C++ Templates
The Complete Guide
SECOND EDITION

David Vandevoorde
Nicolai M. Josuttis
Douglas Gregor

FREE SAMPLE CHAPTER
SHARE WITH OTHERS
C++ Templates
The Complete Guide
Second Edition

David Vandevoorde
Nicolai M. Josuttis
Douglas Gregor
To Alessandra & Cassandra
   —David

To those who care
   for people and mankind
   —Nico

To Amy, Tessa & Molly
   —Doug
This page intentionally left blank
Contents

Preface xxiii
Acknowledgments for the Second Edition xxv
Acknowledgments for the First Edition xxvii
About This Book xxix

What You Should Know Before Reading This Book . . . . . . . . . . . . . . . . . . . . . . . xxx
Overall Structure of the Book . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxx
How to Read This Book . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxi
Some Remarks About Programming Style . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxi
The C++11, C++14, and C++17 Standards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxiii
Example Code and Additional Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxiv
Feedback . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xxxiv

Part I: The Basics 1

1 Function Templates 3

1.1 A First Look at Function Templates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.1.1 Defining the Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.1.2 Using the Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.1.3 Two-Phase Translation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.2 Template Argument Deduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.3 Multiple Template Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
1.3.1 Template Parameters for Return Types . . . . . . . . . . . . . . . . . . . . . . . . 10
1.3.2 Deducing the Return Type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
<table>
<thead>
<tr>
<th>viii</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.3.3 Return Type as Common Type .................................. 12</td>
</tr>
<tr>
<td></td>
<td>1.4 Default Template Arguments ...................................... 13</td>
</tr>
<tr>
<td></td>
<td>1.5 Overloading Function Templates .................................. 15</td>
</tr>
<tr>
<td></td>
<td>1.6 But, Shouldn’t We ...? ........................................... 20</td>
</tr>
<tr>
<td></td>
<td>1.6.1 Pass by Value or by Reference? ................................. 20</td>
</tr>
<tr>
<td></td>
<td>1.6.2 Why Not inline? .................................................. 20</td>
</tr>
<tr>
<td></td>
<td>1.6.3 Why Not constexpr? .............................................. 21</td>
</tr>
<tr>
<td></td>
<td>1.7 Summary ............................................................. 21</td>
</tr>
<tr>
<td></td>
<td>2 Class Templates ...................................................... 23</td>
</tr>
<tr>
<td></td>
<td>2.1 Implementation of Class Template Stack ......................... 23</td>
</tr>
<tr>
<td></td>
<td>2.1.1 Declaration of Class Templates ............................... 24</td>
</tr>
<tr>
<td></td>
<td>2.1.2 Implementation of Member Functions ......................... 26</td>
</tr>
<tr>
<td></td>
<td>2.2 Use of Class Template Stack ...................................... 27</td>
</tr>
<tr>
<td></td>
<td>2.3 Partial Usage of Class Templates ................................ 29</td>
</tr>
<tr>
<td></td>
<td>2.3.1 Concepts ......................................................... 29</td>
</tr>
<tr>
<td></td>
<td>2.4 Friends .............................................................. 30</td>
</tr>
<tr>
<td></td>
<td>2.5 Specializations of Class Templates ............................ 31</td>
</tr>
<tr>
<td></td>
<td>2.6 Partial Specialization ............................................ 33</td>
</tr>
<tr>
<td></td>
<td>2.7 Default Class Template Arguments ............................... 36</td>
</tr>
<tr>
<td></td>
<td>2.8 Type Aliases ........................................................ 38</td>
</tr>
<tr>
<td></td>
<td>2.9 Class Template Argument Deduction .............................. 40</td>
</tr>
<tr>
<td></td>
<td>2.10 Templatized Aggregates .......................................... 43</td>
</tr>
<tr>
<td></td>
<td>2.11 Summary ............................................................. 44</td>
</tr>
<tr>
<td></td>
<td>3 Nontype Template Parameters ......................................... 45</td>
</tr>
<tr>
<td></td>
<td>3.1 Nontype Class Template Parameters .............................. 45</td>
</tr>
<tr>
<td></td>
<td>3.2 Nontype Function Template Parameters ........................... 48</td>
</tr>
<tr>
<td></td>
<td>3.3 Restrictions for Nontype Template Parameters .................. 49</td>
</tr>
<tr>
<td></td>
<td>3.4 Template Parameter Type auto .................................... 50</td>
</tr>
<tr>
<td></td>
<td>3.5 Summary ............................................................. 54</td>
</tr>
</tbody>
</table>
## Contents ix

