaks in Linux with Valgrind 196
Double-Deleting and Invalid Pointers 197
Summary 197

CHAPTER 8: GAINING PROFICIENCY WITH CLASSES AND OBJECTS 199

Introducing the Spreadsheet Example 200
Writing Classes 200
Class Definitions 200
Class Members 201
Access Control 201
Order of Declarations 203
In-Class Member Initializers 203
<table>
<thead>
<tr>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining Methods</td>
<td>203</td>
</tr>
<tr>
<td>Accessing Data Members</td>
<td>204</td>
</tr>
<tr>
<td>Calling Other Methods</td>
<td>204</td>
</tr>
<tr>
<td>The this Pointer</td>
<td>206</td>
</tr>
<tr>
<td>Using Objects</td>
<td>207</td>
</tr>
<tr>
<td>Objects on the Stack</td>
<td>207</td>
</tr>
<tr>
<td>Objects on the Heap</td>
<td>207</td>
</tr>
<tr>
<td><strong>Object Life Cycles</strong></td>
<td>208</td>
</tr>
<tr>
<td>Object Creation</td>
<td>208</td>
</tr>
<tr>
<td>Writing Constructors</td>
<td>209</td>
</tr>
<tr>
<td>Using Constructors</td>
<td>210</td>
</tr>
<tr>
<td>Providing Multiple Constructors</td>
<td>211</td>
</tr>
<tr>
<td>Default Constructors</td>
<td>212</td>
</tr>
<tr>
<td>Constructor Initializers</td>
<td>215</td>
</tr>
<tr>
<td>Copy Constructors</td>
<td>218</td>
</tr>
<tr>
<td>Initializer-List Constructors</td>
<td>220</td>
</tr>
<tr>
<td>Delegating Constructors</td>
<td>222</td>
</tr>
<tr>
<td>Summary of Compiler-Generated Constructors</td>
<td>222</td>
</tr>
<tr>
<td>Object Destruction</td>
<td>224</td>
</tr>
<tr>
<td>Assigning to Objects</td>
<td>225</td>
</tr>
<tr>
<td>Declaring an Assignment Operator</td>
<td>225</td>
</tr>
<tr>
<td>Defining an Assignment Operator</td>
<td>226</td>
</tr>
<tr>
<td>Explicitly Defaulted and Deleted Assignment Operator</td>
<td>227</td>
</tr>
<tr>
<td>Compiler-Generated Copy Constructor and Copy Assignment Operator</td>
<td>228</td>
</tr>
<tr>
<td>Distinguishing Copying from Assignment</td>
<td>228</td>
</tr>
<tr>
<td>Objects as Return Values</td>
<td>228</td>
</tr>
<tr>
<td>Copy Constructors and Object Members</td>
<td>229</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>230</td>
</tr>
</tbody>
</table>

**CHAPTER 9: MASTERING CLASSES AND OBJECTS**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friends</td>
<td>232</td>
</tr>
<tr>
<td>Dynamic Memory Allocation in Objects</td>
<td>233</td>
</tr>
<tr>
<td>The Spreadsheet Class</td>
<td>233</td>
</tr>
<tr>
<td>Freeing Memory with Destructors</td>
<td>235</td>
</tr>
<tr>
<td>Handling Copying and Assignment</td>
<td>236</td>
</tr>
<tr>
<td>The Spreadsheet Copy Constructor</td>
<td>239</td>
</tr>
<tr>
<td>The Spreadsheet Assignment Operator</td>
<td>240</td>
</tr>
<tr>
<td>Disallowing Assignment and Pass-By-Value</td>
<td>242</td>
</tr>
<tr>
<td>Handling Moving with Move Semantics</td>
<td>243</td>
</tr>
<tr>
<td>Rvalue References</td>
<td>243</td>
</tr>
<tr>
<td>Implementing Move Semantics</td>
<td>245</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Overriding Methods</td>
<td>281</td>
</tr>
<tr>
<td>How I Learned to Stop Worrying and Make Everything virtual</td>
<td>281</td>
</tr>
<tr>
<td>Syntax for Overriding a Method</td>
<td>282</td>
</tr>
<tr>
<td>A Client’s View of Overridden Methods</td>
<td>283</td>
</tr>
<tr>
<td>The override Keyword</td>
<td>284</td>
</tr>
<tr>
<td>The Truth about virtual</td>
<td>286</td>
</tr>
<tr>
<td>Preventing Overriding</td>
<td>290</td>
</tr>
<tr>
<td>Inheritance for Reuse</td>
<td>291</td>
</tr>
<tr>
<td>The WeatherPrediction Class</td>
<td>291</td>
</tr>
<tr>
<td>Adding Functionality in a Derived Class</td>
<td>292</td>
</tr>
<tr>
<td>Replacing Functionality in a Derived Class</td>
<td>293</td>
</tr>
<tr>
<td>Respect Your Parents</td>
<td>294</td>
</tr>
<tr>
<td>Parent Constructors</td>
<td>294</td>
</tr>
<tr>
<td>Parent Destructors</td>
<td>296</td>
</tr>
<tr>
<td>Referring to Parent Names</td>
<td>297</td>
</tr>
<tr>
<td>Casting Up and Down</td>
<td>299</td>
</tr>
<tr>
<td>Inheritance for Polymorphism</td>
<td>301</td>
</tr>
<tr>
<td>Return of the Spreadsheet</td>
<td>301</td>
</tr>
<tr>
<td>Designing the Polymorphic Spreadsheet Cell</td>
<td>301</td>
</tr>
<tr>
<td>The SpreadsheetCell Base Class</td>
<td>302</td>
</tr>
<tr>
<td>A First Attempt</td>
<td>302</td>
</tr>
<tr>
<td>Pure Virtual Methods and Abstract Base Classes</td>
<td>303</td>
</tr>
<tr>
<td>The Individual Derived Classes</td>
<td>304</td>
</tr>
<tr>
<td>StringSpreadsheetCell Class Definition</td>
<td>304</td>
</tr>
<tr>
<td>StringSpreadsheetCell Implementation</td>
<td>304</td>
</tr>
<tr>
<td>DoubleSpreadsheetCell Class Definition and Implementation</td>
<td>305</td>
</tr>
<tr>
<td>Leveraging Polymorphism</td>
<td>306</td>
</tr>
<tr>
<td>Future Considerations</td>
<td>306</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>308</td>
</tr>
<tr>
<td>Inheriting from Multiple Classes</td>
<td>308</td>
</tr>
<tr>
<td>Naming Collisions and Ambiguous Base Classes</td>
<td>309</td>
</tr>
<tr>
<td>Name Ambiguity</td>
<td>309</td>
</tr>
<tr>
<td>Ambiguous Base Classes</td>
<td>311</td>
</tr>
<tr>
<td>Uses for Multiple Inheritance</td>
<td>312</td>
</tr>
<tr>
<td>Interesting and Obscure Inheritance Issues</td>
<td>312</td>
</tr>
<tr>
<td>Changing the Overridden Method’s Characteristics</td>
<td>313</td>
</tr>
<tr>
<td>Changing the Method Return Type</td>
<td>313</td>
</tr>
<tr>
<td>Changing the Method Parameters</td>
<td>315</td>
</tr>
<tr>
<td>Inherited Constructors</td>
<td>316</td>
</tr>
<tr>
<td>Special Cases in Overriding Methods</td>
<td>320</td>
</tr>
</tbody>
</table>
## CONTENTS

The Base Class Method Is static 320  
The Base Class Method Is Overloaded 321  
The Base Class Method Is private or protected 322  
The Base Class Method Has Default Arguments 324  
The Base Class Method Has a Different Access Level 325  
Copy Constructors and Assignment Operators in Derived Classes 327  
Run-Time Type Facilities 329  
Non-public Inheritance 331  
Virtual Base Classes 331  
**Summary** 332  

**CHAPTER 11: C++ QUIRKS, ODDITIES, AND INCIDENTIALS** 333  

References 334  
Reference Variables 334  
  
  Modifying References 335  
  References to Pointers and Pointers to References 336  
Reference Data Members 336  
Reference Parameters 336  
  References from Pointers 337  
  Pass-by-Reference versus Pass-by-Value 337  
Reference Return Values 338  
Rvalue References 338  
Deciding between References and Pointers 339  

**Keyword Confusion** 343  
The const Keyword 343  
  const Variables and Parameters 343  
  const Methods 345  
  The constexpr Keyword 346  
The static Keyword 347  
  static Data Members and Methods 347  
  static Linkage 347  
  static Variables in Functions 350  
Order of Initialization of Nonlocal Variables 351  
Order of Destruction of Nonlocal Variables 351  

**Types and Casts** 351  
Type Aliases 352  
Type Aliases for Function Pointers 353  
Type Aliases for Pointers to Methods and Data Members 355  
typedefs 356
CONTENTS

Casts 357
  const_cast() 357
  static_cast() 358
  reinterpret_cast() 359
  dynamic_cast() 360
  Summary of Casts 361
Scope Resolution 362
Attributes 363
  [[noreturn]] 363
  [[deprecated]] 364
  [[fallthrough]] 364
  [[nodiscard]] 364
  [[maybe_unused]] 365
Vendor-Specific Attributes 365
User-Defined Literals 365
  Standard User-Defined Literals 367
Header Files 367
C Utilities 369
  Variable-Length Argument Lists 369
    Accessing the Arguments 370
    Why You Shouldn’t Use C-Style Variable-Length Argument Lists 371
  Preprocessor Macros 371
Summary 372

CHAPTER 12: WRITING GENERIC CODE WITH TEMPLATES 373

Overview of Templates 374
Class Templates 375
  Writing a Class Template 375
    Coding without Templates 375
    A Template Grid Class 378
    Using the Grid Template 382
  Angle Brackets 383
  How the Compiler Processes Templates 383
    Selective Instantiation 384
    Template Requirements on Types 384
  Distributing Template Code between Files 384
    Template Definitions in Header Files 384
    Template Definitions in Source Files 385
  Template Parameters 386
    Non-type Template Parameters 387
<table>
<thead>
<tr>
<th>Default Values for Type Parameters</th>
<th>389</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Parameter Deduction for Constructors</td>
<td>389</td>
</tr>
<tr>
<td>Method Templates</td>
<td>391</td>
</tr>
<tr>
<td>Method Templates with Non-type Parameters</td>
<td>393</td>
</tr>
<tr>
<td>Class Template Specialization</td>
<td>395</td>
</tr>
<tr>
<td>Deriving from Class Templates</td>
<td>397</td>
</tr>
<tr>
<td>Inheritance versus Specialization</td>
<td>399</td>
</tr>
<tr>
<td>Alias Templates</td>
<td>399</td>
</tr>
<tr>
<td><strong>Function Templates</strong></td>
<td>400</td>
</tr>
<tr>
<td>Function Template Specialization</td>
<td>401</td>
</tr>
<tr>
<td>Function Template Overloading</td>
<td>402</td>
</tr>
<tr>
<td>Function Template Overloading and Specialization Together</td>
<td>403</td>
</tr>
<tr>
<td>Friend Function Templates of Class Templates</td>
<td>403</td>
</tr>
<tr>
<td>More on Template Parameter Deduction</td>
<td>404</td>
</tr>
<tr>
<td>Return Type of Function Templates</td>
<td>405</td>
</tr>
<tr>
<td><strong>Variable Templates</strong></td>
<td>407</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>407</td>
</tr>
</tbody>
</table>

**CHAPTER 13: DEMYSTIFYING C++ I/O**

<table>
<thead>
<tr>
<th>Using Streams</th>
<th>410</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is a Stream, Anyway?</td>
<td>410</td>
</tr>
<tr>
<td>Stream Sources and Destinations</td>
<td>411</td>
</tr>
<tr>
<td>Output with Streams</td>
<td>411</td>
</tr>
<tr>
<td>Output Basics</td>
<td>412</td>
</tr>
<tr>
<td>Methods of Output Streams</td>
<td>412</td>
</tr>
<tr>
<td>Handling Output Errors</td>
<td>414</td>
</tr>
<tr>
<td>Output Manipulators</td>
<td>415</td>
</tr>
<tr>
<td>Input with Streams</td>
<td>417</td>
</tr>
<tr>
<td>Input Basics</td>
<td>417</td>
</tr>
<tr>
<td>Handling Input Errors</td>
<td>418</td>
</tr>
<tr>
<td>Input Methods</td>
<td>419</td>
</tr>
<tr>
<td>Input Manipulators</td>
<td>423</td>
</tr>
<tr>
<td>Input and Output with Objects</td>
<td>423</td>
</tr>
<tr>
<td><strong>String Streams</strong></td>
<td>425</td>
</tr>
<tr>
<td><strong>File Streams</strong></td>
<td>426</td>
</tr>
<tr>
<td>Text Mode versus Binary Mode</td>
<td>427</td>
</tr>
<tr>
<td>Jumping around with seek() and tell()</td>
<td>428</td>
</tr>
<tr>
<td>Linking Streams Together</td>
<td>430</td>
</tr>
<tr>
<td><strong>Bidirectional I/O</strong></td>
<td>431</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>432</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Choosing Argument Types</td>
<td>476</td>
</tr>
<tr>
<td>Choosing Return Types</td>
<td>477</td>
</tr>
<tr>
<td>Choosing Behavior</td>
<td>477</td>
</tr>
<tr>
<td>Operators You Shouldn’t Overload</td>
<td>477</td>
</tr>
<tr>
<td>Summary of Overloadable Operators</td>
<td>478</td>
</tr>
<tr>
<td>Rvalue References</td>
<td>481</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>482</td>
</tr>
<tr>
<td>Overloading the Arithmetic Operators</td>
<td>483</td>
</tr>
<tr>
<td>Overloading Unary Minus and Unary Plus</td>
<td>483</td>
</tr>
<tr>
<td>Overloading Increment and Decrement</td>
<td>483</td>
</tr>
<tr>
<td>Overloading the Bitwise and Binary Logical Operators</td>
<td>484</td>
</tr>
<tr>
<td>Overloading the Insertion and Extraction Operators</td>
<td>485</td>
</tr>
<tr>
<td>Overloading the Subscripting Operator</td>
<td>486</td>
</tr>
<tr>
<td>Providing Read-Only Access with operator[]</td>
<td>489</td>
</tr>
<tr>
<td>Non-integral Array Indices</td>
<td>490</td>
</tr>
<tr>
<td>Overloading the Function Call Operator</td>
<td>491</td>
</tr>
<tr>
<td>Overloading the Dereferencing Operators</td>
<td>492</td>
</tr>
<tr>
<td>Implementing operator*</td>
<td>494</td>
</tr>
<tr>
<td>Implementing operator--&gt;</td>
<td>494</td>
</tr>
<tr>
<td>What in the World Are operator.* and operator--&gt;*?</td>
<td>495</td>
</tr>
<tr>
<td>Writing Conversion Operators</td>
<td>496</td>
</tr>
<tr>
<td>Solving Ambiguity Problems with Explicit Conversion Operators</td>
<td>497</td>
</tr>
<tr>
<td>Conversions for Boolean Expressions</td>
<td>498</td>
</tr>
<tr>
<td>Overloading the Memory Allocation and Deallocation Operators</td>
<td>500</td>
</tr>
<tr>
<td>How new and delete Really Work</td>
<td>500</td>
</tr>
<tr>
<td>The New-Expression and operator new</td>
<td>501</td>
</tr>
<tr>
<td>The Delete-Expression and operator delete</td>
<td>501</td>
</tr>
<tr>
<td>Overloading operator new and operator delete</td>
<td>501</td>
</tr>
<tr>
<td>Explicitly Deleting/Defaulting operator new and operator delete</td>
<td>504</td>
</tr>
<tr>
<td>Overloading operator new and operator delete with Extra Parameters</td>
<td>504</td>
</tr>
<tr>
<td>Overloading operator delete with Size of Memory as Parameter</td>
<td>505</td>
</tr>
<tr>
<td>Summary</td>
<td>506</td>
</tr>
</tbody>
</table>

**CHAPTER 16: OVERVIEW OF THE C++ STANDARD LIBRARY**  

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding Principles</td>
<td>508</td>
</tr>
<tr>
<td>Use of Templates</td>
<td>508</td>
</tr>
<tr>
<td>Use of Operator Overloading</td>
<td>509</td>
</tr>
<tr>
<td>Overview of the C++ Standard Library</td>
<td>509</td>
</tr>
<tr>
<td>Strings</td>
<td>509</td>
</tr>
<tr>
<td>Regular Expressions</td>
<td>510</td>
</tr>
<tr>
<td>I/O Streams</td>
<td>510</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Smart Pointers</td>
<td>510</td>
</tr>
<tr>
<td>Exceptions</td>
<td>510</td>
</tr>
<tr>
<td>Mathematical Utilities</td>
<td>511</td>
</tr>
<tr>
<td>Time Utilities</td>
<td>512</td>
</tr>
<tr>
<td>Random Numbers</td>
<td>512</td>
</tr>
<tr>
<td>Initializer Lists</td>
<td>512</td>
</tr>
<tr>
<td>Pair and Tuple</td>
<td>512</td>
</tr>
<tr>
<td>optional, variant, and any</td>
<td>512</td>
</tr>
<tr>
<td>Function Objects</td>
<td>513</td>
</tr>
<tr>
<td>Filesystem</td>
<td>513</td>
</tr>
<tr>
<td>Multithreading</td>
<td>513</td>
</tr>
<tr>
<td>Type Traits</td>
<td>513</td>
</tr>
<tr>
<td>Standard Integer Types</td>
<td>514</td>
</tr>
<tr>
<td>Containers</td>
<td>514</td>
</tr>
<tr>
<td>vector</td>
<td>514</td>
</tr>
<tr>
<td>list</td>
<td>515</td>
</tr>
<tr>
<td>forward_list</td>
<td>515</td>
</tr>
<tr>
<td>deque</td>
<td>516</td>
</tr>
<tr>
<td>array</td>
<td>516</td>
</tr>
<tr>
<td>queue</td>
<td>516</td>
</tr>
<tr>
<td>priority_queue</td>
<td>516</td>
</tr>
<tr>
<td>stack</td>
<td>517</td>
</tr>
<tr>
<td>set and multiset</td>
<td>517</td>
</tr>
<tr>
<td>map and multimap</td>
<td>518</td>
</tr>
<tr>
<td>Unordered Associative Containers/Hash Tables</td>
<td>518</td>
</tr>
<tr>
<td>bitset</td>
<td>519</td>
</tr>
<tr>
<td>Summary of Standard Library Containers</td>
<td>519</td>
</tr>
<tr>
<td>Algorithms</td>
<td>522</td>
</tr>
<tr>
<td>Non-modifying Sequence Algorithms</td>
<td>523</td>
</tr>
<tr>
<td>Modifying Sequence Algorithms</td>
<td>525</td>
</tr>
<tr>
<td>Operational Algorithms</td>
<td>527</td>
</tr>
<tr>
<td>Swap and Exchange Algorithms</td>
<td>527</td>
</tr>
<tr>
<td>Partition Algorithms</td>
<td>527</td>
</tr>
<tr>
<td>Sorting Algorithms</td>
<td>528</td>
</tr>
<tr>
<td>Binary Search Algorithms</td>
<td>529</td>
</tr>
<tr>
<td>Set Algorithms</td>
<td>529</td>
</tr>
<tr>
<td>Heap Algorithms</td>
<td>529</td>
</tr>
<tr>
<td>Minimum/Maximum Algorithms</td>
<td>530</td>
</tr>
<tr>
<td>Numerical Processing Algorithms</td>
<td>530</td>
</tr>
</tbody>
</table>
map 577
    Constructing maps 577
    Inserting Elements 578
map Iterators 580
    Looking Up Elements 581
    Removing Elements 582
    Nodes 582
    map Example: Bank Account 583
multimap 585
    multimap Example: Buddy Lists 586
set 589
    set Example: Access Control List 589
multiset 590
Unordered Associative Containers or Hash Tables 591
    Hash Functions 591
unordered_map 593
    unordered_map Example: Phone Book 596
unordered_multimap 597
unordered_set/unordered_multiset 598
Other Containers 598
    Standard C-Style Arrays 598
    Strings 599
    Streams 600
bitset 600
    bitset Basics 600
    Bitwise Operators 601
    bitset Example: Representing Cable Channels 601
Summary 605

CHAPTER 18: MASTERING STANDARD LIBRARY ALGORITHMS 607

Overview of Algorithms 608
    The find and find_if Algorithms 608
    The accumulate Algorithm 611
    Move Semantics with Algorithms 612
std::function 612
Lambda Expressions 614
    Syntax 614
    Generic Lambda Expressions 617
    Lambda Capture Expressions 618
    Lambda Expressions as Return Type 618
    Lambda Expressions as Parameters 619
<table>
<thead>
<tr>
<th>Example Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples with Standard Library Algorithms</td>
<td>619</td>
</tr>
<tr>
<td>count_if</td>
<td>619</td>
</tr>
<tr>
<td>generate</td>
<td>620</td>
</tr>
<tr>
<td><strong>Function Objects</strong></td>
<td>620</td>
</tr>
<tr>
<td>Arithmetic Function Objects</td>
<td>621</td>
</tr>
<tr>
<td>Transparent Operator Functors</td>
<td>622</td>
</tr>
<tr>
<td>Comparison Function Objects</td>
<td>622</td>
</tr>
<tr>
<td>Logical Function Objects</td>
<td>623</td>
</tr>
<tr>
<td>Bitwise Function Objects</td>
<td>624</td>
</tr>
<tr>
<td>Adaptor Function Objects</td>
<td>624</td>
</tr>
<tr>
<td>Binders</td>
<td>624</td>
</tr>
<tr>
<td>Negators</td>
<td>626</td>
</tr>
<tr>
<td>Calling Member Functions</td>
<td>628</td>
</tr>
<tr>
<td>Invokers</td>
<td>629</td>
</tr>
<tr>
<td>Writing Your Own Function Objects</td>
<td>629</td>
</tr>
<tr>
<td><strong>Algorithm Details</strong></td>
<td>630</td>
</tr>
<tr>
<td>Iterators</td>
<td>631</td>
</tr>
<tr>
<td>Non-modifying Sequence Algorithms</td>
<td>631</td>
</tr>
<tr>
<td>Search Algorithms</td>
<td>631</td>
</tr>
<tr>
<td>Specialized Searchers</td>
<td>633</td>
</tr>
<tr>
<td>Comparison Algorithms</td>
<td>634</td>
</tr>
<tr>
<td>Counting Algorithms</td>
<td>636</td>
</tr>
<tr>
<td>Modifying Sequence Algorithms</td>
<td>636</td>
</tr>
<tr>
<td>transform</td>
<td>637</td>
</tr>
<tr>
<td>copy</td>
<td>638</td>
</tr>
<tr>
<td>move</td>
<td>640</td>
</tr>
<tr>
<td>replace</td>
<td>641</td>
</tr>
<tr>
<td>remove</td>
<td>641</td>
</tr>
<tr>
<td>unique</td>
<td>643</td>
</tr>
<tr>
<td>sample</td>
<td>643</td>
</tr>
<tr>
<td>reverse</td>
<td>644</td>
</tr>
<tr>
<td>shuffle</td>
<td>644</td>
</tr>
<tr>
<td>Operational Algorithms</td>
<td>644</td>
</tr>
<tr>
<td>for_each</td>
<td>644</td>
</tr>
<tr>
<td>for_each_n</td>
<td>646</td>
</tr>
<tr>
<td>Swap and Exchange Algorithms</td>
<td>646</td>
</tr>
<tr>
<td>swap</td>
<td>646</td>
</tr>
<tr>
<td>exchange</td>
<td>647</td>
</tr>
<tr>
<td>Partition Algorithms</td>
<td>647</td>
</tr>
<tr>
<td>Sorting Algorithms</td>
<td>649</td>
</tr>
<tr>
<td>Binary Search Algorithms</td>
<td>649</td>
</tr>
</tbody>
</table>
CONTENTS

Set Algorithms 650
Minimum/Maximum Algorithms 653
Parallel Algorithms 655
Numerical Processing Algorithms 655
inner_product 656
iota 656
gcd and lcm 656
reduce 656
transform_reduce 657
Scan Algorithms 657
Algorithms Example: Auditing Voter Registrations 657
The Voter Registration Audit Problem Statement 658
The auditVoterRolls Function 658
The getDuplicates Function 659
Testing the auditVoterRolls Function 660
Summary 661

CHAPTER 19: STRING LOCALIZATION AND REGULAR EXPRESSIONS 663

Localization 663
  Localizing String Literals 664
  Wide Characters 664
  Non-Western Character Sets 665
  Conversions 667
  Locales and Facets 668
    Using Locales 668
    Character Classification 669
    Character Conversion 670
    Using Facets 670
Regular Expressions 671
  ECMAScript Syntax 672
    Anchors 673
    Wildcards 673
    Alternation 673
    Grouping 673
    Repetition 673
    Precedence 674
    Character Set Matches 674
    Word Boundaries 676
    Back References 677
## CONTENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookahead</td>
<td>677</td>
</tr>
<tr>
<td>Regular Expressions and Raw String Literals</td>
<td>677</td>
</tr>
<tr>
<td>The regex Library</td>
<td>678</td>
</tr>
<tr>
<td><code>regex_match()</code></td>
<td>679</td>
</tr>
<tr>
<td><code>regex_match()</code> Example</td>
<td>680</td>
</tr>
<tr>
<td><code>regex_search()</code></td>
<td>682</td>
</tr>
<tr>
<td><code>regex_search()</code> Example</td>
<td>683</td>
</tr>
<tr>
<td><code>regex_iterator</code></td>
<td>683</td>
</tr>
<tr>
<td><code>regex_iterator</code> Example</td>
<td>684</td>
</tr>
<tr>
<td><code>regex_token_iterator</code></td>
<td>685</td>
</tr>
<tr>
<td><code>regex_token_iterator</code> Examples</td>
<td>685</td>
</tr>
<tr>
<td><code>regex_replace()</code></td>
<td>687</td>
</tr>
<tr>
<td><code>regex_replace()</code> Examples</td>
<td>688</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>690</td>
</tr>
</tbody>
</table>

### CHAPTER 20: ADDITIONAL LIBRARY UTILITIES

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratios</td>
<td>691</td>
</tr>
<tr>
<td><strong>The Chrono Library</strong></td>
<td>694</td>
</tr>
<tr>
<td>Duration</td>
<td>694</td>
</tr>
<tr>
<td>Clock</td>
<td>698</td>
</tr>
<tr>
<td>Time Point</td>
<td>700</td>
</tr>
<tr>
<td><strong>Random Number Generation</strong></td>
<td>702</td>
</tr>
<tr>
<td>Random Number Engines</td>
<td>703</td>
</tr>
<tr>
<td>Random Number Engine Adaptors</td>
<td>705</td>
</tr>
<tr>
<td>Predefined Engines and Engine Adaptors</td>
<td>705</td>
</tr>
<tr>
<td>Generating Random Numbers</td>
<td>706</td>
</tr>
<tr>
<td>Random Number Distributions</td>
<td>708</td>
</tr>
<tr>
<td><code>optional</code></td>
<td>711</td>
</tr>
<tr>
<td><code>variant</code></td>
<td>712</td>
</tr>
<tr>
<td><code>any</code></td>
<td>713</td>
</tr>
<tr>
<td><strong>Tuples</strong></td>
<td>714</td>
</tr>
<tr>
<td>Decompose Tuples</td>
<td>717</td>
</tr>
<tr>
<td>Structured Bindings</td>
<td>717</td>
</tr>
<tr>
<td><code>tie</code></td>
<td>717</td>
</tr>
<tr>
<td>Concatenation</td>
<td>718</td>
</tr>
<tr>
<td>Comparisons</td>
<td>718</td>
</tr>
<tr>
<td><code>make_from_tuple</code></td>
<td>719</td>
</tr>
<tr>
<td><code>apply</code></td>
<td>719</td>
</tr>
<tr>
<td><strong>Filesystem Support Library</strong></td>
<td>720</td>
</tr>
<tr>
<td>Path</td>
<td>720</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>Directory Entry</td>
<td>721</td>
</tr>
<tr>
<td>Helper Functions</td>
<td>721</td>
</tr>
<tr>
<td>Directory Iteration</td>
<td>722</td>
</tr>
<tr>
<td>Summary</td>
<td>723</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PART IV: MASTERING ADVANCED FEATURES OF C++</th>
</tr>
</thead>
</table>

| CHAPTER 21: CUSTOMIZING AND EXTENDING THE STANDARD LIBRARY | 727 |
|----------------------------------------------------------|
| Allocators | 728 |
| Stream Iterators | 729 |
| Output Stream Iterator | 729 |
| Input Stream Iterator | 730 |
| Iterator Adaptors | 730 |
| Reverse Iterators | 730 |
| Insert Iterators | 731 |
| Move Iterators | 733 |
| Extending the Standard Library | 735 |
| Why Extend the Standard Library? | 735 |
| Writing a Standard Library Algorithm | 735 |
| find_all() | 735 |
| Iterator Traits | 737 |
| Writing a Standard Library Container | 737 |
| A Basic Hash Map | 738 |
| Making hash_map a Standard Library Container | 747 |
| Note on Allocators | 760 |
| Note on Reversible Containers | 760 |
| Making hash_map an Unordered Associative Container | 760 |
| Note on Sequential Containers | 773 |
| Summary | 773 |

| CHAPTER 22: ADVANCED TEMPLATES | 775 |
|--------------------------------|
| More about Template Parameters | 776 |
| More about Template Type Parameters | 776 |
| Introducing Template Template Parameters | 778 |
| More about Non-type Template Parameters | 780 |
| Class Template Partial Specialization | 782 |
| Emulating Function Partial Specialization with Overloading | 786 |
| Template Recursion | 787 |
| An N-Dimensional Grid: First Attempt | 788 |
| A Real N-Dimensional Grid | 789 |
Variadic Templates 792
  Type-Safe Variable-Length Argument Lists 792
  Variable Number of Mixin Classes 795
  Folding Expressions 796
Metaprogramming 797
  Factorial at Compile Time 798
  Loop Unrolling 799
  Printing Tuples 800
    constexpr if 802
    Using a Compile-Time Integer Sequence with Folding 803
  Type Traits 803
    Using Type Categories 805
    Using Type Relations 807
    Using enable_if 808
    Using constexpr if to Simplify enable_if Constructs 810
  Logical Operator Traits 811
  Metaprogramming Conclusion 811
Summary 812
CHAPTER 23: MULTITHREADED PROGRAMMING WITH C++ 813
Introduction 814
  Race Conditions 815
  Tearing 817
  Deadlocks 817
  False-Sharing 818
Threads 819
  Thread with Function Pointer 819
  Thread with Function Object 820
  Thread with Lambda 822
  Thread with Member Function 823
  Thread Local Storage 823
  Cancelling Threads 824
  Retrieving Results from Threads 824
  Copying and Rethrowing Exceptions 824
Atomic Operations Library 827
  Atomic Type Example 828
  Atomic Operations 830
Mutual Exclusion 831
  Mutex Classes 831
    Non-timed Mutex Classes 832
    Timed Mutex Classes 832
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUP as a Product</td>
<td>868</td>
</tr>
<tr>
<td>RUP as a Process</td>
<td>869</td>
</tr>
<tr>
<td>RUP in Practice</td>
<td>869</td>
</tr>
<tr>
<td>Scrum</td>
<td>869</td>
</tr>
<tr>
<td>Roles</td>
<td>870</td>
</tr>
<tr>
<td>The Process</td>
<td>870</td>
</tr>
<tr>
<td>Benefits of Scrum</td>
<td>871</td>
</tr>
<tr>
<td>Drawbacks of Scrum</td>
<td>872</td>
</tr>
<tr>
<td>Extreme Programming</td>
<td>872</td>
</tr>
<tr>
<td>XP in Theory</td>
<td>872</td>
</tr>
<tr>
<td>XP in Practice</td>
<td>876</td>
</tr>
<tr>
<td>Software Triage</td>
<td>876</td>
</tr>
<tr>
<td>Building Your Own Process and Methodology</td>
<td>877</td>
</tr>
<tr>
<td>Be Open to New Ideas</td>
<td>877</td>
</tr>
<tr>
<td>Bring New Ideas to the Table</td>
<td>877</td>
</tr>
<tr>
<td>Recognize What Works and What Doesn’t Work</td>
<td>877</td>
</tr>
<tr>
<td>Don’t Be a Renegade</td>
<td>878</td>
</tr>
<tr>
<td>Source Code Control</td>
<td>878</td>
</tr>
<tr>
<td>Summary</td>
<td>880</td>
</tr>
<tr>
<td><strong>CHAPTER 25: WRITING EFFICIENT C++</strong></td>
<td>881</td>
</tr>
<tr>
<td>Overview of Performance and Efficiency</td>
<td>882</td>
</tr>
<tr>
<td>Two Approaches to Efficiency</td>
<td>882</td>
</tr>
<tr>
<td>Two Kinds of Programs</td>
<td>882</td>
</tr>
<tr>
<td>Is C++ an Inefficient Language?</td>
<td>882</td>
</tr>
<tr>
<td>Language-Level Efficiency</td>
<td>883</td>
</tr>
<tr>
<td>Handle Objects Efficientally</td>
<td>884</td>
</tr>
<tr>
<td>Pass-by-Reference</td>
<td>884</td>
</tr>
<tr>
<td>Return-by-Reference</td>
<td>886</td>
</tr>
<tr>
<td>Catch Exceptions by Reference</td>
<td>886</td>
</tr>
<tr>
<td>Use Move Semantics</td>
<td>886</td>
</tr>
<tr>
<td>Avoid Creating Temporary Objects</td>
<td>886</td>
</tr>
<tr>
<td>The Return-Value Optimization</td>
<td>887</td>
</tr>
<tr>
<td>Pre-allocate Memory</td>
<td>888</td>
</tr>
<tr>
<td>Use Inline Methods and Functions</td>
<td>888</td>
</tr>
<tr>
<td>Design-Level Efficiency</td>
<td>889</td>
</tr>
<tr>
<td>Cache Where Necessary</td>
<td>889</td>
</tr>
<tr>
<td>Use Object Pools</td>
<td>890</td>
</tr>
<tr>
<td>An Object Pool Implementation</td>
<td>891</td>
</tr>
<tr>
<td>Using the Object Pool</td>
<td>893</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling</td>
<td>894</td>
</tr>
<tr>
<td>Profiling Example with gprof</td>
<td>895</td>
</tr>
<tr>
<td>First Design Attempt</td>
<td>895</td>
</tr>
<tr>
<td>Profiling the First Design Attempt</td>
<td>898</td>
</tr>
<tr>
<td>Second Design Attempt</td>
<td>900</td>
</tr>
<tr>
<td>Profiling the Second Design Attempt</td>
<td>901</td>
</tr>
<tr>
<td>Profiling Example with Visual C++ 2017</td>
<td>902</td>
</tr>
<tr>
<td>Summary</td>
<td>907</td>
</tr>
</tbody>
</table>

### CHAPTER 26: BECOMING ADEPT AT TESTING  909

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Control</td>
<td>910</td>
</tr>
<tr>
<td>Whose Responsibility Is Testing?</td>
<td>910</td>
</tr>
<tr>
<td>The Life Cycle of a Bug</td>
<td>910</td>
</tr>
<tr>
<td>Bug-Tracking Tools</td>
<td>912</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>913</td>
</tr>
<tr>
<td>Approaches to Unit Testing</td>
<td>914</td>
</tr>
<tr>
<td>The Unit Testing Process</td>
<td>915</td>
</tr>
<tr>
<td>Define the Granularity of Your Tests</td>
<td>915</td>
</tr>
<tr>
<td>Brainstorm the Individual Tests</td>
<td>917</td>
</tr>
<tr>
<td>Create Sample Data and Results</td>
<td>918</td>
</tr>
<tr>
<td>Write the Tests</td>
<td>918</td>
</tr>
<tr>
<td>Run the Tests</td>
<td>919</td>
</tr>
<tr>
<td>Unit Testing in Action</td>
<td>919</td>
</tr>
<tr>
<td>Introducing the Microsoft Visual C++ Testing Framework</td>
<td>920</td>
</tr>
<tr>
<td>Writing the First Test</td>
<td>921</td>
</tr>
<tr>
<td>Building and Running Tests</td>
<td>922</td>
</tr>
<tr>
<td>Negative Tests</td>
<td>923</td>
</tr>
<tr>
<td>Adding the Real Tests</td>
<td>923</td>
</tr>
<tr>
<td>Debugging Tests</td>
<td>927</td>
</tr>
<tr>
<td>Basking in the Glorious Light of Unit Test Results</td>
<td>927</td>
</tr>
<tr>
<td>Higher-Level Testing</td>
<td>927</td>
</tr>
<tr>
<td>Integration Tests</td>
<td>928</td>
</tr>
<tr>
<td>Sample Integration Tests</td>
<td>928</td>
</tr>
<tr>
<td>Methods of Integration Testing</td>
<td>929</td>
</tr>
<tr>
<td>System Tests</td>
<td>929</td>
</tr>
<tr>
<td>Regression Tests</td>
<td>930</td>
</tr>
<tr>
<td>Tips for Successful Testing</td>
<td>930</td>
</tr>
<tr>
<td>Summary</td>
<td>931</td>
</tr>
</tbody>
</table>
CHAPTER 27: CONQUERING DEBUGGING

The Fundamental Law of Debugging 934
Bug Taxonomies 934
Avoiding Bugs 934
Planning for Bugs 935
   Error Logging 935
   Debug Traces 937
      Debug Mode 937
      Ring Buffers 942
   Assertions 945
   Crash Dumps 946
Static Assertions 947
Debugging Techniques 948
   Reproducing Bugs 948
   Debugging Reproducible Bugs 949
   Debugging Nonreproducible Bugs 950
   Debugging Regressions 951
   Debugging Memory Problems 951
      Categories of Memory Errors 952
      Tips for Debugging Memory Errors 954
   Debugging Multithreaded Programs 956
   Debugging Example: Article Citations 957
      Buggy Implementation of an ArticleCitations Class 957
      Testing the ArticleCitations class 960
      Lessons from the ArticleCitations Example 969
Summary 969

CHAPTER 28: INCORPORATING DESIGN TECHNIQUES AND FRAMEWORKS 971

“I Can Never Remember How to…” 972
   …Write a Class 972
   …Derive from an Existing Class 974
   …Use the Copy-and-Swap Idiom 975
   …Throw and Catch Exceptions 976
   …Read from a File 976
   …Write to a File 977
   …Write a Template Class 977
There Must Be a Better Way 979
   Resource Acquisition Is Initialization 979
   Double Dispatch 981
CONTENTS

<table>
<thead>
<tr>
<th>Attempt #1: Brute Force</th>
<th>982</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempt #2: Single Polymorphism with Overloading</td>
<td>983</td>
</tr>
<tr>
<td>Attempt #3: Double Dispatch</td>
<td>984</td>
</tr>
<tr>
<td>Mixin Classes</td>
<td>985</td>
</tr>
<tr>
<td>Designing a Mixin Class</td>
<td>986</td>
</tr>
<tr>
<td>Implementing a Mixin Class</td>
<td>987</td>
</tr>
<tr>
<td>Using a Mixin Class</td>
<td>988</td>
</tr>
<tr>
<td><strong>Object-Oriented Frameworks</strong></td>
<td>988</td>
</tr>
<tr>
<td>Working with Frameworks</td>
<td>988</td>
</tr>
<tr>
<td>The Model-View-Controller Paradigm</td>
<td>989</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>990</td>
</tr>
</tbody>
</table>