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Variadic Templates</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Variadic Templates</td>
<td>55</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Variadic Templates by Example</td>
<td>55</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Overloading Variadic and Nonvariadic Templates</td>
<td>57</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Operator sizeof</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Fold Expressions</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Application of Variadic Templates</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Variadic Class Templates and Variadic Expressions</td>
<td>61</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Variadic Expressions</td>
<td>62</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Variadic Indices</td>
<td>63</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Variadic Class Templates</td>
<td>63</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Variadic Deduction Guides</td>
<td>64</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Variadic Base Classes and using</td>
<td>65</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>Tricky Basics</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Keyword typename</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Zero Initialization</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>Using this-&gt;</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Templates for Raw Arrays and String Literals</td>
<td>71</td>
</tr>
<tr>
<td>5.5</td>
<td>Member Templates</td>
<td>74</td>
</tr>
<tr>
<td>5.5.1</td>
<td>The .template Construct</td>
<td>79</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Generic Lambdas and Member Templates</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>Variable Templates</td>
<td>80</td>
</tr>
<tr>
<td>5.7</td>
<td>Template Template Parameters</td>
<td>83</td>
</tr>
<tr>
<td>5.8</td>
<td>Summary</td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td>Move Semantics and enable_if&lt;&gt;</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Perfect Forwarding</td>
<td>91</td>
</tr>
<tr>
<td>6.2</td>
<td>Special Member Function Templates</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>Disable Templates with enable_if&lt;&gt;</td>
<td>98</td>
</tr>
<tr>
<td>6.4</td>
<td>Using enable_if&lt;&gt;</td>
<td>99</td>
</tr>
<tr>
<td>6.5</td>
<td>Using Concepts to Simplify enable_if&lt;&gt; Expressions</td>
<td>103</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6</td>
<td>Summary</td>
<td>104</td>
</tr>
<tr>
<td>7</td>
<td>By Value or by Reference?</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Passing by Value</td>
<td>106</td>
</tr>
<tr>
<td>7.2</td>
<td>Passing by Reference</td>
<td>108</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Passing by Constant Reference</td>
<td>108</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Passing by Nonconstant Reference</td>
<td>110</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Passing by Forwarding Reference</td>
<td>111</td>
</tr>
<tr>
<td>7.3</td>
<td>Using std::ref() and std::cref()</td>
<td>112</td>
</tr>
<tr>
<td>7.4</td>
<td>Dealing with String Literals and Raw Arrays</td>
<td>115</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Special Implementations for String Literals and Raw Arrays</td>
<td>116</td>
</tr>
<tr>
<td>7.5</td>
<td>Dealing with Return Values</td>
<td>117</td>
</tr>
<tr>
<td>7.6</td>
<td>Recommended Template Parameter Declarations</td>
<td>118</td>
</tr>
<tr>
<td>7.7</td>
<td>Summary</td>
<td>121</td>
</tr>
<tr>
<td>8</td>
<td>Compile-Time Programming</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Template Metaprogramming</td>
<td>123</td>
</tr>
<tr>
<td>8.2</td>
<td>Computing with constexpr</td>
<td>125</td>
</tr>
<tr>
<td>8.3</td>
<td>Execution Path Selection with Partial Specialization</td>
<td>127</td>
</tr>
<tr>
<td>8.4</td>
<td>SFINAE (Substitution Failure Is Not An Error)</td>
<td>129</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Expression SFINAE with decltype</td>
<td>133</td>
</tr>
<tr>
<td>8.5</td>
<td>Compile-Time if</td>
<td>134</td>
</tr>
<tr>
<td>8.6</td>
<td>Summary</td>
<td>135</td>
</tr>
<tr>
<td>9</td>
<td>Using Templates in Practice</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>The Inclusion Model</td>
<td>137</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Linker Errors</td>
<td>137</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Templates in Header Files</td>
<td>139</td>
</tr>
<tr>
<td>9.2</td>
<td>Templates and inline</td>
<td>140</td>
</tr>
<tr>
<td>9.3</td>
<td>Precompiled Headers</td>
<td>141</td>
</tr>
<tr>
<td>9.4</td>
<td>Decoding the Error Novel</td>
<td>143</td>
</tr>
<tr>
<td>9.5</td>
<td>Afternotes</td>
<td>149</td>
</tr>
<tr>
<td>9.6</td>
<td>Summary</td>
<td>150</td>
</tr>
</tbody>
</table>
Contents