**CHAPTER 29: APPLYING DESIGN PATTERNS** 991

<table>
<thead>
<tr>
<th>The Iterator Pattern</th>
<th>992</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Singleton Pattern</td>
<td>993</td>
</tr>
<tr>
<td>Example: A Logging Mechanism</td>
<td>993</td>
</tr>
<tr>
<td>Implementation of a Singleton</td>
<td>994</td>
</tr>
<tr>
<td>Using a Singleton</td>
<td>997</td>
</tr>
<tr>
<td>The Abstract Factory Pattern</td>
<td>997</td>
</tr>
<tr>
<td>Example: A Car Factory Simulation</td>
<td>998</td>
</tr>
<tr>
<td>Implementation of a Factory</td>
<td>999</td>
</tr>
<tr>
<td>Using a Factory</td>
<td>1002</td>
</tr>
<tr>
<td>Other Uses of Factories</td>
<td>1003</td>
</tr>
<tr>
<td>The Proxy Pattern</td>
<td>1004</td>
</tr>
<tr>
<td>Example: Hiding Network Connectivity Issues</td>
<td>1004</td>
</tr>
<tr>
<td>Implementation of a Proxy</td>
<td>1004</td>
</tr>
<tr>
<td>Using a Proxy</td>
<td>1005</td>
</tr>
<tr>
<td>The Adaptor Pattern</td>
<td>1006</td>
</tr>
<tr>
<td>Example: Adapting a Logger Class</td>
<td>1006</td>
</tr>
<tr>
<td>Implementation of an Adaptor</td>
<td>1007</td>
</tr>
<tr>
<td>Using an Adaptor</td>
<td>1008</td>
</tr>
<tr>
<td>The Decorator Pattern</td>
<td>1008</td>
</tr>
<tr>
<td>Example: Defining Styles in Web Pages</td>
<td>1008</td>
</tr>
<tr>
<td>Implementation of a Decorator</td>
<td>1009</td>
</tr>
<tr>
<td>Using a Decorator</td>
<td>1010</td>
</tr>
<tr>
<td>The Chain of Responsibility Pattern</td>
<td>1010</td>
</tr>
<tr>
<td>Example: Event Handling</td>
<td>1011</td>
</tr>
<tr>
<td>Implementation of a Chain of Responsibility</td>
<td>1011</td>
</tr>
<tr>
<td>Chain of Responsibility without Hierarchy</td>
<td>1012</td>
</tr>
<tr>
<td>The Observer Pattern</td>
<td>1014</td>
</tr>
<tr>
<td>Implementation of an Observer</td>
<td>1014</td>
</tr>
</tbody>
</table>
### Implementation of an Observable  
Using an Observer  
Summary

**CHAPTER 30: DEVELOPING CROSS-PLATFORM AND CROSS-LANGUAGE APPLICATIONS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Platform Development</td>
<td>1018</td>
</tr>
<tr>
<td>Architecture Issues</td>
<td>1018</td>
</tr>
<tr>
<td>Size of Integers</td>
<td>1018</td>
</tr>
<tr>
<td>Binary Compatibility</td>
<td>1019</td>
</tr>
<tr>
<td>Address Sizes</td>
<td>1020</td>
</tr>
<tr>
<td>Byte Order</td>
<td>1020</td>
</tr>
<tr>
<td>Implementation Issues</td>
<td>1021</td>
</tr>
<tr>
<td>Compiler Quirks and Extensions</td>
<td>1021</td>
</tr>
<tr>
<td>Library Implementations</td>
<td>1022</td>
</tr>
<tr>
<td>Platform-Specific Features</td>
<td>1022</td>
</tr>
<tr>
<td>Cross-Language Development</td>
<td>1024</td>
</tr>
<tr>
<td>Mixing C and C++</td>
<td>1024</td>
</tr>
<tr>
<td>Shifting Paradigms</td>
<td>1024</td>
</tr>
<tr>
<td>Linking with C Code</td>
<td>1028</td>
</tr>
<tr>
<td>Calling C++ Code from C#</td>
<td>1030</td>
</tr>
<tr>
<td>Calling C++ Code from Java with JNI</td>
<td>1031</td>
</tr>
<tr>
<td>Calling Scripts from C++ Code</td>
<td>1033</td>
</tr>
<tr>
<td>Calling C++ Code from Scripts</td>
<td>1034</td>
</tr>
<tr>
<td>A Practical Example: Encrypting Passwords</td>
<td>1034</td>
</tr>
<tr>
<td>Calling Assembly Code from C++</td>
<td>1036</td>
</tr>
<tr>
<td>Summary</td>
<td>1038</td>
</tr>
</tbody>
</table>

**APPENDIX A: C++ INTERVIEWS**

**APPENDIX B: ANNOTATED BIBLIOGRAPHY**

**APPENDIX C: STANDARD LIBRARY HEADER FILES**

**APPENDIX D: INTRODUCTION TO UML**

**INDEX**
INTRODUCTION

For many years, C++ has served as the de facto language for writing fast, powerful, and enterprise-class object-oriented programs. As popular as C++ has become, the language is surprisingly difficult to grasp in full. There are simple, but powerful, techniques that professional C++ programmers use that don’t show up in traditional texts, and there are useful parts of C++ that remain a mystery even to experienced C++ programmers.

Too often, programming books focus on the syntax of the language instead of its real-world use. The typical C++ text introduces a major part of the language in each chapter, explaining the syntax and providing an example. Professional C++ does not follow this pattern. Instead of giving you just the nuts and bolts of the language with little practical context, this book will teach you how to use C++ in the real world. It will show you the little-known features that will make your life easier, and the programming techniques that separate novices from professional programmers.

WHO THIS BOOK IS FOR

Even if you have used the language for years, you might still be unfamiliar with the more-advanced features of C++, or you might not be using the full capabilities of the language. Perhaps you write competent C++ code, but would like to learn more about design and good programming style in C++. Or maybe you’re relatively new to C++, but want to learn the “right” way to program from the start. This book will meet those needs and bring your C++ skills to the professional level.

Because this book focuses on advancing from basic or intermediate knowledge of C++ to becoming a professional C++ programmer, it assumes that you have some knowledge of the language. Chapter 1 covers the basics of C++ as a refresher, but it is not a substitute for actual training and use of the language. If you are just starting with C++, but you have significant experience in another programming language such as C, Java, or C#, you should be able to pick up most of what you need from Chapter 1.

In any case, you should have a solid foundation in programming fundamentals. You should know about loops, functions, and variables. You should know how to structure a program, and you should be familiar with fundamental techniques such as recursion. You should have some knowledge of common data structures such as queues, and useful algorithms such as sorting and searching. You don’t need to know about object-oriented programming just yet—that is covered in Chapter 5.

You will also need to be familiar with the compiler you will be using to develop your code. Two compilers, Microsoft Visual C++ and GCC, are introduced later in this introduction. For other compilers, refer to the documentation that came with your compiler.
INTRODUCTION

WHAT THIS BOOK COVERS

Professional C++ uses an approach to C++ programming that will both increase the quality of your code and improve your programming efficiency. You will find discussions on new C++17 features throughout this fourth edition. These features are not just isolated to a few chapters or sections; instead, examples have been updated to use new features when appropriate.

Professional C++ teaches you more than just the syntax and language features of C++. It also emphasizes programming methodologies, reusable design patterns, and good programming style. The Professional C++ methodology incorporates the entire software development process, from designing and writing code, to debugging, and working in groups. This approach will enable you to master the C++ language and its idiosyncrasies, as well as take advantage of its powerful capabilities for large-scale software development.

Imagine users who have learned all of the syntax of C++ without seeing a single example of its use. They know just enough to be dangerous! Without examples, they might assume that all code should go in the main() function of the program, or that all variables should be global—practices that are generally not considered hallmarks of good programming.

Professional C++ programmers understand the correct way to use the language, in addition to the syntax. They recognize the importance of good design, the theories of object-oriented programming, and the best ways to use existing libraries. They have also developed an arsenal of useful code and reusable ideas.

By reading and understanding this book, you will become a professional C++ programmer. You will expand your knowledge of C++ to cover lesser-known and often misunderstood language features. You will gain an appreciation for object-oriented design, and acquire top-notch debugging skills. Perhaps most important, you will finish this book armed with a wealth of reusable ideas that you can actually apply to your daily work.

There are many good reasons to make the effort to be a professional C++ programmer, as opposed to a programmer who knows C++. Understanding the true workings of the language will improve the quality of your code. Learning about different programming methodologies and processes will help you to work better with your team. Discovering reusable libraries and common design patterns will improve your daily efficiency and help you stop reinventing the wheel. All of these lessons will make you a better programmer and a more valuable employee. While this book can’t guarantee you a promotion, it certainly won’t hurt.

HOW THIS BOOK IS STRUCTURED

This book is made up of five parts.

Part I, “Introduction to Professional C++,” begins with a crash course in C++ basics to ensure a foundation of C++ knowledge. Following the crash course, Part I goes deeper into working with strings and string views because strings are used extensively in most examples throughout the book. The last chapter of Part I explores how to write readable C++ code.
Part II, “Professional C++ Software Design,” discusses C++ design methodologies. You will read about the importance of design, the object-oriented methodology, and the importance of code reuse.

Part III, “C++ Coding the Professional Way,” provides a technical tour of C++ from the professional point of view. You will read about the best ways to manage memory in C++, how to create reusable classes, and how to leverage important language features such as inheritance. You will also learn about the unusual and quirky parts of the language, techniques for input and output, error handling, string localization, and how to work with regular expressions. You will read about how to implement operator overloading, and how to write templates. This part also explains the C++ Standard Library, including containers, iterators, and algorithms. You will also read about some additional libraries that are available in the standard, such as the libraries to work with time, random numbers, and the filesystem.

Part IV, “Mastering Advanced Features of C++,” demonstrates how you can get the most out of C++. This part of the book exposes the mysteries of C++ and describes how to use some of its more-advanced features. You will read about how to customize and extend the C++ Standard Library to your needs, advanced details on template programming, including template metaprogramming, and how to use multithreading to take advantage of multiprocessor and multicore systems.

Part V, “C++ Software Engineering,” focuses on writing enterprise-quality software. You’ll read about the engineering practices being used by programming organizations today; how to write efficient C++ code; software testing concepts, such as unit testing and regression testing; techniques used to debug C++ programs; how to incorporate design techniques, frameworks, and conceptual object-oriented design patterns into your own code; and solutions for cross-language and cross-platform code.

The book concludes with a useful chapter-by-chapter guide to succeeding in a C++ technical interview, an annotated bibliography, a summary of the C++ header files available in the standard, and a brief introduction to the Unified Modeling Language (UML).

This book is not a reference of every single class, method, and function available in C++. The book C++ Standard Library Quick Reference by Peter Van Weert and Marc Gregoire\(^1\) is a condensed reference to all essential data structures, algorithms, and functions provided by the C++ Standard Library. Appendix B lists a couple more references. Two excellent online references are:

- [www.cppreference.com](http://www.cppreference.com)
  You can use this reference online, or download an offline version for use when you are not connected to the Internet.


When I refer to a “Standard Library Reference” in this book, I am referring to one of these detailed C++ references.

WHAT YOU NEED TO USE THIS BOOK

All you need to use this book is a computer with a C++ compiler. This book focuses only on parts of C++ that have been standardized, and not on vendor-specific compiler extensions.

Note that this book includes new features introduced with the C++17 standard. At the time of this writing, some compilers are not yet fully C++17 compliant.

You can use whichever C++ compiler you like. If you don’t have a C++ compiler yet, you can download one for free. There are a lot of choices. For example, for Windows, you can download Microsoft Visual Studio 2017 Community Edition, which is free and includes Visual C++. For Linux, you can use GCC or Clang, which are also free.

The following two sections briefly explain how to use Visual C++ and GCC. Refer to the documentation that came with your compiler for more details.

Microsoft Visual C++

First, you need to create a project. Start Visual C++ and click File ➤ New ➤ Project. In the project template tree on the left, select Visual C++ ➤ Win32 (or Windows Desktop). Then select the Win32 Console Application (or Windows Console Application) template in the list in the middle of the window. At the bottom, specify a name for the project and a location where to save it, and click OK.

A wizard opens. In this wizard, click Next, select Console Application, Empty Project, and click Finish.

Once your new project is loaded, you can see a list of project files in the Solution Explorer. If this docking window is not visible, go to View ➤ Solution Explorer. You can add new files or existing files to a project by right-clicking the project name in the Solution Explorer and then clicking Add ➤ New Item or Add ➤ Existing Item.

Use Build ➤ Build Solution to compile your code. When it compiles without errors, you can run it with Debug ➤ Start Debugging.

If your program exits before you have a chance to view the output, use Debug ➤ Start without Debugging. This adds a pause to the end of the program so you can view the output.

At the time of this writing, Visual C++ 2017 does not yet automatically enable C++17 features. To enable C++17 features, in the Solution Explorer window, right-click your project and click Properties. In the properties window, go to Configuration Properties ➤ C/C++ ➤ Language, and set the C++ Language Standard option to “ISO C++17 Standard” or “ISO C++ Latest Draft Standard,” whichever is available in your version of Visual C++. These options are only accessible if your project contains at least one .cpp file.

Visual C++ supports so-called precompiled headers, a topic outside the scope of this book. In general, I recommend using precompiled headers if your compiler supports them. However, the source

---

2Depending on your version of VC++ 2017, you might not see any wizard. Instead, a new project will be created automatically containing four files: stdafx.h, stdafx.cpp, targetver.h, and <projectname>.cpp. If that is the case, and you want to compile source code files from the downloadable source archive for this book, then you have to select those files in the Solution Explorer (View ➤ Solution Explorer) and delete them.
code files in the downloadable source code archive do not use precompiled headers, so you have to disable that feature for them to compile without errors. In the Solution Explorer window, right-click your project and click Properties. In the properties window, go to Configuration Properties ➪ C/C++ ➪ Precompiled Headers, and set the Precompiled Header option to “Not Using Precompiled Headers.”

**GCC**

Create your source code files with any text editor you prefer and save them to a directory. To compile your code, open a terminal and run the following command, specifying all your `.cpp` files that you want to compile:

```
gcc -lstdc++ -std=c++17 -o <executable_name> <source1.cpp> [source2.cpp ...]
```

The `-std=c++17` option is required to tell GCC to enable C++17 support.

For example, you can compile the `AirlineTicket` example from Chapter 1 by changing to the directory containing the code and running the following command:

```
gcc -lstdc++ -std=c++17 -o AirlineTicket AirlineTicket.cpp AirlineTicketTest.cpp
```

When it compiles without errors, you can run it as follows:

```
./AirlineTicket
```

**CONVENTIONS**

To help you get the most from the text and keep track of what’s happening, a number of conventions are used throughout this book.

**WARNING** Boxes like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.

**NOTE** Tips, hints, tricks, and asides to the current discussion are placed in boxes like this one.

As for styles in the text:

- Important words are *highlighted* when they are introduced.
- Keyboard strokes are shown like this: Ctrl+A.
- Filenames and code within the text are shown like so: `monkey.cpp`. 
**INTRODUCTION**

URLs are shown like this: www.wrox.com.

Code is presented in three different ways:

// Comments in code are shown like this.
In code examples, new and important code is highlighted like this.
Code that's less important in the present context or that has been shown before is formatted like this.

Paragraphs or sections that are specific to the C++17 standard have a little C++17 icon on the left, just as this paragraph does. C++11 and C++14 features are not marked with any icon.

**SOURCE CODE**

As you work through the examples in this book, you may choose either to type in all the code manually, or to use the source code files that accompany the book. However, I suggest you type in all the code manually because it greatly benefits the learning process and your memory. All of the source code used in this book is available for download at www.wiley.com/go/proc++4e.

Alternatively, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code that is available for this book and all other Wrox books.

NOTE  Because many books have similar titles, you may find it easiest to search by ISBN; for this book, the ISBN is 978-1-119-42130-6.

Once you’ve downloaded the code, just decompress it with your favorite decompression tool.

**ERRATA**

At Wrox, we make every effort to ensure that there are no errors in the text or in the code of our books. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wrox.com and locate the title by using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list, including links to each book’s errata, is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don’t spot “your” error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s errata page and fix the problem in subsequent editions of the book.
PART I
Introduction to Professional C++

▸ CHAPTER 1: A Crash Course in C++ and the Standard Library
▸ CHAPTER 2: Working with Strings and String Views
▸ CHAPTER 3: Coding with Style
A Crash Course in C++ and the Standard Library

WHAT’S IN THIS CHAPTER?

➤ A brief overview of the most important parts and syntax of the C++ language and the Standard Library
➤ The basics of smart pointers

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of the chapter’s code download on this book’s website at www.wrox.com/go/proc++4e on the Download Code tab.

The goal of this chapter is to cover briefly the most important parts of C++ so that you have a base of knowledge before embarking on the rest of this book. This chapter is not a comprehensive lesson in the C++ programming language or the Standard Library. Certain basic points, such as what a program is and what recursion is, are not covered. Esoteric points, such as the definition of a union, or the volatile keyword, are also omitted. Certain parts of the C language that are less relevant in C++ are also left out, as are parts of C++ that get in-depth coverage in later chapters.

This chapter aims to cover the parts of C++ that programmers encounter every day. For example, if you’ve been away from C++ for a while and you’ve forgotten the syntax of a for loop, you’ll find that syntax in this chapter. Also, if you’re fairly new to C++ and don’t understand what a reference variable is, you’ll learn about that kind of variable here, as well. You’ll also learn the basics on how to use the functionality available in the Standard Library, such as vector containers, string objects, and smart pointers.
CHAPTER 1  A CRASH COURSE IN C++ AND THE STANDARD LIBRARY

If you already have significant experience with C++, skim this chapter to make sure that there aren’t any fundamental parts of the language on which you need to brush up. If you’re new to C++, read this chapter carefully and make sure you understand the examples. If you need additional introductory information, consult the titles listed in Appendix B.

THE BASICS OF C++

The C++ language is often viewed as a “better C” or a “superset of C.” It was mainly designed to be an object-oriented C, commonly called as “C with classes.” Later on, many of the annoyances and rough edges of the C language were addressed as well. Because C++ is based on C, much of the syntax you’ll see in this section will look familiar to you if you are an experienced C programmer. The two languages certainly have their differences, though. As evidence, The C++ Programming Language by C++ creator Bjarne Stroustrup (Fourth Edition; Addison-Wesley Professional, 2013) weighs in at 1,368 pages, while Kernighan and Ritchie’s The C Programming Language (Second Edition; Prentice Hall, 1988) is a scant 274 pages. So, if you’re a C programmer, be on the lookout for new or unfamiliar syntax!

The Obligatory Hello, World

In all its glory, the following code is the simplest C++ program you’re likely to encounter:

```cpp
// helloworld.cpp
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

This code, as you might expect, prints the message, “Hello, World!” on the screen. It is a simple program and unlikely to win any awards, but it does exhibit the following important concepts about the format of a C++ program:

- Comments
- Preprocessor directives
- The main() function
- I/O streams

These concepts are briefly explained in the following sections.

Comments

The first line of the program is a comment, a message that exists for the programmer only and is ignored by the compiler. In C++, there are two ways to delineate a comment. In the preceding and following examples, two slashes indicate that whatever follows on that line is a comment.

```cpp
// helloworld.cpp
```
The same behavior (this is to say, none) would be achieved by using a *multiline comment*. Multiline comments start with /* and end with */. The following code shows a multiline comment in action (or, more appropriately, inaction).

```cpp
/* This is a multiline comment. 
The compiler will ignore it. */
```

Comments are covered in detail in Chapter 3.

**Preprocessor Directives**

Building a C++ program is a three-step process. First, the code is run through a *preprocessor*, which recognizes meta-information about the code. Next, the code is *compiled*, or translated into machine-readable object files. Finally, the individual object files are *linked* together into a single application.

Directives aimed at the preprocessor start with the # character, as in the line `#include <iostream>` in the previous example. In this case, an `#include` directive tells the preprocessor to take everything from the `<iostream>` header file and make it available to the current file. The most common use of header files is to declare functions that will be defined elsewhere. A function *declaration* tells the compiler how a function is called, declaring the number and types of parameters, and the function return type. A *definition* contains the actual code for the function. In C++, declarations usually go into *header files*, typically with extension .h, while definitions usually go into *source files*, typically with extension .cpp. A lot of other programming languages, such as C# and Java, do not separate declarations and definitions into separate files.

The `<iostream>` header declares the input and output mechanisms provided by C++. If the program did not include that header, it would be unable to perform its only task of outputting text.

**NOTE** In C, the names of the Standard Library header files usually end in .h, such as `<stdio.h>`, and namespaces are not used.

In C++, the .h suffix is omitted for Standard Library headers, such as `<iostream>`, and everything is defined in the `std` namespace or a sub-namespace of `std`.

The Standard Library headers from C still exist in C++ but in two versions:

- The new and recommended versions without a .h suffix but with a c pre-fix. These versions put everything in the `std` namespace (for example, `<cstdio>`).
- The old versions with the .h suffix. These versions do not use namespaces (for example, `<stdio.h>`).
The following table shows some of the most common preprocessor directives.

<table>
<thead>
<tr>
<th>PREPROCESSOR DIRECTIVE</th>
<th>FUNCTIONALITY</th>
<th>COMMON USES</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include [file]</td>
<td>The specified file is inserted into the code at the location of the directive.</td>
<td>Almost always used to include header files so that code can make use of functionality defined elsewhere.</td>
</tr>
<tr>
<td>#define [key] [value]</td>
<td>Every occurrence of the specified key is replaced with the specified value.</td>
<td>Often used in C to define a constant value or a macro. C++ provides better mechanisms for constants and most types of macros. Macros can be dangerous, so use them cautiously. See Chapter 11 for details.</td>
</tr>
<tr>
<td>#ifdef [key] #endif</td>
<td>Code within the ifdef (&quot;if defined&quot;) or ifndef (&quot;if not defined&quot;) blocks are conditionally included or omitted based on whether the specified key has been defined with #define.</td>
<td>Used most frequently to protect against circular includes. Each header file starts with an ifndef checking the absence of a key, followed by a #define directive to define that key. The header file ends with an #endif. This prevents the file from being included multiple times; see the example after this table.</td>
</tr>
<tr>
<td>#ifndef [key] #endif</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma [xyz]</td>
<td>xyz is compiler dependent. It often allows the programmer to display a warning or error if the directive is reached during preprocessing.</td>
<td>See the example after this table.</td>
</tr>
</tbody>
</table>

One example of using preprocessor directives is to avoid multiple includes, as shown here:

```c
#ifdef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif
```

If your compiler supports the #pragma once directive, and most modern compilers do, then this can be rewritten as follows:

```c
#pragma once
// ... the contents of this header file
```

Chapter 11 discusses this in more details.

**The main() Function**

`main()` is, of course, where the program starts. The return type of `main()` is an `int`, indicating the result status of the program. You can omit any explicit return statements in `main()`, in which
case zero is returned automatically. The `main()` function either takes no parameters, or takes two parameters as follows:

```cpp
int main(int argc, char* argv[])
```

`argc` gives the number of arguments passed to the program, and `argv` contains those arguments. Note that `argv[0]` can be the program name, but it might as well be an empty string, so do not rely on it; instead, use platform-specific functionality to retrieve the program name. The important thing to remember is that the actual parameters start at index 1.

I/O Streams

I/O streams are covered in depth in Chapter 13, but the basics of output and input are very simple. Think of an output stream as a laundry chute for data. Anything you toss into it will be output appropriately. `std::cout` is the chute corresponding to the user console, or *standard out*. There are other chutes, including `std::cerr`, which outputs to the error console. The `<<` operator tosses data down the chute. In the preceding example, a quoted string of text is sent to standard out. Output streams allow multiple types of data to be sent down the stream sequentially on a single line of code. The following code outputs text, followed by a number, followed by more text:

```cpp
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

`std::endl` represents an end-of-line sequence. When the output stream encounters `std::endl`, it will output everything that has been sent down the chute so far and move to the next line. An alternate way of representing the end of a line is by using the `\n` character. The `\n` character is an *escape sequence*, which refers to a new-line character. Escape sequences can be used within any quoted string of text. The following table shows the most common ones:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>new line</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>carriage return</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>tab</td>
</tr>
<tr>
<td><code>\\</code></td>
<td>backslash character</td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td>quotation mark</td>
</tr>
</tbody>
</table>

Streams can also be used to accept input from the user. The simplest way to do this is to use the `>>` operator with an input stream. The `std::cin` input stream accepts keyboard input from the user. Here is an example:

```cpp
int value;
std::cin >> value;
```

User input can be tricky because you can never know what kind of data the user will enter. See Chapter 13 for a full explanation of how to use input streams.

If you're new to C++ and coming from a C background, you're probably wondering what has been done with the trusty old `printf()` and `scanf()` functions. While these functions can still be used in C++, I recommend using the streams library instead, mainly because the `printf()` and `scanf()` family of functions do not provide any type safety.
Namespaces

Namespaces address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code. You can’t change the library’s function name, and it would be a big pain to change your own.

Namespaces come to the rescue in such scenarios because you can define the context in which names are defined. To place code in a namespace, enclose it within a namespace block. For example, the following could be the contents of a file called `namespaces.h`:

```cpp
namespace mycode {
    void foo();
}
```

The implementation of a method or function can also be handled in a namespace. The `foo()` function, for instance, could be implemented in `namespaces.cpp` as follows:

```cpp
#include <iostream>
#include "namespaces.h"

void mycode::foo()
{
    std::cout << "foo() called in the mycode namespace" << std::endl;
}
```

Or alternatively:

```cpp
#include <iostream>
#include "namespaces.h"

namespace mycode {
    void foo()
    {
        std::cout << "foo() called in the mycode namespace" << std::endl;
    }
}
```

By placing your version of `foo()` in the namespace “mycode,” you are isolating it from the `foo()` function provided by the third-party library. To call the namespace-enabled version of `foo()`, prepend the namespace onto the function name by using `::`, also called the scope resolution operator, as follows:

```
mycode::foo();    // Calls the "foo" function in the "mycode" namespace
```

Any code that falls within a “mycode” namespace block can call other code within the same namespace without explicitly prepending the namespace. This implicit namespace is useful in making the code more readable. You can also avoid prepending of namespaces with the using directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the code that follows:

```cpp
#include "namespaces.h"

using namespace mycode;
```
```cpp
int main()
{
    foo(); // Implies mycode::foo();
    return 0;
}
```

A single source file can contain multiple `using` directives, but beware of overusing this shortcut. In the extreme case, if you declare that you're using every namespace known to humanity, you're effectively eliminating namespaces entirely! Name conflicts will again result if you are using two namespaces that contain the same names. It is also important to know in which namespace your code is operating so that you don't end up accidentally calling the wrong version of a function.

You've seen the namespace syntax before—you used it in the Hello, World program, where `cout` and `endl` are actually names defined in the `std` namespace. You could have written Hello, World with the `using` directive as shown here:

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

A `using` declaration can be used to refer to a particular item within a namespace. For example, if the only part of the `std` namespace that you intend to use is `cout`, you can refer to it as follows:

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

Subsequent code can refer to `cout` without prepending the namespace, but other items in the `std` namespace will still need to be explicit:

```cpp
using std::cout;
cout << "Hello, World!" << std::endl;
```

**WARNING** Never put a `using` directive or `using` declaration in a header file; otherwise, you force it on everyone who is including your header file.

C++17 makes it easier to work with nested namespaces. A nested namespace is a namespace inside another one. Before C++17, you had to use nested namespaces as follows:

```cpp
namespace MyLibraries {
    namespace Networking {
        namespace FTP {
            /* ... */
        }
    }
}
```
This can be simplified a lot with C++17:

```cpp
namespace MyLibraries::Networking::FTP {
    /* ... */
}
```

A namespace alias can be used to give a new and possibly shorter name to another namespace. For example:

```cpp
namespace MyFTP = MyLibraries::Networking::FTP;
```

### Literals

Literals are used to write numbers or strings in your code. C++ supports a number of standard literals. Numbers can be specified with the following literals (the examples in the list represent the same number, 123):

- Decimal literal, 123
- Octal literal, 0173
- Hexadecimal literal, 0x7B
- Binary literal, 0b1111011

Other examples of literals in C++ include:

- A floating-point value (such as 3.14f)
- A double floating-point value (such as 3.14)
- A single character (such as 'a')
- A zero-terminated array of characters (such as "character array")

It is also possible to define your own type of literals, which is an advanced feature explained in Chapter 11.

Digits separators can be used in numeric literals. A digits separator is a single quote character. For example,

- 23'456'789
- 0.123'456f

C++17 adds support for hexadecimal floating-point literals—for example, 0x3.ABCp-10, 0Xb.cp12l.

### Variables

In C++, variables can be declared just about anywhere in your code and can be used anywhere in the current block below the line where they are declared. Variables can be declared without being given a value. These uninitialized variables generally end up with a semi-random value based on whatever is in memory at that time, and are therefore the source of countless bugs. Variables in C++ can alternatively be assigned an initial value when they are declared. The code that follows shows both flavors of variable declaration, both using ints, which represent integer values.

```cpp
int uninitializedInt;
int initializedInt = 7;
cout << uninitializedInt << " is a random value" << endl;
cout << initializedInt << " was assigned an initial value" << endl;
```
The following table shows the most common types used in C++.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(signed) int</td>
<td>Positive and negative integers; the range depends on</td>
<td></td>
</tr>
<tr>
<td>signed</td>
<td>the compiler (usually 4 bytes).</td>
<td>int i = -7;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed int i = -6;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed i = -5;</td>
</tr>
<tr>
<td>(signed) short (int)</td>
<td>Short integer (usually 2 bytes)</td>
<td>short s = 13;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short int s = 14;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed short s = 15;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed short int s = 16;</td>
</tr>
<tr>
<td>(signed) long (int)</td>
<td>Long integer (usually 4 bytes)</td>
<td>long l = -7L;</td>
</tr>
<tr>
<td>(signed) long long (int)</td>
<td>Long long integer; the range</td>
<td>long long ll = 14LL;</td>
</tr>
<tr>
<td></td>
<td>depends on the compiler, but is at least the same</td>
<td></td>
</tr>
<tr>
<td></td>
<td>as for long (usually 8 bytes).</td>
<td></td>
</tr>
<tr>
<td>unsigned (int)</td>
<td>Limits the preceding types to values &gt;= 0</td>
<td>unsigned int i = 2U;</td>
</tr>
<tr>
<td>unsigned short (int)</td>
<td></td>
<td>unsigned j = 5U;</td>
</tr>
<tr>
<td>unsigned long (int)</td>
<td></td>
<td>unsigned short s = 23U;</td>
</tr>
<tr>
<td>unsigned long long (int)</td>
<td></td>
<td>unsigned long l = 5400UL;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long ll = 140ULL;</td>
</tr>
<tr>
<td>float</td>
<td>Floating-point numbers</td>
<td>float f = 7.2f;</td>
</tr>
<tr>
<td>double</td>
<td>Double precision numbers; precision is at least the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>same as for float.</td>
<td>double d = 7.2;</td>
</tr>
<tr>
<td>long double</td>
<td>Long double precision numbers; precision is at least</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the same as for double.</td>
<td>long double d = 16.98L;</td>
</tr>
<tr>
<td>char</td>
<td>A single character</td>
<td>char ch = 'm';</td>
</tr>
<tr>
<td>char16_t</td>
<td>A single 16-bit character</td>
<td>char16_t c16 = u'm';</td>
</tr>
<tr>
<td>char32_t</td>
<td>A single 32-bit character</td>
<td>char32_t c32 = U'm';</td>
</tr>
</tbody>
</table>

NOTE: Most compilers will issue a warning or an error when code is using uninitialized variables. Some compilers will generate code that will report an error at run time.

continues
Variables can be converted to other types by casting them. For example, a float can be cast to an int. C++ provides three ways to explicitly change the type of a variable. The first method is a holdover from C; it is not recommended but unfortunately still commonly used. The second method is rarely used. The third method is the most verbose, but is also the cleanest one, and is therefore recommended.

```cpp
float myFloat = 3.14f;
int i1 = (int)myFloat; // method 1
int i2 = int(myFloat); // method 2
int i3 = static_cast<int>(myFloat); // method 3
```

The resulting integer will be the value of the floating-point number with the fractional part truncated. Chapter 11 describes the different casting methods in more detail. In some contexts, variables can be automatically cast, or coerced. For example, a short can be automatically converted into a long because a long represents the same type of data with at least the same precision.

```cpp
long someLong = someShort; // no explicit cast needed
```

When automatically casting variables, you need to be aware of the potential loss of data. For example, casting a float to an int throws away information (the fractional part of the number). Most compilers will issue a warning or even an error if you assign a float to an int without an explicit cast. If you are certain that the left-hand side type is fully compatible with the right-hand side type, it’s okay to cast implicitly.
Operators

What good is a variable if you don’t have a way to change it? The following table shows the most common operators used in C++ and sample code that makes use of them. Note that operators in C++ can be binary (operate on two expressions), unary (operate on a single expression), or even ternary (operate on three expressions). There is only one ternary operator in C++, and it is explained in the “Conditional Statements” section later in this chapter.

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>DESCRIPTION</th>
<th>USAGE</th>
</tr>
</thead>
</table>
| =        | Binary operator to assign the value on the right to the expression on the left | int i;  
i = 3;  
int j;  
j = i; |
| !        | Unary operator to complement the true/false (non-0/0) status of an expression | bool b = !true;  
bool b2 = !b; |
| +        | Binary operator for addition | int i = 3 + 2;  
int j = i + 5;  
int k = i + j; |
| -        | Binary operators for subtraction, multiplication, and division | int i = 5 - 1;  
int j = 5 * 2;  
int k = j / i; |
| *        | Binary operator for the remainder of a division operation. This is also referred to as the mod or modulo operator. | int remainder = 5 % 2; |
| /        | Unary operator to increment an expression by 1. If the operator occurs after the expression, or post-increment, the result of the expression is the unincremented value. If the operator occurs before the expression, or pre-increment, the result of the expression is the new value. | i++;  
++i; |
| %        | Unary operator to decrement an expression by 1 | i--;  
--i; |
| +=       | Shorthand syntax for i = i + j | i += j; |
| -=       | Shorthand syntax for | i -= j; |
| *=       | Shorthand syntax for | i *= j; |
| /=       | Shorthand syntax for | i /= j; |
| %=       | Shorthand syntax for | i %= j; |

continues
OPERATOR | DESCRIPTION | USAGE
---|---|---
& | Takes the raw bits of one expression and performs a bitwise “AND” with the other expression | \( i = j \& k; \)
&= | \( j \&= k; \)
| | Takes the raw bits of one expression and performs a bitwise “OR” with the other expression | \( i = j \mid k; \)
|= | \( j \mid= k; \)
<< | Takes the raw bits of an expression and “shifts” each bit left (<<) or right (>>) the specified number of places | \( i = i \ll 1; \)
| | \( j \ll= 1; \)
| | \( i \ll= 1; \)
| | \( i \gg= 4; \)
^ | Performs a bitwise “exclusive or,” also called “XOR” operation, on two expressions | \( i = i \oplus j; \)
|= | \( i \oplus= j; \)

The following program shows the most common variable types and operators in action. If you are unsure about how variables and operators work, try to figure out what the output of this program will be, and then run it to confirm your answer.

```cpp
int someInteger = 256;
short someShort;
long someLong;
float someFloat;
double someDouble;

someInteger++;
someInteger *= 2;
someShort = static_cast<short>(someInteger);
someLong = someShort * 10000;
someFloat = someLong + 0.785f;
someDouble = static_cast<double>(someFloat) / 100000;
cout << someDouble << endl;
```

The C++ compiler has a recipe for the order in which expressions are evaluated. If you have a complicated line of code with many operators, the order of execution may not be obvious. For that reason, it’s probably better to break up a complicated expression into several smaller expressions, or explicitly group sub-expressions by using parentheses. For example, the following line of code is confusing unless you happen to know the C++ operator precedence table by heart:

```
int i = 34 + 8 * 2 + 21 / 7 % 2;
```

Adding parentheses makes it clear which operations are happening first:

```
int i = 34 + (8 * 2) + ( (21 / 7) % 2 );
```

For those of you playing along at home, both approaches are equivalent and end up with \( i \) equal to 51. If you assumed that C++ evaluated expressions from left to right, your answer would have been 1. C++ evaluates /=, *=, and %= first (in left-to-right order), followed by addition and subtraction, then bitwise operators. Parentheses let you explicitly tell the compiler that a certain operation should be evaluated separately.
Types

In C++, you can use the basic types (int, bool, and so on) to build more complex types of your own design. Once you are an experienced C++ programmer, you will rarely use the following techniques, which are features brought in from C, because classes are far more powerful. Still, it is important to know about the following ways of building types so that you will recognize the syntax.

Enumerated Types

An integer really represents a value within a sequence—the sequence of numbers. Enumerated types let you define your own sequences so that you can declare variables with values in that sequence. For example, in a chess program, you could represent each piece as an int, with constants for the piece types, as shown in the following code. The integers representing the types are marked const to indicate that they can never change.

```cpp
const int PieceTypeKing = 0;
const int PieceTypeQueen = 1;
const int PieceTypeRook = 2;
const int PieceTypePawn = 3;
//etc.
int myPiece = PieceTypeKing;
```

This representation is fine, but it can become dangerous. Since a piece is just an int, what would happen if another programmer added code to increment the value of a piece? By adding 1, a king becomes a queen, which really makes no sense. Worse still, someone could come in and give a piece a value of -1, which has no corresponding constant.

Enumerated types solve these problems by tightly defining the range of values for a variable. The following code declares a new type, PieceType, which has four possible values, representing four of the chess pieces:

```cpp
enum PieceType { PieceTypeKing, PieceTypeQueen, PieceTypeRook, PieceTypePawn };
```

Behind the scenes, an enumerated type is just an integer value. The real value of PieceTypeKing is 0. However, by defining the possible values for variables of type PieceType, your compiler can give you a warning or an error if you attempt to perform arithmetic on PieceType variables or treat them as integers. The following code, which declares a PieceType variable, and then attempts to use it as an integer, results in a warning or an error on most compilers:

```cpp
PieceType myPiece;
myPiece = 0;
```

It’s also possible to specify the integer values for members of an enumeration. The syntax is as follows:

```cpp
enum PieceType { PieceTypeKing = 1, PieceTypeQueen, PieceTypeRook = 10, PieceTypePawn };
```

In this example, PieceTypeKing has the integer value 1, PieceTypeQueen has the value 2 assigned by the compiler, PieceTypeRook has the value 10, and PieceTypePawn has the value 11 assigned automatically by the compiler.
If you do not assign a value to an enumeration member, the compiler automatically assigns it a value that is the previous enumeration member incremented by 1. If you do not assign a value to the first enumeration member yourself, the compiler assigns it the value 0.

**Strongly Typed Enumerations**

Enumerations as explained in the previous section are not strongly typed, meaning they are not *type safe*. They are always interpreted as integers, and thus you can compare enumeration values from completely different enumeration types.

The strongly-typed `enum class` enumerations solve this problem. For example, the following defines a type-safe version of the earlier-defined `PieceType` enumeration:

```cpp
enum class PieceType
{
    King = 1,
    Queen,
    Rook = 10,
    Pawn
};
```

For an `enum class`, the enumeration value names are not automatically exported to the enclosing scope, which means that you always have to use the scope resolution operator:

```cpp
PieceType piece = PieceType::King;
```

This also means that you can give shorter names to the enumeration values, for example, `King` instead of `PieceTypeKing`.