10 Basic Template Terminology 151
  10.1 “Class Template” or “Template Class”? 151
  10.2 Substitution, Instantiation, and Specialization 152
  10.3 Declarations versus Definitions 153
    10.3.1 Complete versus Incomplete Types 154
  10.4 The One-Definition Rule 154
  10.5 Template Arguments versus Template Parameters 155
  10.6 Summary 156

11 Generic Libraries 157
  11.1 Callables 157
    11.1.1 Supporting Function Objects 158
    11.1.2 Dealing with Member Functions and Additional Arguments 160
    11.1.3 Wrapping Function Calls 162
  11.2 Other Utilities to Implement Generic Libraries 164
    11.2.1 Type Traits 164
    11.2.2 std::addressof() 166
    11.2.3 std::declval() 166
  11.3 Perfect Forwarding Temporaries 167
  11.4 References as Template Parameters 167
  11.5 Defer Evaluations 171
  11.6 Things to Consider When Writing Generic Libraries 172
  11.7 Summary 173

Part II: Templates in Depth 175

12 Fundamentals in Depth 177
  12.1 Parameterized Declarations 177
    12.1.1 Virtual Member Functions 182
    12.1.2 Linkage of Templates 182
    12.1.3 Primary Templates 184
  12.2 Template Parameters 185
    12.2.1 Type Parameters 185
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.2</td>
<td>Nontype Parameters</td>
<td>186</td>
</tr>
<tr>
<td>12.2.3</td>
<td>Template Template Parameters</td>
<td>187</td>
</tr>
<tr>
<td>12.2.4</td>
<td>Template Parameter Packs</td>
<td>188</td>
</tr>
<tr>
<td>12.2.5</td>
<td>Default Template Arguments</td>
<td>190</td>
</tr>
<tr>
<td>12.3</td>
<td>Template Arguments</td>
<td>192</td>
</tr>
<tr>
<td>12.3.1</td>
<td>Function Template Arguments</td>
<td>192</td>
</tr>
<tr>
<td>12.3.2</td>
<td>Type Arguments</td>
<td>194</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Nontype Arguments</td>
<td>194</td>
</tr>
<tr>
<td>12.3.4</td>
<td>Template Template Arguments</td>
<td>197</td>
</tr>
<tr>
<td>12.3.5</td>
<td>Equivalence</td>
<td>199</td>
</tr>
<tr>
<td>12.4</td>
<td>Variadic Templates</td>
<td>200</td>
</tr>
<tr>
<td>12.4.1</td>
<td>Pack Expansions</td>
<td>201</td>
</tr>
<tr>
<td>12.4.2</td>
<td>Where Can Pack Expansions Occur?</td>
<td>202</td>
</tr>
<tr>
<td>12.4.3</td>
<td>Function Parameter Packs</td>
<td>204</td>
</tr>
<tr>
<td>12.4.4</td>
<td>Multiple and Nested Pack Expansions</td>
<td>205</td>
</tr>
<tr>
<td>12.4.5</td>
<td>Zero-Length Pack Expansions</td>
<td>207</td>
</tr>
<tr>
<td>12.4.6</td>
<td>Fold Expressions</td>
<td>207</td>
</tr>
<tr>
<td>12.5</td>
<td>Friends</td>
<td>209</td>
</tr>
<tr>
<td>12.5.1</td>
<td>Friend Classes of Class Templates</td>
<td>209</td>
</tr>
<tr>
<td>12.5.2</td>
<td>Friend Functions of Class Templates</td>
<td>211</td>
</tr>
<tr>
<td>12.5.3</td>
<td>Friend Templates</td>
<td>213</td>
</tr>
<tr>
<td>12.6</td>
<td>Afternotes</td>
<td>213</td>
</tr>
<tr>
<td>13</td>
<td>Names in Templates</td>
<td>215</td>
</tr>
<tr>
<td>13.1</td>
<td>Name Taxonomy</td>
<td>215</td>
</tr>
<tr>
<td>13.2</td>
<td>Looking Up Names</td>
<td>217</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Argument-Dependent Lookup</td>
<td>219</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Argument-Dependent Lookup of Friend Declarations</td>
<td>220</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Injected Class Names</td>
<td>221</td>
</tr>
<tr>
<td>13.2.4</td>
<td>Current Instantiations</td>
<td>223</td>
</tr>
<tr>
<td>13.3</td>
<td>Parsing Templates</td>
<td>224</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Context Sensitivity in Nontemplates</td>
<td>225</td>
</tr>
<tr>
<td>13.3.2</td>
<td>Dependent Names of Types</td>
<td>228</td>
</tr>
<tr>
<td>13.3.3</td>
<td>Dependent Names of Templates</td>
<td>230</td>
</tr>
<tr>
<td>Contents</td>
<td>xiii</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>13.3.4 Dependent Names in Using Declarations</td>
<td>231</td>
<td></td>
</tr>
<tr>
<td>13.3.5 ADL and Explicit Template Arguments</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>13.3.6 Dependent Expressions</td>
<td>233</td>
<td></td>
</tr>
<tr>
<td>13.3.7 Compiler Errors</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>13.4 Inheritance and Class Templates</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>13.4.1 Nondependent Base Classes</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>13.4.2 Dependent Base Classes</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>13.5 Afternotes</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>14 Instantiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.1 On-Demand Instantiation</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>14.2 Lazy Instantiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.2.1 Partial and Full Instantiation</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>14.2.2 Instantiated Components</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>14.3 The C++ Instantiation Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.3.1 Two-Phase Lookup</td>
<td>249</td>
<td></td>
</tr>
<tr>
<td>14.3.2 Points of Instantiation</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>14.3.3 The Inclusion Model</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>14.4 Implementation Schemes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.4.1 Greedy Instantiation</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>14.4.2 Queried Instantiation</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>14.4.3 Iterated Instantiation</td>
<td>257</td>
<td></td>
</tr>
<tr>
<td>14.5 Explicit Instantiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.5.1 Manual Instantiation</td>
<td>259</td>
<td></td>
</tr>
<tr>
<td>14.5.2 Explicit Instantiation Declarations</td>
<td>260</td>
<td></td>
</tr>
<tr>
<td>14.6 Compile-Time if Statements</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>14.7 In the Standard Library</td>
<td>263</td>
<td></td>
</tr>
<tr>
<td>14.8 Afternotes</td>
<td>265</td>
<td></td>
</tr>
<tr>
<td>15 Template Argument Deduction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.1 The Deduction Process</td>
<td>269</td>
<td></td>
</tr>
<tr>
<td>15.2 Deduced Contexts</td>
<td>271</td>
<td></td>
</tr>
<tr>
<td>15.3 Special Deduction Situations</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>15.4 Initializer Lists</td>
<td>274</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>15.5</td>
<td>Parameter Packs</td>
<td>275</td>
</tr>
<tr>
<td>15.5.1</td>
<td>Literal Operator Templates</td>
<td>277</td>
</tr>
<tr>
<td>15.6</td>
<td>Rvalue References</td>
<td>277</td>
</tr>
<tr>
<td>15.6.1</td>
<td>Reference Collapsing Rules</td>
<td>277</td>
</tr>
<tr>
<td>15.6.2</td>
<td>Forwarding References</td>
<td>278</td>
</tr>
<tr>
<td>15.6.3</td>
<td>Perfect Forwarding</td>
<td>280</td>
</tr>
<tr>
<td>15.6.4</td>
<td>Deduction Surprises</td>
<td>283</td>
</tr>
<tr>
<td>15.7</td>
<td>SFINAE (Substitution Failure Is Not An Error)</td>
<td>284</td>
</tr>
<tr>
<td>15.7.1</td>
<td>Immediate Context</td>
<td>285</td>
</tr>
<tr>
<td>15.8</td>
<td>Limitations of Deduction</td>
<td>286</td>
</tr>
<tr>
<td>15.8.1</td>
<td>Allowable Argument Conversions</td>
<td>287</td>
</tr>
<tr>
<td>15.8.2</td>
<td>Class Template Arguments</td>
<td>288</td>
</tr>
<tr>
<td>15.8.3</td>
<td>Default Call Arguments</td>
<td>289</td>
</tr>
<tr>
<td>15.8.4</td>
<td>Exception Specifications</td>
<td>290</td>
</tr>
<tr>
<td>15.9</td>
<td>Explicit Function Template Arguments</td>
<td>291</td>
</tr>
<tr>
<td>15.10</td>
<td>Deduction from Initializers and Expressions</td>
<td>293</td>
</tr>
<tr>
<td>15.10.1</td>
<td>The auto Type Specifier</td>
<td>294</td>
</tr>
<tr>
<td>15.10.2</td>
<td>Expressing the Type of an Expression with decltype</td>
<td>298</td>
</tr>
<tr>
<td>15.10.3</td>
<td>decltype(auto)</td>
<td>301</td>
</tr>
<tr>
<td>15.10.4</td>
<td>Special Situations for auto Deduction</td>
<td>303</td>
</tr>
<tr>
<td>15.10.5</td>
<td>Structured Bindings</td>
<td>306</td>
</tr>
<tr>
<td>15.10.6</td>
<td>Generic Lambdas</td>
<td>309</td>
</tr>
<tr>
<td>15.11</td>
<td>Alias Templates</td>
<td>312</td>
</tr>
<tr>
<td>15.12</td>
<td>Class Template Argument Deduction</td>
<td>313</td>
</tr>
<tr>
<td>15.12.1</td>
<td>Deduction Guides</td>
<td>314</td>
</tr>
<tr>
<td>15.12.2</td>
<td>Implicit Deduction Guides</td>
<td>316</td>
</tr>
<tr>
<td>15.12.3</td>
<td>Other Subleties</td>
<td>318</td>
</tr>
<tr>
<td>15.13</td>
<td>Afternotes</td>
<td>321</td>
</tr>
<tr>
<td>16</td>
<td>Specialization and Overloading</td>
<td>323</td>
</tr>
<tr>
<td>16.1</td>
<td>When “Generic Code” Doesn’t Quite Cut It</td>
<td>323</td>
</tr>
<tr>
<td>16.1.1</td>
<td>Transparent Customization</td>
<td>324</td>
</tr>
<tr>
<td>16.1.2</td>
<td>Semantic Transparency</td>
<td>325</td>
</tr>
</tbody>
</table>
Contents xv