Additionally, the enumeration values are not automatically converted to integers, which means the following is illegal:

```cpp
if (PieceType::Queen == 2) {...}
```

By default, the underlying type of an enumeration value is an integer, but this can be changed as follows:

```cpp
enum class PieceType : unsigned long
{
    King = 1,
    Queen,
    Rook = 10,
    Pawn
};
```

**NOTE** *It is recommended to use the strongly-typed `enum class` enumerations instead of the type-unsafe `enum` enumerations.*

**Structs**

*Structs* let you encapsulate one or more existing types into a new type. The classic example of a struct is a database record. If you are building a personnel system to keep track of employee
information, you might want to store the first initial, last initial, employee number, and salary for each employee. A struct that contains all of this information is shown in the `employeestruct.h` header file that follows:

```cpp
struct Employee {
    char firstInitial;
    char lastInitial;
    int employeeNumber;
    int salary;
};
```

A variable declared with type `Employee` will have all of these fields built in. The individual fields of a struct can be accessed by using the “.” operator. The example that follows creates and then outputs the record for an employee:

```cpp
#include <iostream>
#include "employeestruct.h"

using namespace std;

int main()
{
    // Create and populate an employee.
    Employee anEmployee;
    anEmployee.firstInitial = 'M';
    anEmployee.lastInitial = 'G';
    anEmployee.employeeNumber = 42;
    anEmployee.salary = 80000;
    // Output the values of an employee.
    cout << "Employee: " << anEmployee.firstInitial << 
        anEmployee.lastInitial << endl;
    cout << "Number: " << anEmployee.employeeNumber << endl;
    cout << "Salary: $" << anEmployee.salary << endl;
    return 0;
}
```

### Conditional Statements

Conditional statements let you execute code based on whether or not something is true. As shown in the following sections, there are three main types of conditional statements in C++: if/else statements, switch statements, and conditional operators.

#### if/else Statements

The most common conditional statement is the if statement, which can be accompanied by an else. If the condition given inside the if statement is true, the line or block of code is executed. If not, execution continues with the else case if present, or with the code following the conditional statement. The following code shows a cascading if statement, a fancy way of saying that the if statement has an else statement that in turn has another if statement, and so on:

```cpp
if (i > 4) {
    // Do something.
} else if (i > 2) {
```
The expression between the parentheses of an if statement must be a Boolean value or evaluate to a Boolean value. A value of 0 evaluates to false, while any non-zero value evaluates to true. For example: if(0) is equivalent to if(false). Logical evaluation operators, described later, provide ways of evaluating expressions to result in a true or false Boolean value.

Initializers for if Statements

C++17 allows you to include an initializer inside an if statement using the following syntax:

```cpp
if (<initializer> ; <conditional_expression>) { <body> }
```

Any variable introduced in the <initializer> is only available in the <conditional_expression> and in the <body>. Such variables are not available outside the if statement.

It is too early in this book to give a useful example of this feature, but here is what it looks like:

```cpp
if (Employee employee = GetEmployee() ; employee.salary > 1000) { ... }
```

In this example, the initializer gets an employee and the condition checks whether the salary of the retrieved employee exceeds 1000. Only in that case is the body of the if statement executed.

More concrete examples will be given throughout this book.

switch Statements

The switch statement is an alternate syntax for performing actions based on the value of an expression. In C++, the expression of a switch statement must be of an integral type, a type convertible to an integral type, an enumerated type, or a strongly typed enumeration, and must be compared to constants. Each constant value represents a “case.” If the expression matches the case, the subsequent lines of code are executed until a break statement is reached. You can also provide a default case, which is matched if none of the other cases match. The following pseudocode shows a common use of the switch statement:

```cpp
switch (menuItem) {
    case OpenMenuItem:
        // Code to open a file
        break;
    case SaveMenuItem:
        // Code to save a file
        break;
    default:
        // Code to give an error message
        break;
}
```

A switch statement can always be converted into if/else statements. The previous switch statement can be converted as follows:

```cpp
if (menuItem == OpenMenuItem) {
    // Code to open a file
```
switch statements are generally used when you want to do something based on more than 1 specific value of an expression, as opposed to some test on the expression. In such a case, the switch statement avoids cascading if-else statements. If you only need to inspect 1 value, an if or if-else statement is fine.

Once a case expression matching the switch condition is found, all statements that follow it are executed until a break statement is reached. This execution continues even if another case expression is encountered, which is called fallthrough. The following example has a single set of statements that is executed for several different cases:

```cpp
switch (backgroundColor) {
    case Color::DarkBlue:
    case Color::Black:
        // Code to execute for both a dark blue or black background color
        break;
    case Color::Red:
        // Code to execute for a red background color
        break;
}
```

Fallthrough can be a source of bugs, for example if you accidentally forget a break statement. Because of this, compilers might give a warning if a fallthrough is detected in a switch statement, unless the case is empty as in the above example. Starting with C++17, you can tell the compiler that a fallthrough is intentional using the `[[fallthrough]]` attribute as follows:

```cpp
switch (backgroundColor) {
    case Color::DarkBlue:
        doSomethingForDarkBlue();
        [[fallthrough]];  // Code is executed for both a dark blue or black background color
    case Color::Black:
        doSomethingForBlackOrDarkBlue();
        break;
    case Color::Red:
    case Color::Green:
        // Code to execute for a red or green background color
        break;
}
```

Initializers for switch Statements

Just as for if statements, C++17 adds support for initializers to switch statements. The syntax is as follows:

```cpp
switch (<initializer> ; <expression>) { <body> }
```

Any variables introduced in the `<initializer>` are only available in the `<expression>` and in the `<body>`. They are not available outside the switch statement.
The Conditional Operator

C++ has one operator that takes three arguments, known as a ternary operator. It is used as a shorthand conditional expression of the form “if [something] then [perform action], otherwise [perform some other action].” The conditional operator is represented by a `?` and a `:`. The following code outputs “yes” if the variable `i` is greater than 2, and “no” otherwise:

```cpp
std::cout << ((i > 2) ? "yes" : "no");
```

The parentheses around `i > 2` are optional, so the following is equivalent:

```cpp
std::cout << (i > 2 ? "yes" : "no");
```

The advantage of the conditional operator is that it can occur within almost any context. In the preceding example, the conditional operator is used within code that performs output. A convenient way to remember how the syntax is used is to treat the question mark as though the statement that comes before it really is a question. For example, “Is `i` greater than 2? If so, the result is ‘yes’; if not, the result is ‘no.’”

Unlike an `if` statement or a `switch` statement, the conditional operator doesn’t execute code blocks based on the result. Instead, it is used within code, as shown in the preceding example. In this way, it really is an operator (like `+` and `-`) as opposed to a true conditional statement, such as `if` and `switch`.

Logical Evaluation Operators

You have already seen a logical evaluation operator without a formal definition. The `>` operator compares two values. The result is “true” if the value on the left is greater than the value on the right. All logical evaluation operators follow this pattern—they all result in a `true` or `false`.

The following table shows common logical evaluation operators:

<table>
<thead>
<tr>
<th>OP</th>
<th>DESCRIPTION</th>
<th>USAGE</th>
</tr>
</thead>
</table>
| `<` | Determines if the left-hand side is less than, less than or equal to, greater than, or greater than or equal to the right-hand side | if (i < 0) {
   std::cout << "i is negative";
   }
| `<=` | | |
| `>` | | |
| `>=` | | |
| `==` | Determines if the left-hand side equals the right-hand side. Don’t confuse this with the `= (assignment)` operator! | if (i == 3) {
   std::cout << "i is 3";
   }
| `!=` | Not equals. The result of the statement is true if the left-hand side does not equal the right-hand side. | if (i != 3) {
   std::cout << "i is not 3";
   } |
**OP** | **DESCRIPTION** | **USAGE**
--- | --- | ---
! | Logical NOT. This complements the true/false status of a Boolean expression. This is a unary operator. | if (!someBoolean) {
    std::cout << "someBoolean is false";
}

&;& | Logical AND. The result is true if both parts of the expression are true. | if (someBoolean && someOtherBoolean) {
    std::cout << "both are true";
}

|| | Logical OR. The result is true if either part of the expression is true. | if (someBoolean || someOtherBoolean) {
    std::cout << "at least one is true";
}

C++ uses *short-circuit logic* when evaluating logical expressions. That means that once the final result is certain, the rest of the expression won’t be evaluated. For example, if you are performing a logical OR operation of several Boolean expressions, as shown in the following code, the result is known to be true as soon as one of them is found to be true. The rest won’t even be checked.

```
bool result = bool1 || bool2 || (i > 7) || (27 / 13 % i + 1) < 2;
```

In this example, if *bool1* is found to be true, the entire expression must be true, so the other parts aren’t evaluated. In this way, the language saves your code from doing unnecessary work. It can, however, be a source of hard-to-find bugs if the later expressions in some way influence the state of the program (for example, by calling a separate function). The following code shows a statement using && that short-circuits after the second term because 0 always evaluates to false:

```
bool result = bool1 && 0 && (i > 7) && !done;
```

Short-circuiting can be beneficial for performance. You can put cheaper tests first so that more expensive tests are not even executed when the logic short-circuits. It is also useful in the context of pointers to avoid parts of the expression to be executed when a pointer is not valid. Pointers and short-circuiting with pointers are discussed later in this chapter.

### Functions

For programs of any significant size, placing all the code inside of `main()` is unmanageable. To make programs easy to understand, you need to break up, or *decompose*, code into concise functions.

In C++, you first declare a function to make it available for other code to use. If the function is used inside only a particular file, you generally declare and define the function in the source file. If the function is for use by other modules or files, you generally put the declaration in a header file and the definition in a source file.
A function declaration is shown in the following code. This example has a return type of `void`, indicating that the function does not provide a result to the caller. The caller must provide two arguments for the function to work with—an integer and a character.

```cpp
void myFunction(int i, char c);
```

Without an actual definition to match this function declaration, the link stage of the compilation process will fail because code that makes use of the function will be calling nonexistent code. The following definition prints the values of the two parameters:

```cpp
void myFunction(int i, char c)
{
    std::cout << "the value of i is " << i << std::endl;
    std::cout << "the value of c is " << c << std::endl;
}
```

Elsewhere in the program, you can make calls to `myFunction()` and pass in arguments for the two parameters. Some sample function calls are shown here:

```cpp
myFunction(8, 'a');
myFunction(someInt, 'b');
myFunction(5, someChar);
```

C++ functions can also return a value to the caller. The following function adds two numbers and returns the result:

```cpp
int addNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

This function can be called as follows:

```cpp
int sum = addNumbers(5, 3);
```

**Function Return Type Deduction**

With C++14, you can ask the compiler to figure out the return type of a function automatically. To make use of this functionality, you need to specify `auto` as the return type:

```cpp
auto addNumbers(int number1, int number2)
{
    return number1 + number2;
}
```
The compiler deduces the return type based on the expressions used for the return statements. There can be multiple return statements in the function, but they should all resolve to the same type. Such a function can even include recursive calls (calls to itself), but the first return statement in the function must be a non-recursive call.

**Current Function’s Name**

Every function has a local predefined variable __func__ containing the name of the current function. One use of this variable would be for logging purposes:

```cpp
int addNumbers(int number1, int number2)
{
    std::cout << "Entering function " << __func__ << std::endl;
    return number1 + number2;
}
```

**C-Style Arrays**

Arrays hold a series of values, all of the same type, each of which can be accessed by its position in the array. In C++, you must provide the size of the array when the array is declared. You cannot give a variable as the size—it must be a constant, or a constant expression (constexpr). Constant expressions are discussed in Chapter 11. The code that follows shows the declaration of an array of three integers followed by three lines to initialize the elements to 0:

```cpp
int myArray[3];
myArray[0] = 0;
myArray[1] = 0;
myArray[2] = 0;
```

**WARNING** In C++, the first element of an array is always at position 0, not position 1! The last position of the array is always the size of the array minus 1!

The next section discusses loops that you can use to initialize each element. However, instead of using loops, or using the previous initialization mechanism, you can also accomplish the zero-initialization with the following one-liner:

```cpp
int myArray[3] = {0};
```

You can even drop the 0 as follows:

```cpp
int myArray[3] = {};
```

An array can also be initialized with an initializer list, in which case the compiler can deduce the size of the array automatically. For example,

```cpp
int myArray[] = {1, 2, 3, 4}; // The compiler creates an array of 4 elements.
```

If you do specify the size of the array, and the initializer list has less elements than the given size, the remaining elements are set to 0. For example, the following code only sets the first element in the array to the value 2, and sets all the other elements to 0:

```cpp
int myArray[3] = {2};
```
To get the size of a stack-based C-style array, you can use the C++17 `std::size()` function (requires `<array>`). For example:

```cpp
unsigned int arraySize = std::size(myArray);
```

If your compiler is not yet C++17 compliant, the old trick to get the size of a stack-based C-style array is to use the `sizeof` operator. The `sizeof` operator returns the size of its argument in bytes. To get the number of elements in a stack-based array, you divide the size in bytes of the array by the size in bytes of the first element. For example:

```cpp
unsigned int arraySize = sizeof(myArray) / sizeof(myArray[0]);
```

The preceding examples show a one-dimensional array, which you can think of as a line of integers, each with its own numbered compartment. C++ allows multi-dimensional arrays. You might think of a two-dimensional array as a checkerboard, where each location has a position along the x-axis and a position along the y-axis. Three-dimensional and higher arrays are harder to picture and are rarely used. The following code shows the syntax for allocating a two-dimensional array of characters for a Tic-Tac-Toe board and then putting an “o” in the center square:

```cpp
char ticTacToeBoard[3][3];
ticTacToeBoard[1][1] = 'o';
```

Figure 1-1 shows a visual representation of this board with the position of each square.

![Tic-Tac-Toe Board](image)

**FIGURE 1-1**

**NOTE** In C++, it’s best to avoid C-style arrays as discussed in this section, and instead use Standard Library functionality, such as `std::array`, and `std::vector`, as discussed in the next two sections.
std::array

The arrays discussed in the previous section come from C, and still work in C++. However, C++ has a special type of fixed-size container called std::array, defined in the <array> header file. It’s basically a thin wrapper around C-style arrays.

There are a number of advantages to using std::arrays instead of C-style arrays. They always know their own size, are not automatically cast to a pointer to avoid certain types of bugs, and have iterators to easily loop over the elements. Iterators are discussed in detail in Chapter 17.

The following example demonstrates how to use the array container. The use of angle brackets after array, as in array<int, 3>, will become clear during the discussion of templates in Chapter 12. However, for now, just remember that you have to specify two parameters between the angle brackets. The first parameter represents the type of the elements in the array, and the second one represents the size of the array.

```cpp
array<int, 3> arr = {9, 8, 7};
cout << "Array size = " << arr.size() << endl;
cout << "2nd element = " << arr[1] << endl;
```

NOTE Both the C-style arrays and the std::arrays have a fixed size, which must be known at compile time. They cannot grow or shrink at run time.

If you want an array with a dynamic size, it is recommended to use std::vector, as explained in the next section. A vector automatically increases in size when you add new elements to it.

std::vector

The C++ Standard Library provides a number of different non-fixed-size containers that can be used to store information. std::vector, declared in <vector>, is an example of such a container. The vector replaces the concept of C-style arrays with a much more flexible and safer mechanism. As a user, you need not worry about memory management, as the vector automatically allocates enough memory to hold its elements. A vector is dynamic, meaning that elements can be added and removed at run time. Chapter 17 goes into more detail regarding containers, but the basic use of a vector is straightforward, which is why it’s introduced in the beginning of this book so that it can be used in examples. The following code demonstrates the basic functionality of vector.

```cpp
// Create a vector of integers
vector<int> myVector = { 11, 22 };

// Add some more integers to the vector using push_back()
myVector.push_back(33);
myVector.push_back(44);

// Access elements
cout << "1st element: " << myVector[0] << endl;
```
myVector is declared as `vector<int>`. The angle brackets are required to specify the template parameters, just as with `std::array`. A `vector` is a generic container. It can contain almost any kind of object; that’s why you have to specify the type of object you want in your `vector` between the angle brackets. Templates are discussed in detail in Chapters 12 and 22.

To add elements to a `vector`, you can use the `push_back()` method. Individual elements can be accessed using a similar syntax as for arrays, i.e. `operator[]`.

**Structured Bindings**

C++17 introduces the concept of *structured bindings*. Structured bindings allow you to declare multiple variables that are initialized with elements from an array, struct, pair, or tuple.

For example, assume you have the following array:

```cpp
std::array<int, 3> values = { 11, 22, 33 };
```

You can declare three variables, `x`, `y`, and `z`, initialized with the three values from the array as follows. Note that you have to use the `auto` keyword for structured bindings. You cannot, for example, specify `int` instead of `auto`.

```cpp
auto [x, y, z] = values;
```

The number of variables declared with the structured binding has to match the number of values in the expression on the right.

Structured bindings also work with structures if all non-static members are public. For example,

```cpp
struct Point { double mX, mY, mZ; };
Point point;
point.mX = 1.0; point.mY = 2.0; point.mZ = 3.0;
auto [x, y, z] = point;
```

Examples with `std::pair` and `std::tuple` are given in chapters 17 and 20 respectively.

**Loops**

Computers are great for doing the same thing over and over. C++ provides four looping mechanisms: the `while` loop, `do/while` loop, `for` loop, and *range-based for* loop.

**The while Loop**

The `while` loop lets you perform a block of code repeatedly as long as an expression evaluates to true. For example, the following completely silly code will output “This is silly.” five times:

```cpp
int i = 0;
while (i < 5) {
    std::cout << "This is silly." << std::endl;
    ++i;
}
```
The keyword `break` can be used within a loop to immediately get out of the loop and continue execution of the program. The keyword `continue` can be used to return to the top of the loop and reevaluate the `while` expression. However, using `continue` in loops is often considered poor style because it causes the execution of a program to jump around somewhat haphazardly, so use it sparingly.

**The do/while Loop**

C++ also has a variation on the `while` loop called `do/while`. It works similarly to the `while` loop, except that the code to be executed comes first, and the conditional check for whether or not to continue happens at the end. In this way, you can use a loop when you want a block of code to always be executed at least once and possibly additional times based on some condition. The example that follows outputs the statement, “This is silly.” once, even though the condition ends up being false:

```cpp
int i = 100;
do {
    std::cout << "This is silly." << std::endl;
    ++i;
} while (i < 5);
```

**The for Loop**

The `for` loop provides another syntax for looping. Any `for` loop can be converted to a `while` loop and vice versa. However, the `for` loop syntax is often more convenient because it looks at a loop in terms of a starting expression, an ending condition, and a statement to execute at the end of every iteration. In the following code, `i` is initialized to 0; the loop continues as long as `i` is less than 5; and at the end of every iteration, `i` is incremented by 1. This code does the same thing as the `while` loop example, but is more readable because the starting value, ending condition, and per-iteration statement are all visible on one line.

```cpp
for (int i = 0; i < 5; ++i) {
    std::cout << "This is silly." << std::endl;
}
```

**The Range-Based for Loop**

The range-based `for` loop is the fourth looping mechanism. It allows for easy iteration over elements of a container. This type of loop works for C-style arrays, initializer lists (discussed later in this chapter), and any type that has `begin()` and `end()` methods returning iterators (see Chapter 17), such as `std::array`, `std::vector`, and all other Standard Library containers discussed in Chapter 17.

The following example first defines an array of four integers. The range-based `for` loop then iterates over a copy of every element in this array and prints each value. To iterate over the elements themselves without making copies, use a reference variable, as I discuss later in this chapter.

```cpp
std::array<int, 4> arr = {1, 2, 3, 4};
for (int i : arr) {
    std::cout << i << std::endl;
}
```
Initializer Lists

Initializer lists are defined in the `<initializer_list>` header file and make it easy to write functions that can accept a variable number of arguments. The `initializer_list` class is a template and so it requires you to specify the type of elements in the list between angle brackets, similar to how you have to specify the type of object stored in a `vector`. The following example shows how to use an initializer list:

```cpp
#include <initializer_list>

using namespace std;

int makeSum(initializer_list<int> lst)
{
    int total = 0;
    for (int value : lst) {
        total += value;
    }
    return total;
}
```

The function `makeSum()` accepts an initializer list of integers as argument. The body of the function uses a range-based `for` loop to accumulate the total sum. This function can be used as follows:

```cpp
int a = makeSum({1,2,3});
int b = makeSum({10,20,30,40,50,60});
```

Initializer lists are type safe and define which type is allowed to be in the list. For the `makeSum()` function shown here, all elements of the initializer list must be integers. Trying to call it with a `double` results in a compiler error or warning, as shown here:

```cpp
int c = makeSum({1,2,3.0});
```

Those Are the Basics

At this point, you have reviewed the basic essentials of C++ programming. If this section was a breeze, skim the next section to make sure that you are up to speed on the more-advanced material. If you struggled with this section, you may want to obtain one of the fine introductory C++ books mentioned in Appendix B before continuing.

DIVING DEEPER INTO C++

Loops, variables, and conditionals are terrific building blocks, but there is much more to learn. The topics covered next include many features designed to help C++ programmers with their code as well as a few features that are often more confusing than helpful. If you are a C programmer with little C++ experience, you should read this section carefully.
Strings in C++

There are three ways to work with strings of text in C++: the C-style, which represents strings as arrays of characters; the C++ style, which wraps that representation in an easier-to-use string type; and the general class of nonstandard approaches. Chapter 2 provides a detailed discussion.

For now, the only thing you need to know is that the C++ string type is defined in the `<string>` header file, and that you can use a C++ string almost like a basic type. Just like I/O streams, the string type lives in the std namespace. The following example shows that strings can be used just like character arrays:

```cpp
string myString = "Hello, World";
cout << "The value of myString is " << myString << endl;
cout << "The second letter is " << myString[1] << endl;
```

Pointers and Dynamic Memory

Dynamic memory allows you to build programs with data that is not of fixed size at compile time. Most nontrivial programs make use of dynamic memory in some form.

The Stack and the Heap

Memory in your C++ application is divided into two parts—the stack and the heap. One way to visualize the stack is as a deck of cards. The current top card represents the current scope of the program, usually the function that is currently being executed. All variables declared inside the current function will take up memory in the top stack frame, the top card of the deck. If the current function, which I’ll call `foo()`, calls another function `bar()`, a new card is put on the deck so that `bar()` has its own stack frame to work with. Any parameters passed from `foo()` to `bar()` are copied from the `foo()` stack frame into the `bar()` stack frame. Figure 1-2 shows what the stack might look like during the execution of a hypothetical function `foo()` that has declared two integer values.

Stack frames are nice because they provide an isolated memory workspace for each function. If a variable is declared inside the `foo()` stack frame, calling the `bar()` function won’t change it unless you specifically tell it to. Also, when the `foo()` function is done running, the stack frame goes away, and all of the variables declared within the function no longer take up memory. Variables that are stack-allocated do not need to be deallocated (deleted) by the programmer; it happens automatically.

The heap is an area of memory that is completely independent of the current function or stack frame. You can put variables on the heap if you want them to exist even when the function in which they were created has completed. The heap is less structured than the stack. You can think of it as just a pile of bits. Your program can add new bits to the pile at any time or modify bits that are already in the pile. You have to make sure that you deallocate (delete) any memory that you allocated on the heap. This does not happen automatically, unless you use smart pointers, which are discussed in the section “Smart Pointers.”
Working with Pointers

You can put anything on the heap by explicitly allocating memory for it. For example, to put an integer on the heap, you need to allocate memory for it, but first you need to declare a pointer:

```cpp
int* myIntegerPointer;
```

The * after the `int` type indicates that the variable you are declaring refers or points to some integer memory. Think of the pointer as an arrow that points at the dynamically allocated heap memory. It does not yet point to anything specific because you haven’t assigned it to anything; it is an uninitialized variable. Uninitialized variables should be avoided at all times, and especially uninitialized pointers because they point to some random place in memory. Working with such pointers will most likely make your program crash. That’s why you should always declare and initialize your pointers at the same time. You can initialize them to a null pointer (`nullptr—for more information, see the “Null Pointer Constant” section) if you don’t want to allocate memory right away:

```cpp
int* myIntegerPointer = nullptr;
```

A null pointer is a special default value that no valid pointer will ever have, and converts to `false` when used in a Boolean expression. For example:

```cpp
if (!myIntegerPointer) { /* myIntegerPointer is a null pointer */ }
```

You use the `new` operator to allocate the memory:

```cpp
myIntegerPointer = new int;
```

In this case, the pointer points to the address of just a single integer value. To access this value, you need to dereference the pointer. Think of dereferencing as following the pointer’s arrow to the actual value on the heap. To set the value of the newly allocated heap integer, you would use code like the following:

```cpp
*myIntegerPointer = 8;
```

Notice that this is not the same as setting `myIntegerPointer` to the value 8. You are not changing the pointer; you are changing the memory that it points to. If you were to reassign the pointer value, it would point to the memory address 8, which is probably random garbage that will eventually make your program crash.

After you are finished with your dynamically allocated memory, you need to deallocate the memory using the `delete` operator. To prevent the pointer from being used after having deallocated the memory it points to, it’s recommended to set your pointer to `nullptr`:

```cpp
delete myIntegerPointer;
myIntegerPointer = nullptr;
```

---

**WARNING** A pointer must be valid before it is dereferenced. Dereferencing a null pointer or an uninitialized pointer causes undefined behavior. Your program might crash, but it might just as well keep running and start giving strange results.
Pointers don’t always point to heap memory. You can declare a pointer that points to a variable on the stack, even another pointer. To get a pointer to a variable, you use the & (“address of”) operator:

```cpp
int i = 8;
int* myIntegerPointer = &i;  // Points to the variable with the value 8
```

C++ has a special syntax for dealing with pointers to structures. Technically, if you have a pointer to a structure, you can access its fields by first dereferencing it with *, then using the normal syntax, as in the code that follows, which assumes the existence of a function called `getEmployee()`.

```cpp
Employee* anEmployee = getEmployee();
cout << (*anEmployee).salary << endl;
```

This syntax is a little messy. The -> (arrow) operator lets you perform both the dereference and the field access in one step. The following code is equivalent to the preceding code, but is easier to read:

```cpp
Employee* anEmployee = getEmployee();
cout << anEmployee->salary << endl;
```

Remember the concept of short-circuiting logic, which was discussed earlier in this chapter? This can be useful in combination with pointers to avoid using an invalid pointer, as in the following example:

```cpp
bool isValidSalary = (anEmployee && anEmployee->salary > 0);
```

Or, a little bit more verbose:

```cpp
bool isValidSalary = (anEmployee != nullptr && anEmployee->salary > 0);
```

`anEmployee` is only dereferenced to get the salary if it is a valid pointer. If it is a null pointer, the logical operation short-circuits, and the `anEmployee` pointer is not dereferenced.

**Dynamically Allocated Arrays**

The heap can also be used to dynamically allocate arrays. You use the `new[]` operator to allocate memory for an array.

```cpp
int arraySize = 8;
int* myVariableSizedArray = new int[arraySize];
```

This allocates memory for enough integers to satisfy the `arraySize` variable. Figure 1-3 shows what the stack and the heap both look like after this code is executed. As you can see, the pointer variable still resides on the stack, but the array that was dynamically created lives on the heap.

Now that the memory has been allocated, you can work with `myVariableSizedArray` as though it were a regular stack-based array.

```cpp
myVariableSizedArray[3] = 2;
```
When your code is done with the array, it should remove the array from the heap so that other variables can use the memory. In C++, you use the `delete[]` operator to do this.

```cpp
delete[] myVariableSizedArray;
myVariableSizedArray = nullptr;
```

The brackets after `delete` indicate that you are deleting an array!

### NOTE
*Avoid using `malloc()` and `free()` from C. Instead, use `new` and `delete`, or `new[]` and `delete[]`.)*

### WARNING
*To prevent memory leaks, every call to `new` should be paired with a call to `delete`, and every call to `new[]` should be paired with a call to `delete[]`. Not calling `delete` or `delete[]`, or mismatching calls, results in memory leaks. Memory leaks are discussed in Chapter 7.*

#### Null Pointer Constant

Before C++11, the constant `NULL` was used for null pointers. `NULL` is simply defined as the constant 0, and this can cause problems. Take the following example:

```cpp
void func(char* str) {cout << "char* version" << endl;}
void func(int i) {cout << "int version" << endl;}

int main()
{
    func(NULL);
    return 0;
}
```
The `main()` function is calling `func()` with parameter `NULL`, which is supposed to be a null pointer constant. In other words, you are expecting the `char*` version of `func()` to be called with a null pointer as argument. However, since `NULL` is not a pointer, but identical to the integer 0, the integer version of `func()` is called.

This problem is solved with the introduction of a real null pointer constant, `nullptr`. The following code calls the `char*` version:

```
func(nullptr);
```

**Smart Pointers**

To avoid common memory problems, you should use *smart pointers* instead of “raw,” also called “naked,” C-style pointers. Smart pointers automatically deallocate memory when the smart pointer object goes out of scope, for example, when the function has finished executing.

The following are the two most important smart pointer types in C++, both defined in `<memory>` and in the `std` namespace:

- `std::unique_ptr`
- `std::shared_ptr`

`unique_ptr` is analogous to an ordinary pointer, except that it automatically frees the memory or resource when the `unique_ptr` goes out of scope or is deleted. As such, `unique_ptr` has sole ownership of the object pointed to. One advantage of a `unique_ptr` is that memory and resources are always freed, even when return statements are executed, or when exceptions (discussed later in this chapter) are thrown. This, for example, simplifies coding when a function has multiple return statements, because you don’t have to remember to free the resources before each return statement.

To create a `unique_ptr`, you should use `std::make_unique<>()`. For example, instead of writing the following,

```cpp
Employee* anEmployee = new Employee;
// ...  
delete anEmployee;
```

you should write this:

```cpp
auto anEmployee = make_unique<Employee>();
```

Note that you do not call `delete` anymore; it happens automatically for you. The `auto` keyword is discussed in more detail in the “Type Inference” section later in this chapter. For now, it suffices to know that the `auto` keyword tells the compiler to automatically deduce the type of a variable, so that you don’t have to manually specify the full type.

`unique_ptr` is a generic smart pointer that can point to any kind of memory. That's why it is a template. Templates require the angle brackets, `< >`, to specify the template parameters. Between the brackets, you have to specify the type of memory you want your `unique_ptr` to point to. Templates are discussed in detail in Chapters 12 and 22, but the smart pointers are introduced in Chapter 1 so that they can be used throughout the book—and as you will see, they are easy to use.
make_unique() has been available since C++14. If your compiler is not yet C++14 compliant, you can make your unique_ptr as follows (note that you now have to specify the type, Employee, twice):

    unique_ptr<Employee> anEmployee(new Employee);

You can use the anEmployee smart pointer in the same way as a normal pointer, for example:

    if (anEmployee) {
        cout << "Salary: " << anEmployee->salary << endl;
    }

A unique_ptr can also be used to store a C-style array. The following example creates an array of ten Employee instances, stores it in a unique_ptr, and shows how to access an element from the array:

    auto employees = make_unique<Employee[]>(10);
    cout << "Salary: " << employees[0].salary << endl;

shared_ptr allows for distributed ownership of the data. Each time a shared_ptr is assigned, a reference count is incremented indicating there is one more owner of the data. When a shared_ptr goes out of scope, the reference count is decremented. When the reference count goes to zero, it means there is no longer any owner of the data, and the object referenced by the pointer is freed.

To create a shared_ptr, you should use std::make_shared<>(), which is similar to make_unique<>():

    auto anEmployee = make_shared<Employee>();
    if (anEmployee) {
        cout << "Salary: " << anEmployee->salary << endl;
    }

Starting with C++17, you can also store an array in a shared_ptr, whereas older versions of C++ did not allow this. Note however that make_shared<>() of C++17 cannot be used in this case. Here is an example:

    shared_ptr<Employee[]> employees(new Employee[10]);
    cout << "Salary: " << employees[0].salary << endl;

Chapter 7 discusses memory management and smart pointers in more details, but because the basic use of unique_ptr and shared_ptr is straightforward, they are already used in examples throughout this book.

NOTE Raw pointers are only allowed if there is no ownership involved. Otherwise, use unique_ptr by default, and shared_ptr if you need shared ownership. If you know about auto_ptr, forget it; it was deprecated in C++11/14, and has been removed from C++17.
The Many Uses of const

The keyword `const` can be used in several different ways in C++. All of its uses are related, but there are subtle differences. The subtleties of `const` make for excellent interview questions! Chapter 11 explains in detail all the ways that `const` can be used. This section outlines two common use-cases.

**const Constants**

If you assumed that the keyword `const` has something to do with constants, you have correctly uncovered one of its uses. In the C language, programmers often use the preprocessor `#define` mechanism to declare symbolic names for values that won’t change during the execution of the program, such as the version number. In C++, programmers are encouraged to avoid `#define` in favor of using `const` to define constants. Defining a constant with `const` is just like defining a variable, except that the compiler guarantees that code cannot change the value.

```cpp
const int versionNumberMajor = 2;
const int versionNumberMinor = 1;
const std::string productName = "Super Hyper Net Modulator";
```

**const to Protect Parameters**

In C++, you can cast a non-`const` variable to a `const` variable. Why would you want to do this? It offers some degree of protection from other code changing the variable. If you are calling a function that a coworker of yours is writing, and you want to ensure that the function doesn’t change the value of a parameter you pass in, you can tell your coworker to have the function take a `const` parameter. If the function attempts to change the value of the parameter, it will not compile.

In the following code, a `string*` is automatically cast to a `const string*` in the call to `mysteryFunction()`. If the author of `mysteryFunction()` attempts to change the value of the passed string, the code will not compile. There are ways around this restriction, but using them requires conscious effort. C++ only protects against accidentally changing `const` variables.

```cpp
void mysteryFunction(const std::string* someString)
{
    *someString = "Test";  // Will not compile.
}

int main()
{
    std::string myString = "The string";
    mysteryFunction(&myString);
    return 0;
}
```

**References**

A reference in C++ allows you to give another name to an existing variable. For example:

```cpp
int x = 42;
int& xReference = x;
```
Attaching `&` to a type indicates that the variable is a reference. It is still used as though it was a normal variable, but behind the scenes, it is really a pointer to the original variable. Both the variable `x` and the reference variable `xReference` point to exactly the same value. If you change the value through either one of them, the change is visible through the other one as well.

Pass By Reference

Normally, when you pass a variable into a function, you are passing by value. If a function takes an integer parameter, it is really a copy of the integer that you pass in, so you cannot modify the value of the original variable. Pointers to stack variables are often used in C to allow functions to modify variables in other stack frames. By dereferencing the pointer, the function can change the memory that represents the variable even though that variable isn’t in the current stack frame. The problem with this approach is that it brings the messiness of pointer syntax into what is really a simple task.

Instead of passing pointers to functions, C++ offers a better mechanism, called pass by reference, where parameters are references instead of pointers. Following are two implementations of an `addOne()` function. The first one has no effect on the variable that is passed in because it is passed by value and thus the function receives a copy of the value passed to it. The second one uses a reference and thus changes the original variable.

```cpp
void addOne(int i)
{
    i++;  // Has no real effect because this is a copy of the original
}

void addOne(int& i)
{
    i++;  // Actually changes the original variable
}
```

The syntax for the call to the `addOne()` function with an integer reference is no different than if the function just took an integer:

```cpp
int myInt = 7;
addOne(myInt);
```

**NOTE** There is a subtle difference between the two `addOne()` implementations. The version using pass-by-value accepts literals without a problem; for example, “`addOne(3);`” is legal. However, doing the same with the pass-by-reference version of `addOne()` will result in a compiler error. This can be solved by using `const` references, discussed in the next section, or rvalue references, an advanced C++ feature explained in Chapter 9.

If you have a function that needs to return a big structure or class (discussed later in this chapter) that is expensive to copy, you’ll often see the function taking a non-`const` reference to such a structure or class which the function then modifies, instead of directly returning it. This was the recommended way a long time ago to prevent the performance penalty of creating a copy when you return the structure or class from the function. Since C++11, this is not necessary anymore. Thanks to
move semantics, directly returning structures or classes from functions is efficient without any copying. Move semantics is discussed in detail in Chapter 9.

**Pass By const Reference**

You will often find code that uses `const` reference parameters for functions. At first, that seems like a contradiction. Reference parameters allow you to change the value of a variable from within another context. `const` seems to prevent such changes.

The main value in `const` reference parameters is efficiency. When you pass a value into a function, an entire copy is made. When you pass a reference, you are really just passing a pointer to the original so the computer doesn’t need to make a copy. By passing a `const` reference, you get the best of both worlds: no copy is made but the original variable cannot be changed.

`const` references become more important when you are dealing with objects because they can be large and making copies of them can have unwanted side effects. Subtle issues like this are covered in Chapter 11. The following example shows how to pass an `std::string` to a function as a `const` reference:

```cpp
void printString(const std::string& myString)
{
    std::cout << myString << std::endl;
}

int main()
{
    std::string someString = "Hello World";
    printString(someString);
    printString("Hello World"); // Passing literals works
    return 0;
}
```

**NOTE** If you need to pass an object to a function, prefer to pass it by `const` reference instead of by value. This prevents unnecessary copying. Pass it by non-`const` reference if the function needs to modify the object.

**Exceptions**

C++ is a very flexible language, but not a particularly safe one. The compiler will let you write code that scribbles on random memory addresses or tries to divide by zero (computers don’t deal well with infinity). One language feature that attempts to add a degree of safety back to the language is exceptions.

An exception is an unexpected situation. For example, if you are writing a function that retrieves a web page, several things could go wrong. The Internet host that contains the page might be down, the page might come back blank, or the connection could be lost. One way you could handle this situation is by returning a special value from the function, such as `nullptr` or an error code. Exceptions provide a much better mechanism for dealing with problems.
Exceptions come with some new terminology. When a piece of code detects an exceptional situation, it *throws* an exception. Another piece of code *catches* the exception and takes appropriate action. The following example shows a function, `divideNumbers()`, that throws an exception if the caller passes in a denominator of zero. The use of `std::invalid_argument` requires `<stdexcept>`.

```cpp
double divideNumbers(double numerator, double denominator)
{
    if (denominator == 0) {
        throw invalid_argument("Denominator cannot be 0.");
    }
    return numerator / denominator;
}
```

When the `throw` line is executed, the function immediately ends without returning a value. If the caller surrounds the function call with a `try/catch` block, as shown in the following code, it receives the exception and is able to handle it:

```cpp
try {
    cout << divideNumbers(2.5, 0.5) << endl;
    cout << divideNumbers(2.3, 0) << endl;
    cout << divideNumbers(4.5, 2.5) << endl;
} catch (const invalid_argument& exception) {
    cout << "Exception caught: " << exception.what() << endl;
}
```

The first call to `divideNumbers()` executes successfully, and the result is output to the user. The second call throws an exception. No value is returned, and the only output is the error message that is printed when the exception is caught. The third call is never executed because the second call throws an exception, causing the program to jump to the `catch` block. The output for the preceding block of code is as follows:

```
5
An exception was caught: Denominator cannot be 0.
```

Exceptions can get tricky in C++. To use exceptions properly, you need to understand what happens to the stack variables when an exception is thrown, and you have to be careful to properly catch and handle the necessary exceptions. Also, the preceding example uses the built-in `std::invalid_argument` type, but it is preferable to write your own exception types that are more specific to the error being thrown. Lastly, the C++ compiler doesn’t force you to catch every exception that might occur. If your code never catches any exceptions but an exception is thrown, it will be caught by the program itself, which will be terminated. These trickier aspects of exceptions are covered in much more detail in Chapter 14.

**Type Inference**

*Type inference* allows the compiler to automatically deduce the type of an expression. There are two keywords for type inference: `auto` and `decltype`. 


The auto Keyword

The `auto` keyword has a number of completely different uses:

➤ Deducing a function’s return type, as explained earlier in this chapter.
➤ Structured bindings, as explained earlier in this chapter.
➤ Deducing the type of an expression, as discussed later in this section.
➤ Deducing the type of non-type template parameters, see Chapter 12.
➤ `decltype(auto)`, see Chapter 12.
➤ Alternative function syntax, see Chapter 12.
➤ Generic lambda expressions, see Chapter 18.

`auto` can be used to tell the compiler to automatically deduce the type of a variable at compile time. The following line shows the simplest use of the `auto` keyword in that context:

```cpp
auto x = 123;    // x will be of type int
```

In this example, you don’t win much by typing `auto` instead of `int`; however, it becomes useful for more complicated types. Suppose you have a function called `getFoo()` that has a complicated return type. If you want to assign the result of calling `getFoo()` to a variable, you can spell out the complicated type, or you can simply use `auto` and let the compiler figure it out:

```cpp
auto result = getFoo();
```

This has the added benefit that you can easily change the function’s return type without having to update all the places in the code where that function is called.

However, using `auto` to deduce the type of an expression strips away reference and `const` qualifiers. Suppose you have the following function:

```cpp
#include <string>

const std::string message = "Test";

const std::string& foo()
{
    return message;
}
```

You can call `foo()` and store the result in a variable with the type specified as `auto`, as follows:

```cpp
auto f1 = foo();
```

Because `auto` strips away reference and `const` qualifiers, `f1` is of type `string`, and thus a `copy` is made. If you want a `const` reference, you can explicitly make it a reference and mark it `const`, as follows:

```cpp
const auto& f2 = foo();
```
The `decltype` Keyword

The `decltype` keyword takes an expression as argument, and computes the type of that expression, as shown here:

```cpp
int x = 123;
dcltype(x) y = 456;
```

In this example, the compiler deduces the type of `y` to be `int` because that is the type of `x`.

The difference between `auto` and `decltype` is that `decltype` does not strip reference and `const` qualifiers. Take again the function `foo()` returning a `const` reference to a string. Defining `f2` using `decltype` as follows results in `f2` being of type `const string&`, and thus no copy is made.