16.2 Overloading Function Templates ........................................ 326
  16.2.1 Signatures ....................................................... 328
  16.2.2 Partial Ordering of Overloaded Function Templates .......... 330
  16.2.3 Formal Ordering Rules ......................................... 331
  16.2.4 Templates and Nontemplates ................................. 332
  16.2.5 Variadic Function Templates .................................. 335

16.3 Explicit Specialization .............................................. 338
  16.3.1 Full Class Template Specialization .......................... 338
  16.3.2 Full Function Template Specialization ....................... 342
  16.3.3 Full Variable Template Specialization ...................... 344
  16.3.4 Full Member Specialization .................................. 344

16.4 Partial Class Template Specialization ............................ 347

16.5 Partial Variable Template Specialization ......................... 351

16.6 Afternotes .................................................................. 352

17 Future Directions ......................................................... 353
  17.1 Relaxed typename Rules ............................................. 354
  17.2 Generalized Nontype Template Parameters ...................... 354
  17.3 Partial Specialization of Function Templates .................. 356
  17.4 Named Template Arguments ....................................... 358
  17.5 Overloaded Class Templates ....................................... 359
  17.6 Deduction for Nonfinal Pack Expansions ....................... 360
  17.7 Regularization of void ............................................. 361
  17.8 Type Checking for Templates ..................................... 361
  17.9 Reflective Metaprogramming ..................................... 363
  17.10 Pack Facilities ........................................................ 365
  17.11 Modules ................................................................ 366

Part III: Templates and Design ............................................ 367

18 The Polymorphic Power of Templates ................................. 369
  18.1 Dynamic Polymorphism .............................................. 369
  18.2 Static Polymorphism .................................................. 372
# Contents