```cpp
dcltype(foo()) f2 = foo();
```

On first sight, `decltype` doesn’t seem to add much value. However, it is pretty powerful in the context of templates, discussed in Chapters 12 and 22.

---

**C++ AS AN OBJECT-ORIENTED LANGUAGE**

If you are a C programmer, you may have viewed the features covered so far in this chapter as convenient additions to the C language. As the name C++ implies, in many ways the language is just a “better C.” There is one major point that this view overlooks: unlike C, C++ is an object-oriented language.

Object-oriented programming (OOP) is a very different, arguably more natural, way to write code. If you are used to procedural languages such as C or Pascal, don’t worry. Chapter 5 covers all the background information you need to know to shift your mindset to the object-oriented paradigm. If you already know the theory of OOP, the rest of this section will get you up to speed (or refresh your memory) on basic C++ object syntax.

**Defining Classes**

A class defines the characteristics of an object. In C++, classes are usually defined in a header file (.h), while their definitions usually are in a corresponding source file (.cpp).

A basic class definition for an airline ticket class is shown in the following example. The class can calculate the price of the ticket based on the number of miles in the flight and whether or not the customer is a member of the “Elite Super Rewards Program.” The definition begins by declaring the class name. Inside a set of curly braces, the data members (properties) of the class and its methods (behaviors) are declared. Each data member and method is associated with a particular access level: public, protected, or private. These labels can occur in any order and can be repeated. Members that are public can be accessed from outside the class, while members that are private cannot be
accessed from outside the class. It’s recommended to make all your data members private, and if needed, to give access to them with public getters and setters. This way, you can easily change the representation of your data while keeping the public interface the same. The use of protected is explained in the context of inheritance in Chapters 5 and 10.

```cpp
#include <string>

class AirlineTicket
{
public:
    AirlineTicket();
    ~AirlineTicket();

double calculatePriceInDollars() const;

    const std::string& getPassengerName() const;
    void setPassengerName(const std::string& name);

    int getNumberOfMiles() const;
    void setNumberOfMiles(int miles);

    bool hasEliteSuperRewardsStatus() const;
    void setHasEliteSuperRewardsStatus(bool status);

private:
    std::string mPassengerName;
    int mNumberOfMiles;
    bool mHasEliteSuperRewardsStatus;
};
```

This book follows the convention to prefix each data member of a class with a lowercase ‘m’, such as `mPassengerName`.

**NOTE** To follow the const-correctness principle, it’s always a good idea to declare member functions that do not change any data member of the object as being const. These member functions are also called “inspectors,” compared to “mutators” for non-const member functions.

The method that has the same name as the class with no return type is a constructor. It is automatically called when an object of the class is created. The method with a tilde (~) character followed by the class name is a destructor. It is automatically called when the object is destroyed.

There are two ways of initializing data members with a constructor. The recommended way is to use a constructor initializer, which follows a colon after the constructor name. Here is the `AirlineTicket` constructor with a constructor initializer:

```cpp
AirlineTicket::AirlineTicket()
    : mPassengerName("Unknown Passenger")
    , mNumberOfMiles(0)
    , mHasEliteSuperRewardsStatus(false)
{
}
```
A second way is to put the initializations in the body of the constructor, as shown here:

```cpp
AirlineTicket::AirlineTicket()
{
    // Initialize data members
    mPassengerName = "Unknown Passenger";
    mNumberOfMiles = 0;
    mHasEliteSuperRewardsStatus = false;
}
```

If the constructor is only initializing data members without doing anything else, then there is no real need for a constructor because data members can be initialized directly inside the class definition. For example, instead of writing an `AirlineTicket` constructor, you can modify the definition of the data members in the class definition as follows:

```cpp
private:
    std::string mPassengerName = "Unknown Passenger";
    int mNumberOfMiles = 0;
    bool mHasEliteSuperRewardsStatus = false;
```

If your class additionally needs to perform some other types of initialization, such as opening a file, allocating memory, and so on, then you still need to write a constructor to handle those.

Here is the destructor for the `AirlineTicket` class:

```cpp
AirlineTicket::~AirlineTicket()
{
    // Nothing much to do in terms of cleanup
}
```

This destructor doesn’t do anything, and can simply be removed from this class. It is just shown here so you know the syntax of destructors. Destructors are required if you need to perform some cleanup, such as closing files, freeing memory, and so on. Chapters 8 and 9 discuss destructors in more detail.

The definitions of some of the `AirlineTicket` class methods are shown here:

```cpp
double AirlineTicket::calculatePriceInDollars() const
{
    if (hasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }
    // The cost of the ticket is the number of miles times 0.1.
    // Real airlines probably have a more complicated formula!
    return getNumberOfMiles() * 0.1;
}

const string& AirlineTicket::getPassengerName() const
{
    return mPassengerName;
}

void AirlineTicket::setPassengerName(const string& name)
{
    mPassengerName = name;
}
Using Classes

The following sample program makes use of the `AirlineTicket` class. This example shows the creation of a stack-based `AirlineTicket` object as well as a heap-based one:

```cpp
AirlineTicket myTicket; // Stack-based AirlineTicket
myTicket.setPassengerName("Sherman T. Socketwrench");
myTicket.setNumberOfMiles(700);
double cost = myTicket.calculatePriceInDollars();
cout << "This ticket will cost $" << cost << endl;

// Heap-based AirlineTicket with smart pointer
auto myTicket2 = make_unique<AirlineTicket>();
myTicket2->setPassengerName("Laudimore M. Hallidue");
myTicket2->setNumberOfMiles(2000);
myTicket2->setHasEliteSuperRewardsStatus(true);
double cost2 = myTicket2->calculatePriceInDollars();
cout << "This other ticket will cost $" << cost2 << endl;
// No need to delete myTicket2, happens automatically

// Heap-based AirlineTicket without smart pointer (not recommended)
AirlineTicket* myTicket3 = new AirlineTicket();
// ... Use ticket 3
delete myTicket3; // delete the heap object!
```

The preceding example exposes you to the general syntax for creating and using classes. Of course, there is much more to learn. Chapters 8, 9, and 10 go into more depth about the specific C++ mechanisms for defining classes.

UNIFORM INITIALIZATION

Before C++11, initialization of types was not always uniform. For example, take the following definition of a circle, once as a structure, and once as a class:

```cpp
struct CircleStruct
{
    int x, y;
    double radius;
};

class CircleClass
{
    public:
        CircleClass(int x, int y, double radius) :
            mX(x), mY(y), mRadius(radius) {}
    private:
        int mX, mY;
        double mRadius;
};
```
In pre-C++11, initialization of a variable of type `CircleStruct` and a variable of type `CircleClass` looks different:

```cpp
CircleStruct myCircle1 = {10, 10, 2.5};
CircleClass myCircle2(10, 10, 2.5);
```

For the structure version, you can use the `{...}` syntax. However, for the class version, you need to call the constructor using function notation (...).

Since C++11, you can more uniformly use the `{...}` syntax to initialize types, as follows:

```cpp
CircleStruct myCircle3 = {10, 10, 2.5};
CircleClass myCircle4 = {10, 10, 2.5};
```

The definition of `myCircle4` automatically calls the constructor of `CircleClass`. Even the use of the equal sign is optional, so the following is identical:

```cpp
CircleStruct myCircle5{10, 10, 2.5};
CircleClass myCircle6{10, 10, 2.5};
```

Uniform initialization is not limited to structures and classes. You can use it to initialize anything in C++. For example, the following code initializes all four variables with the value 3:

```cpp
int a = 3;
int b(3);
int c = {3};  // Uniform initialization
int d{3};     // Uniform initialization
```

Uniform initialization can be used to perform zero-initialization* of variables; you just specify an empty set of curly braces, as shown here:

```cpp
int e{};      // Uniform initialization, e will be 0
```

Using uniform initialization prevents narrowing. C++ implicitly performs narrowing, as shown here:

```cpp
void func(int i) { /* ... */ }

int main()
{
    int x = 3.14;
    func(3.14);
}
```

In both cases, C++ automatically truncates 3.14 to 3 before assigning it to `x` or calling `func()`. Note that some compilers *might* issue a warning about this narrowing, while others won't. With uniform initialization, both the assignment to `x` and the call to `func()` *must* generate a compiler error if your compiler fully conforms to the C++11 standard:

```cpp
void func(int i) { /* ... */ }

int main()
{
    int x = {3.14};    // Error because narrowing
    func({3.14});      // Error because narrowing
}
```

*Zero-initialization* constructs objects with the default constructor, and initializes primitive integer types (such as `char`, `int`, and so on) to zero, primitive floating-point types to 0.0, and pointer types to `nullptr`. 
Uniform initialization can be used to initialize dynamically allocated arrays, as shown here:

```cpp
int* pArray = new int[4]{0, 1, 2, 3};
```

It can also be used in the constructor initializer to initialize arrays that are members of a class.

```cpp
class MyClass
{
   public:
      MyClass() : mArray{0, 1, 2, 3} {}
   private:
      int mArray[4];
};
```

Uniform initialization can be used with the Standard Library containers as well—such as the `std::vector`, as demonstrated later in this chapter.

**Direct List Initialization versus Copy List Initialization**

There are two types of initialization that use braced initializer lists:

- **Copy list initialization.** `T obj = {arg1, arg2, ...};`
- **Direct list initialization.** `T obj {arg1, arg2, ...};`

In combination with auto type deduction, there is an important difference between copy- and direct list initialization introduced with C++17.

Starting with C++17, you have the following results:

```cpp
// Copy list initialization
auto a = {11}; // initializer_list<int>
auto b = {11, 22}; // initializer_list<int>

// Direct list initialization
auto c {11}; // int
auto d {11, 22}; // Error, too many elements.
```

Note that for copy list initialization, all the elements in the braced initializer must be of the same type. For example, the following does not compile:

```cpp
auto b = {11, 22.33}; // Compilation error
```

In earlier versions of the standard (C++11/14), both copy- and direct list initialization deduce an `initializer_list<>`:

```cpp
// Copy list initialization
auto a = {11}; // initializer_list<int>
auto b = {11, 22}; // initializer_list<int>

// Direct list initialization
auto c {11}; // initializer_list<int>
auto d {11, 22}; // initializer_list<int>
```
THE STANDARD LIBRARY

C++ comes with a Standard Library, which contains a lot of useful classes that can easily be used in your code. The benefit of using these classes is that you don’t need to reinvent certain classes and you don’t need to waste time on implementing things that have already been implemented for you. Another benefit is that the classes available in the Standard Library are heavily tested and verified for correctness by thousands of users. The Standard Library classes are also tuned for high performance, so using them will most likely result in better performance compared to making your own implementation.

A lot of functionality is available to you in the Standard Library. Chapters 16 to 20 provide more details; however, when you start working with C++ it is a good idea to understand what the Standard Library can do for you from the very beginning. This is especially important if you are a C programmer. As a C programmer, you might try to solve problems in C++ the same way you would solve them in C. However, in C++ there is probably an easier and safer solution to the problem that involves using Standard Library classes.

You already saw some Standard Library classes earlier in this chapter—for example, `std::string`, `std::array`, `std::vector`, `std::unique_ptr`, and `std::shared_ptr`. Many more classes are introduced in Chapters 16 to 20.

YOUR FIRST USEFUL C++ PROGRAM

The following program builds on the employee database example used earlier in the discussion on structs. This time, you will end up with a fully functional C++ program that uses many of the features discussed in this chapter. This real-world example includes the use of classes, exceptions, vectors, namespaces, references, and other language features.

An Employee Records System

A program to manage a company’s employee records needs to be flexible and have useful features. The feature set for this program includes the following abilities:

- To add an employee
- To fire an employee
- To promote an employee
- To view all employees, past and present
- To view all current employees
- To view all former employees

The design for this program divides the code into three parts. The `Employee` class encapsulates the information describing a single employee. The `Database` class manages all the employees of the company. A separate `UserInterface` file provides the interactivity of the program.
The Employee Class

The Employee class maintains all the information about an employee. Its methods provide a way to query and change that information. Employees also know how to display themselves on the console. Methods also exist to adjust the employee’s salary and employment status.

Employee.h

The Employee.h file defines the Employee class. The sections of this file are described individually in the text that follows.

The first line contains a #pragma once to prevent the file from being included multiple times, followed by the inclusion of the string functionality.

This code also declares that the subsequent code, contained within the curly braces, lives in the Records namespace. Records is the namespace that is used throughout this program for application-specific code.

```cpp
#pragma once
#include <string>
namespace Records {

const int kDefaultStartingSalary = 30000;
```

Note that this book uses the convention to prefix constants with a lowercase ‘k’, from the German “Konstant,” meaning “Constant.”

The Employee class is defined, along with its public methods. The promote() and demote() methods both have integer parameters that are specified with a default value. In this way, other code can omit the integer parameters and the default will automatically be used.

A number of setters and getters provide mechanisms to change the information about an employee or to query the current information about an employee.

The Employee class includes an explicitly defaulted constructor, as discussed in Chapter 8. It also includes a constructor that accepts a first and last name.

```cpp
class Employee {
    public:
        Employee() = default;
        Employee(const std::string& firstName,
                 const std::string& lastName);
        void promote(int raiseAmount = 1000);
        void demote(int demeritAmount = 1000);
        void hire(); // Hires or rehires the employee
        void fire(); // Dismisses the employee
        void display() const; // Outputs employee info to console
```
// Getters and setters
void setFirstName(const std::string& firstName);
const std::string& getFirstName() const;

void setLastName(const std::string& lastName);
const std::string& getLastName() const;

void setEmployeeNumber(int employeeNumber);
int getEmployeeNumber() const;

void setSalary(int newSalary);
int getSalary() const;

bool isHired() const;

Finally, the data members are declared as private so that other parts of the code cannot modify them directly. The setters and getters provide the only public way of modifying or querying those values. The data members are also initialized here instead of in a constructor. By default, new employees have no name, an employee number of -1, the default starting salary, and a status of not hired.

private:
    std::string mFirstName;
    std::string mLastName;
    int mEmployeeNumber = -1;
    int mSalary = kDefaultStartingSalary;
    bool mHired = false;

}

Employee.cpp

The constructor accepting a first and last name just sets the corresponding data members:

#include <iostream>
#include "Employee.h"

using namespace std;

namespace Records {
    Employee::Employee(const std::string& firstName,
                        const std::string& lastName)
        : mFirstName(firstName), mLastName(lastName)
    {
    }
}

The promote() and demote() methods simply call the setSalary() method with a new value. Note that the default values for the integer parameters do not appear in the source file; they are only allowed in a function declaration, not in a definition.

void Employee::promote(int raiseAmount)
{
    setSalary(getSalary() + raiseAmount);
}
void Employee::demote(int demeritAmount)
{
    setSalary(getSalary() - demeritAmount);
}

The hire() and fire() methods just set the mHired data member appropriately.

void Employee::hire()
{
    mHired = true;
}

void Employee::fire()
{
    mHired = false;
}

The display() method uses the console output stream to display information about the current employee. Because this code is part of the Employee class, it could access data members, such as mSalary, directly instead of using getters, such as getSalary(). However, it is considered good style to make use of getters and setters when they exist, even within the class.

void Employee::display() const
{
    cout << "Employee: " << getLastName() << ", " << getFirstName() << endl;
    cout << "-------------------------" << endl;
    cout << (isHired() ? "Current Employee" : "Former Employee") << endl;
    cout << "Employee Number: " << getEmployeeNumber() << endl;
    cout << "Salary: $" << getSalary() << endl;
    cout << endl;
}

A number of getters and setters perform the task of getting and setting values. Even though these methods seem trivial, it's better to have trivial getters and setters than to make your data members public. For example, in the future, you may want to perform bounds checking in the setSalary() method. Getters and setters also make debugging easier because you can insert a breakpoint in them to inspect values when they are retrieved or set. Another reason is that when you decide to change how you are storing the data in your class, you only need to modify these getters and setters.

// Getters and setters
void Employee::setFirstName(const string& firstName)
{
    mFirstName = firstName;
}

const string& Employee::getFirstName() const
{
    return mFirstName;
}

// ... other getters and setters omitted for brevity
EmployeeTest.cpp

As you write individual classes, it is often useful to test them in isolation. The following code includes a main() function that performs some simple operations using the Employee class. Once you are confident that the Employee class works, you should remove or comment-out this file so that you don’t attempt to compile your code with multiple main() functions.

```cpp
#include <iostream>
#include "Employee.h"

using namespace std;
using namespace Records;

int main()
{
    cout << "Testing the Employee class." << endl;
    Employee emp;
    emp.setFirstName("John");
    emp.setLastName("Doe");
    emp.setEmployeeNumber(71);
    emp.setSalary(50000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();
    return 0;
}
```

Another way of testing individual classes is with unit testing, which is discussed in Chapter 26.

The Database Class

The Database class uses the std::vector class from the Standard Library to store Employee objects.

Database.h

Because the database will take care of automatically assigning an employee number to a new employee, a constant defines where the numbering begins.

```cpp
#pragma once
#include <iostream>
#include <vector>
#include "Employee.h"

namespace Records {
    const int kFirstEmployeeNumber = 1000;
}
```

The database provides an easy way to add a new employee by providing a first and last name. For convenience, this method returns a reference to the new employee. External code can also get an employee reference by calling the getEmployee() method. Two versions of this method are declared. One allows retrieval by employee number. The other requires a first and last name.

```cpp
class Database
{
    public:
```
Employee& addEmployee(const std::string& firstName,  
                   const std::string& lastName);
Employee& getEmployee(int employeeNumber);
Employee& getEmployee(const std::string& firstName,  
                   const std::string& lastName);

Because the database is the central repository for all employee records, it has methods that output all employees, the employees who are currently hired, and the employees who are no longer hired.

void displayAll() const;
void displayCurrent() const;
void displayFormer() const;

mEmployees contains the Employee objects. The mNextEmployeeNumber data member keeps track of what employee number is assigned to a new employee, and is initialized with the kFirstEmployeeNumber constant.

private:
    std::vector<Employee> mEmployees;
    int mNextEmployeeNumber = kFirstEmployeeNumber;
};

Database.cpp

The addEmployee() method creates a new Employee object, fills in its information, and adds it to the vector. The mNextEmployeeNumber data member is incremented after its use so that the next employee will get a new number.

#include <iostream>
#include <stdexcept>
#include "Database.h"

using namespace std;

namespace Records {
    Employee& Database::addEmployee(const string& firstName, 
                   const string& lastName)
    {
        Employee theEmployee(firstName, lastName);
        theEmployee.setEmployeeNumber(mNextEmployeeNumber++);
        theEmployee.hire();
        mEmployees.push_back(theEmployee);
        return mEmployees[mEmployees.size() - 1];
    }
}

Only one version of getEmployee() is shown. Both versions work in similar ways. The methods loop over all employees in mEmployees using range-based for loops, and check to see if an Employee is a match for the information passed to the method. An exception is thrown if no match is found.