18.3 Dynamic versus Static Polymorphism ........................................... 375  
18.4 Using Concepts ................................................................. 377  
18.5 New Forms of Design Patterns ............................................... 379  
18.6 Generic Programming ......................................................... 380  
18.7 Afternotes ................................................................. 383  

19 Implementing Traits .............................................................. 385  
19.1 An Example: Accumulating a Sequence ...................................... 385  
  19.1.1 Fixed Traits ................................................................. 386  
  19.1.2 Value Traits ............................................................... 389  
  19.1.3 Parameterized Traits .................................................... 394  
19.2 Traits versus Policies and Policy Classes .................................. 394  
  19.2.1 Traits and Policies: What’s the Difference? ......................... 397  
  19.2.2 Member Templates versus Template Template Parameters .......... 398  
  19.2.3 Combining Multiple Policies and/or Traits .......................... 399  
  19.2.4 Accumulation with General Iterators ................................ 399  
19.3 Type Functions ...................................................................... 401  
  19.3.1 Element Types ............................................................... 401  
  19.3.2 Transformation Traits .................................................... 404  
  19.3.3 Predicate Traits ............................................................ 410  
  19.3.4 Result Type Traits .......................................................... 413  
19.4 SFINAE-Based Traits ................................................................ 416  
  19.4.1 SFINAE Out Function Overloads ....................................... 416  
  19.4.2 SFINAE Out Partial Specializations ................................... 420  
  19.4.3 Using Generic Lambdas for SFINAE ................................. 421  
  19.4.4 SFINAE-Friendly Traits .................................................. 424  
19.5 IsConvertibleT ...................................................................... 428  
19.6 Detecting Members .................................................................. 431  
  19.6.1 Detecting Member Types ................................................... 431  
  19.6.2 Detecting Arbitrary Member Types .................................... 433  
  19.6.3 Detecting Nontype Members ............................................. 434  
  19.6.4 Using Generic Lambdas to Detect Members ....................... 438  
19.7 Other Traits Techniques ......................................................... 440  
  19.7.1 If-Then-Else ................................................................. 440
## Contents

19.7.2 Detecting Nonthrowing Operations ........................................ 443
19.7.3 Traits Convenience ............................................................. 446
19.8 Type Classification ................................................................. 448
   19.8.1 Determining Fundamental Types ........................................ 448
   19.8.2 Determining Compound Types .......................................... 451
   19.8.3 Identifying Function Types ............................................. 454
   19.8.4 Determining Class Types ............................................... 456
   19.8.5 Determining Enumeration Types ..................................... 457
19.9 Policy Traits ........................................................................... 458
   19.9.1 Read-Only Parameter Types ............................................ 458
19.10 In the Standard Library ......................................................... 461
19.11 Afternotes ............................................................................. 462

20 Overloading on Type Properties .................................................. 465
   20.1 Algorithm Specialization ....................................................... 465
   20.2 Tag Dispatching .................................................................... 467
   20.3 Enabling/Disabling Function Templates .................................... 469
      20.3.1 Providing Multiple Specializations ................................. 471
      20.3.2 Where Does the EnableIf Go? ....................................... 472
      20.3.3 Compile-Time if ............................................................ 474
      20.3.4 Concepts .................................................................... 475
   20.4 Class Specialization ............................................................... 477
      20.4.1 Enabling/Disabling Class Templates ............................... 477
      20.4.2 Tag Dispatching for Class Templates ............................ 479
   20.5 Instantiation-Safe Templates .................................................. 482
   20.6 In the Standard Library ......................................................... 487
   20.7 Afternotes ............................................................................. 488

21 Templates and Inheritance ............................................................ 489
   21.1 The Empty Base Class Optimization (EBCO) ......................... 489
      21.1.1 Layout Principles ........................................................ 490
      21.1.2 Members as Base Classes .......................................... 492
   21.2 The Curiously Recurring Template Pattern (CRTP) .................. 495
      21.2.1 The Barton-Nackman Trick ....................................... 497
| 21.2.2  | Operator Implementations | 500 |
| 21.2.3  | Facades                  | 501 |
| 21.3    | Mixins                   | 508 |
| 21.3.1  | Curious Mixins           | 510 |
| 21.3.2  | Parameterized Virtuality | 510 |
| 21.4    | Named Template Arguments | 512 |
| 21.5    | Afternotes               | 515 |