Employee& Database::getEmployee(int employeeNumber)
{
    for (auto& employee : mEmployees) {
        if (employee.getEmployeeNumber() == employeeNumber) {

The display methods all use a similar algorithm. They loop through all employees and tell each employee to display itself to the console if the criterion for display matches. `displayFormer()` is similar to `displayCurrent()`.

```cpp
void Database::displayAll() const
{
    for (const auto& employee : mEmployees) {
        employee.display();
    }
}

void Database::displayCurrent() const
{
    for (const auto& employee : mEmployees) {
        if (employee.isHired())
            employee.display();
    }
}
```

### DatabaseTest.cpp

A simple test for the basic functionality of the database is shown here:

```cpp
#include <iostream>
#include "Database.h"

using namespace std;
using namespace Records;

int main()
{
    Database myDB;
    Employee& emp1 = myDB.addEmployee("Greg", "Wallis");
    emp1.fire();

    Employee& emp2 = myDB.addEmployee("Marc", "White");
    emp2.setSalary(100000);

    Employee& emp3 = myDB.addEmployee("John", "Doe");
    emp3.setSalary(10000);
    emp3.promote();

    cout << "all employees: " << endl << endl;
    myDB.displayAll();

    cout << endl << "current employees: " << endl << endl;
    myDB.displayCurrent();
```
cout << endl << "former employees: " << endl << endl;
myDB.displayFormer();
}

The User Interface

The final part of the program is a menu-based user interface that makes it easy for users to work with the employee database.

The `main()` function is a loop that displays the menu, performs the selected action, then does it all again. For most actions, separate functions are defined. For simpler actions, like displaying employees, the actual code is put in the appropriate case.

```cpp
#include <iostream>
#include <stdexcept>
#include <exception>
#include "Database.h"

using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& db);
void doFire(Database& db);
void doPromote(Database& db);
void doDemote(Database& db);

int main()
{
    Database employeeDB;
    bool done = false;
    while (!done) {
        int selection = displayMenu();
        switch (selection) {
        case 0:
            done = true;
            break;
        case 1:
            doHire(employeeDB);
            break;
        case 2:
            doFire(employeeDB);
            break;
        case 3:
            doPromote(employeeDB);
            break;
        case 4:
            employeeDB.displayAll();
            break;
        case 5:
            employeeDB.displayCurrent();
            break;
        case 6:
            employeeDB.displayFormer();
            break;
        }
    }
}
```
break;
default:
    cerr << "Unknown command." << endl;
    break;
}
return 0;
}

The `displayMenu()` function outputs the menu and gets input from the user. One important note is that this code assumes that the user will “play nice” and type a number when a number is requested. When you read about I/O in Chapter 13, you will learn how to protect against bad input.

```cpp
int displayMenu()
{
    int selection;
    cout << endl;
    cout << "Employee Database" << endl;
    cout << "-----------------" << endl;
    cout << "1) Hire a new employee" << endl;
    cout << "2) Fire an employee" << endl;
    cout << "3) Promote an employee" << endl;
    cout << "4) List all employees" << endl;
    cout << "5) List all current employees" << endl;
    cout << "6) List all former employees" << endl;
    cout << "0) Quit" << endl;
    cout << endl;
    cout << "--> ";
    cin >> selection;
    return selection;
}
```

The `doHire()` function gets the new employee’s name from the user and tells the database to add the employee.

```cpp
void doHire(Database& db)
{
    string firstName;
    string lastName;

    cout << "First name? ";
    cin >> firstName;

    cout << "Last name? ";
    cin >> lastName;

    db.addEmployee(firstName, lastName);
}
doFire() and doPromote() both ask the database for an employee by their employee number and then use the public methods of the Employee object to make changes.

```cpp
void doFire(Database& db)
{
    int employeeNumber;

    cout << "Employee number? ";
    cin >> employeeNumber;

    try {
        Employee& emp = db.getEmployee(employeeNumber);
        emp.fire();
        cout << "Employee " << employeeNumber << " terminated." << endl;
    } catch (const std::logic_error& exception) {
        cerr << "Unable to terminate employee: " << exception.what() << endl;
    }
}

void doPromote(Database& db)
{
    int employeeNumber;
    int raiseAmount;

    cout << "Employee number? ";
    cin >> employeeNumber;

    cout << "How much of a raise? ";
    cin >> raiseAmount;

    try {
        Employee& emp = db.getEmployee(employeeNumber);
        emp.promote(raiseAmount);
    } catch (const std::logic_error& exception) {
        cerr << "Unable to promote employee: " << exception.what() << endl;
    }
}
```

Evaluating the Program

The preceding program covers a number of topics from the very simple to the relatively complex. There are a number of ways that you could extend this program. For example, the user interface does not expose all of the functionality of the Database or Employee classes. You could modify the UI to include those features. You could also change the Database class to remove fired employees from mEmployees.

If there are parts of this program that don’t make sense, consult the preceding sections to review those topics. If something is still unclear, the best way to learn is to play with the code and try things out. For example, if you’re not sure how to use the conditional operator, write a short main() function that uses it.
SUMMARY

Now that you know the fundamentals of C++, you are ready to become a professional C++ programmer. When you start getting deeper into the C++ language later in this book, you can refer to this chapter to brush up on parts of the language you may need to review. Going back to some of the sample code in this chapter may be all you need to bring a forgotten concept back to the forefront of your mind.

The next chapter goes deeper in on how strings are handled in C++, because every program you write will have to work with strings one way or another.
Working with Strings and String Views

WHAT’S IN THIS CHAPTER?

➤ The differences between C-style strings and C++ strings
➤ Details of the C++ std::string class
➤ Why you should use std::string_view
➤ What raw string literals are

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of the chapter’s code download on this book’s website at www.wrox.com/go/proc++4e on the Download Code tab.

Every program that you write will use strings of some kind. With the old C language, there is not much choice but to use a dumb null-terminated character array to represent a string. Unfortunately, doing so can cause a lot of problems, such as buffer overflows, which can result in security vulnerabilities. The C++ Standard Library includes a safe and easy-to-use std::string class that does not have these disadvantages.

Because strings are so important, this chapter discusses them in more detail.
DYNAMIC STRINGS

Strings in languages that have supported them as first-class objects tend to have a number of attractive features, such as being able to expand to any size, or to have sub-strings extracted or replaced. In other languages, such as C, strings were almost an afterthought; there was no really good “string” data type, just fixed arrays of bytes. The “string library” was nothing more than a collection of rather primitive functions without even bounds checking. C++ provides a string type as a first-class data type.

C-Style Strings

In the C language, strings are represented as an array of characters. The last character of a string is a null character (‘\0’) so that code operating on the string can determine where it ends. This null character is officially known as NULL, spelled with one L, not two. NULL is not the same as the NULL pointer. Even though C++ provides a better string abstraction, it is important to understand the C technique for strings because they still arise in C++ programming. One of the most common situations is where a C++ program has to call a C-based interface in some third-party library or as part of interfacing to the operating system.

By far, the most common mistake that programmers make with C strings is that they forget to allocate space for the ‘\0’ character. For example, the string “hello” appears to be five characters long, but six characters worth of space are needed in memory to store the value, as shown in Figure 2-1.

C++ contains several functions from the C language that operate on strings. These functions are defined in the <cstring> header. As a general rule of thumb, these functions do not handle memory allocation. For example, the strcpy() function takes two strings as parameters. It copies the second string onto the first, whether it fits or not. The following code attempts to build a wrapper around strcpy() that allocates the correct amount of memory and returns the result, instead of taking in an already allocated string. It uses the strlen() function to obtain the length of the string. The caller is responsible for freeing the memory allocated by copyString().

```c
char* copyString(const char* str)
{
    char* result = new char[strlen(str)];  // BUG! Off by one!
    strcpy(result, str);
    return result;
}
```

The copyString() function as written is incorrect. The strlen() function returns the length of the string, not the amount of memory needed to hold it. For the string “hello”, strlen() returns 5, not 6. The proper way to allocate memory for a string is to add 1 to the amount of space needed for the actual characters. It seems a bit unnatural to have +1 all over the place. Unfortunately, that’s how it works, so keep this in mind when you work with C-style strings. The correct implementation is as follows:

```c
char* copyString(const char* str)
{
    char* result = new char[strlen(str) + 1];
    return result;
}
```
One way to remember that `strlen()` returns only the number of actual characters in the string is to consider what would happen if you were allocating space for a string made up of several other strings. For example, if your function took in three strings and returned a string that was the concatenation of all three, how big would it be? To hold exactly enough space, it would be the length of all three strings, added together, plus one space for the trailing '\0' character. If `strlen()` included the '\0' in the length of the string, the allocated memory would be too big. The following code uses the `strcpy()` and `strcat()` functions to perform this operation. The `cat` in `strcat()` stands for concatenate.

```c
char* appendStrings(const char* str1, const char* str2, const char* str3)
{
    char* result = new char[strlen(str1) + strlen(str2) + strlen(str3) + 1];
    strcpy(result, str1);
    strcat(result, str2);
    strcat(result, str3);
    return result;
}
```

The `sizeof()` operator in C and C++ can be used to get the size of a certain data type or variable. For example, `sizeof(char)` returns 1 because a `char` has a size of 1 byte. However, in the context of C-style strings, `sizeof()` is not the same as `strlen()`. You should never use `sizeof()` to try to get the size of a string. It returns different sizes depending on how the C-style string is stored. If it is stored as a `char[]`, then `sizeof()` returns the actual memory used by the string, including the '\0' character, as in this example:

```c
char text1[] = "abcdef";
size_t s1 = sizeof(text1);  // is 7
size_t s2 = strlen(text1);  // is 6
```

However, if the C-style string is stored as a `char*`, then `sizeof()` returns the size of a pointer!

```c
const char* text2 = "abcdef";
size_t s3 = sizeof(text2);  // is platform-dependent
size_t s4 = strlen(text2);  // is 6
```

Here, `s3` will be 4 when compiled in 32-bit mode, and 8 when compiled in 64-bit mode because it is returning the size of a `const char*`, which is a pointer.

A complete list of C functions to operate on strings can be found in the `<cstring>` header file.

**WARNING** When you use the C-style string functions with Microsoft Visual Studio, the compiler is likely to give you security-related warnings or even errors about these functions being deprecated. You can eliminate these warnings by using other C Standard Library functions, such as `strcpy_s()` or `strcat_s()`, which are part of the “secure C library” standard (ISO/IEC TR 24731). However, the best solution is to switch to the C++ Standard Library `std::string` class, which we discuss in the section “The C++ std::string Class.”
String Literals

You’ve probably seen strings written in a C++ program with quotes around them. For example, the following code outputs the string `hello` by including the string itself, not a variable that contains it:

```cpp
cout << "hello" << endl;
```

In the preceding line, “`hello`” is a *string literal* because it is written as a value, not a variable. String literals are actually stored in a read-only part of memory. This allows the compiler to optimize memory usage by reusing references to equivalent string literals. That is, even if your program uses the string literal “`hello`” 500 times, the compiler is allowed to create just one instance of `hello` in memory. This is called *literal pooling*.

String literals can be *assigned* to variables, but because string literals are in a read-only part of memory and because of the possibility of literal pooling, assigning them to variables can be risky. The C++ standard officially says that string literals are of type “array of `n const char`”; however, for backward compatibility with older non-const-aware code, most compilers do not enforce your program to assign a string literal to a variable of type `const char*`. They let you assign a string literal to a `char*` without `const`, and the program will work fine unless you attempt to change the string. Generally, the behavior of modifying string literals is undefined. It could, for example, cause a crash, or it could keep working with seemingly inexplicable side effects, or the modification could silently be ignored, or it could just work; it all depends on your compiler. For example, the following code exhibits undefined behavior:

```cpp
char* ptr = "hello";       // Assign the string literal to a variable.
ptr[1] = 'a';              // Undefined behavior!
```

A much safer way to code is to use a pointer to `const` characters when referring to string literals. The following code contains the same bug, but because it assigned the literal to a `const char*`, the compiler catches the attempt to write to read-only memory:

```cpp
const char* ptr = "hello"; // Assign the string literal to a variable.
ptr[1] = 'a';              // Error! Attempts to write to read-only memory
```

You can also use a string literal as an initial value for a character array (`char[]`). In this case, the compiler creates an array that is big enough to hold the string and copies the string to this array. The compiler does not put the literal in read-only memory and does not do any literal pooling.

```cpp
char arr[] = "hello";     // Compiler takes care of creating appropriate sized
                         // character array arr.
arr[1] = 'a';             // The contents can be modified.
```

**Raw String Literals**

*Raw string literals* are string literals that can span across multiple lines of code, that don’t require escaping of embedded double quotes, and where escape sequences like `\t` and `\n` are processed as normal text and not as escape sequences. Escape sequences are discussed in Chapter 1. For example, if you write the following with a normal string literal, you will get a compilation error because the string contains non-escaped double quotes:

```cpp
const char* str = "Hello "World"!";   // Error!
```
Normally you have to escape the double quotes as follows:

```c
const char* str = "Hello \"World\"!";
```

With a raw string literal, you can avoid the need to escape the quotes. The raw string literal starts with `R"(` and ends with `)"`.

```c
const char* str = R"((Hello "World"!))";
```

If you need a string consisting of multiple lines, without raw string literals you need to embed \n escape sequences in your string where you want to start a new line. For example:

```c
const char* str = "Line 1
Line 2";
```

If you output this string to the console, you get the following:

```
Line 1
Line 2
```

With a raw string literal, instead of using \n escape sequences to start new lines, you can simply press enter to start real physical new lines in your source code as follows. The output is the same as the previous code snippet using the embedded \n.

```c
const char* str = R"((Line 1
Line 2))";
```

Escape sequences are ignored in raw string literals. For example, in the following raw string literal, the \t escape sequence is not replaced with a tab character, but is kept as the sequence of a back-slash followed by the letter t:

```c
const char* str = R"((Is the following a tab character? \t))";
```

So, if you output this string to the console, you get:

```
Is the following a tab character? \t
```

Because a raw string literal ends with `)"` you cannot embed a `)"` in your string using this syntax. For example, the following string is not valid because it contains the `)"` sequence in the middle of the string:

```c
const char* str = R"((Embedded )" characters)";       // Error!
```

If you need embedded `)"` characters, you need to use the extended raw string literal syntax, which is as follows:

```
R"d-char-sequence(r-char-sequence)d-char-sequence"
```

The `r-char-sequence` is the actual raw string. The `d-char-sequence` is an optional delimiter sequence, which should be the same at the beginning and at the end of the raw string literal. This delimiter sequence can have at most 16 characters. You should choose this delimiter sequence as a sequence that will not appear in the middle of your raw string literal.

The previous example can be rewritten using a unique delimiter sequence as follows:

```c
const char* str = R"-({Embedded }" characters)-";
```
Raw string literals make it easier to work with database querying strings, regular expressions, file paths, and so on. Regular expressions are discussed in Chapter 19.

The C++ std::string Class

C++ provides a much-improved implementation of the concept of a string as part of the Standard Library. In C++, std::string is a class (actually an instantiation of the basic_string class template) that supports many of the same functionalities as the <cstring> functions, but that takes care of memory allocation for you. The string class is defined in the <string> header in the std namespace, and has already been introduced in the previous chapter. Now it’s time to take a deeper look at it.

What Is Wrong with C-Style Strings?

To understand the necessity of the C++ string class, consider the advantages and disadvantages of C-style strings.

**Advantages:**
- They are simple, making use of the underlying basic character type and array structure.
- They are lightweight, taking up only the memory that they need if used properly.
- They are low level, so you can easily manipulate and copy them as raw memory.
- They are well understood by C programmers—why learn something new?

**Disadvantages:**
- They require incredible efforts to simulate a first-class string data type.
- They are unforgiving and susceptible to difficult-to-find memory bugs.
- They don’t leverage the object-oriented nature of C++.
- They require knowledge of their underlying representation on the part of the programmer.

The preceding lists were carefully constructed to make you think that perhaps there is a better way. As you’ll learn, C++ strings solve all the problems of C strings and render most of the arguments about the advantages of C strings over a first-class data type irrelevant.

Using the string Class

Even though string is a class, you can almost always treat it as if it were a built-in type. In fact, the more you think of it that way, the better off you are. Through the magic of operator overloading, C++ strings are much easier to use than C-style strings. For example, the + operator is redefined for strings to mean “string concatenation.” The following code produces 1234:

```cpp
string A("12");
string B("34");
string C;
C = A + B; // C is "1234"
```
The += operator is also overloaded to allow you to easily append a string:

```cpp
string A("12");
string B("34");
A += B;    // A is "1234"
```

Another problem with C strings is that you cannot use == to compare them. Suppose you have the following two strings:

```cpp
char* a = "12";
char b[] = "12";
```

Writing a comparison as follows always returns false, because it compares the pointer values, not the contents of the strings:

```cpp
if (a == b)
```

Note that C arrays and pointers are related. You can think of C arrays, like the b array in the example, as pointers to the first element in the array. Chapter 7 goes deeper in on the array-pointer duality.

To compare C strings, you have to write something as follows:

```cpp
if (strcmp(a, b) == 0)
```

Furthermore, there is no way to use <, <=, >=, or > to compare C strings, so `strcmp()` returns -1, 0, or 1, depending on the lexicographic relationship of the strings. This results in very clumsy and hard-to-read code, which is also error-prone.

With C++ strings, `operator==`, `operator!=`, `operator<`, and so on are all overloaded to work on the actual string characters. Individual characters can still be accessed with `operator[]`.

As the following code shows, when string operations require extending the string, the memory requirements are automatically handled by the string class, so memory overruns are a thing of the past:

```cpp
string myString = "hello";
myString += ", there";
string myOtherString = myString;
if (myString == myOtherString) {
    myOtherString[0] = 'H';
}
cout << myString << endl;
cout << myOtherString << endl;
```

The output of this code is

```
hello, there
Hello, there
```

There are several things to note in this example. One point is that there are no memory leaks even though strings are allocated and resized on a few places. All of these string objects are created as stack variables. While the string class certainly has a bunch of allocating and resizing to do, the string destructors clean up this memory when string objects go out of scope.
Another point to note is that the operators work the way you want them to. For example, the =
operator copies the strings, which is most likely what you want. If you are used to working with
array-based strings, this will either be refreshingly liberating for you or somewhat confusing. Don’t
worry—once you learn to trust the string class to do the right thing, life gets so much easier.

For compatibility, you can use the c_str() method on a string to get a const character pointer,
representing a C-style string. However, the returned const pointer becomes invalid whenever the
string has to perform any memory reallocation, or when the string object is destroyed. You
should call the method just before using the result so that it accurately reflects the current contents
of the string, and you must never return the result of c_str() called on a stack-based string
object from a function.

There is also a data() method which, up until C++14, always returned a const char* just as c_
str(). Starting with C++17, however, data() returns a char* when called on a non-const string.

Consult a Standard Library Reference, see Appendix B, for a complete list of all supported opera-
tions that you can perform on string objects.

std::string Literals

A string literal in source code is usually interpreted as a const char*. You can use the standard
user-defined literal “s” to interpret a string literal as an std::string instead.

```cpp
auto string1 = "Hello World"; // string1 is a const char*
auto string2 = "Hello World"s; // string2 is an std::string
```

The standard user-defined literal “s” requires a using namespace std::string_literals; or
using namespace std;.

High-Level Numeric Conversions

The std namespace includes a number of helper functions that make it easy to convert numeri-
cal values into strings or strings into numerical values. The following functions are available to
convert numerical values into strings. All these functions take care of memory allocations. A new
string object is created and returned from them.

```cpp
➤ string to_string(int val);
➤ string to_string(unsigned val);
➤ string to_string(long val);
➤ string to_string(unsigned long val);
➤ string to_string(long long val);
➤ string to_string(unsigned long long val);
➤ string to_string(float val);
➤ string to_string(double val);
➤ string to_string(long double val);
```