**22 Bridging Static and Dynamic Polymorphism**

| 22.1    | Function Objects, Pointers, and `std::function<>` | 517 |
| 22.2    | Generalized Function Pointers                     | 519 |
| 22.3    | Bridge Interface                                   | 522 |
| 22.4    | Type Erasure                                       | 523 |
| 22.5    | Optional Bridging                                  | 525 |
| 22.6    | Performance Considerations                         | 527 |
| 22.7    | Afternotes                                         | 528 |

**23 Metaprogramming**

| 23.1    | The State of Modern C++ Metaprogramming            | 529 |
| 23.1.1  | Value Metaprogramming                              | 529 |
| 23.1.2  | Type Metaprogramming                               | 531 |
| 23.1.3  | Hybrid Metaprogramming                             | 532 |
| 23.1.4  | Hybrid Metaprogramming for Unit Types              | 534 |
| 23.2    | The Dimensions of Reflective Metaprogramming      | 537 |
| 23.3    | The Cost of Recursive Instantiation                | 539 |
| 23.3.1  | Tracking All Instantiations                        | 540 |
| 23.4    | Computational Completeness                         | 542 |
| 23.5    | Recursive Instantiation versus Recursive Template Arguments | 542 |
| 23.6    | Enumeration Values versus Static Constants         | 543 |
| 23.7    | Afternotes                                         | 545 |

**24 Typelists**

| 24.1    | Anatomy of a Typelist                              | 549 |
## Contents

24.2 Typelist Algorithms .................................................. 551  
24.2.1 Indexing ......................................................... 551  
24.2.2 Finding the Best Match ........................................ 552  
24.2.3 Appending to a Typelist ........................................ 555  
24.2.4 Reversing a Typelist ........................................... 557  
24.2.5 Transforming a Typelist ........................................ 559  
24.2.6 Accumulating Typelists ........................................ 560  
24.2.7 Insertion Sort ................................................... 563  
24.3 Nontype Typelists ................................................... 566  
24.3.1 Deducible Nontype Parameters ................................ 568  
24.4 Optimizing Algorithms with Pack Expansions ...................... 569  
24.5 Cons-style Typelists ................................................ 571  
24.6 Afternotes .......................................................... 573  

25 Tuples ............................................................................. 575  
25.1 Basic Tuple Design .................................................... 576  
25.1.1 Storage ............................................................ 576  
25.1.2 Construction ....................................................... 578  
25.2 Basic Tuple Operations ................................................ 579  
25.2.1 Comparison ........................................................ 579  
25.2.2 Output .............................................................. 580  
25.3 Tuple Algorithms ....................................................... 581  
25.3.1 Tuples as Typelists ................................................. 581  
25.3.2 Adding to and Removing from a Tuple ......................... 582  
25.3.3 Reversing a Tuple ................................................ 584  
25.3.4 Index Lists ......................................................... 585  
25.3.5 Reversal with Index Lists ....................................... 586  
25.3.6 Shuffle and Select ............................................... 588  
25.4 Expanding Tuples ...................................................... 592  
25.5 Optimizing Tuple ....................................................... 593  
25.5.1 Tuples and the EBCO ............................................. 593  
25.5.2 Constant-time get() ............................................. 598  
25.6 Tuple Subscript ......................................................... 599  
25.7 Afternotes ............................................................ 601
## 26 Discriminated Unions

26.1 Storage .......................................................... 604
26.2 Design ............................................................ 606
26.3 Value Query and Extraction .................................... 610
26.4 Element Initialization, Assignment and Destruction ........ 611
  26.4.1 Initialization .................................................. 611
  26.4.2 Destruction ................................................... 612
  26.4.3 Assignment ................................................... 613
26.5 Visitors .......................................................... 617
  26.5.1 Visit Result Type ........................................... 621
  26.5.2 Common Result Type ........................................ 622
26.6 Variant Initialization and Assignment ......................... 624
26.7 Afternotes ....................................................... 628

## 27 Expression Templates

27.1 Temporaries and Split Loops .................................... 630
27.2 Encoding Expressions in Template Arguments ................ 635
  27.2.1 Operands of the Expression Templates .................... 636
  27.2.2 The Array Type ............................................. 639
  27.2.3 The Operators ............................................... 642
  27.2.4 Review ....................................................... 643
  27.2.5 Expression Templates Assignments ...................... 645
27.3 Performance and Limitations of Expression Templates ........ 646
27.4 Afternotes ....................................................... 647

## 28 Debugging Templates

28.1 Shallow Instantiation ........................................... 652
28.2 Static Assertions ............................................... 654
28.3 Archetypes ....................................................... 655
28.4 Tracers .......................................................... 657
28.5 Oracles .......................................................... 662
28.6 Afternotes ....................................................... 662
Appendixes 663

A The One-Definition Rule 663
A.1 Translation Units ........................................... 663
A.2 Declarations and Definitions .................................... 664
A.3 The One-Definition Rule in Detail ............................... 665
  A.3.1 One-per-Program Constraints ............................. 665
  A.3.2 One-per-Translation Unit Constraints ...................... 667
  A.3.3 Cross-Translation Unit Equivalence Constraints ............ 669

B Value Categories 673
B.1 Traditional Lvalues and Rvalues ............................... 673
  B.1.1 Lvalue-to-Rvalue Conversions ........................... 674
B.2 Value Categories Since C++11 .................................. 674
  B.2.1 Temporary Materialization ............................... 676
B.3 Checking Value Categories with decltype ...................... 678
B.4 Reference Types ............................................... 679

C Overload Resolution 681
C.1 When Does Overload Resolution Kick In? ...................... 681
C.2 Simplified Overload Resolution ............................... 682
  C.2.1 The Implied Argument for Member Functions ................ 684
  C.2.2 Refining the Perfect Match .............................. 686
C.3 Overloading Details ........................................... 688
  C.3.1 Prefer Nontemplates or More Specialized Templates ......... 688
  C.3.2 Conversion Sequences .................................... 689
  C.3.3 Pointer Conversions ..................................... 689
  C.3.4 Initializer Lists ......................................... 691
  C.3.5 Functors and Surrogate Functions ........................ 694
  C.3.6 Other Overloading Contexts .............................. 695

D Standard Type Utilities 697
D.1 Using Type Traits ............................................. 697
  D.1.1 std::integral_constant and std::bool_constant ............ 698
  D.1.2 Things You Should Know When Using Traits ................ 700
### Contents

**D.2 Primary and Composite Type Categories**  
D.2.1 Testing for the Primary Type Category  
D.2.2 Test for Composite Type Categories  

**D.3 Type Properties and Operations**  
D.3.1 Other Type Properties  
D.3.2 Test for Specific Operations  
D.3.3 Relationships Between Types  

**D.4 Type Construction**  

**D.5 Other Traits**  

**D.6 Combining Type Traits**  

**D.7 Other Utilities**  

**E Concepts**  
E.1 Using Concepts  
E.2 Defining Concepts  
E.3 Overloading on Constraints  
E.3.1 Constraint Subsumption  
E.3.2 Constraints and Tag Dispatching  
E.4 Concept Tips  
E.4.1 Testing Concepts  
E.4.2 Concept Granularity  
E.4.3 Binary Compatibility  

**Bibliography**  
Forums  
Books and Web Sites  

**Glossary**  

**Index**
Preface

The notion of templates in C++ is over 30 years old. C++ templates were already documented in 1990 in “The Annotated C++ Reference Manual” (ARM; see [EllisStroustrupARM]), and they had been described before then in more specialized publications. However, well over a decade later, we found a dearth of literature that concentrates on the fundamental concepts and advanced techniques of this fascinating, complex, and powerful C++ feature. With the first edition of this book, we wanted to address this issue and decided to write the book about templates (with perhaps a slight lack of humility).

Much has changed in C++ since that first edition was published in late 2002. New iterations of the C++ standard have added new features, and continued innovation in the C++ community has uncovered new template-based programming techniques. The second edition of this book therefore retains the same goals as the first edition, but for “Modern C++.”

We approached the task of writing this book with different backgrounds and with different intentions. David (aka “Daveed”), an experienced compiler implementer and active participant of the C++ Standard Committee working groups that evolve the core language, was interested in a precise and detailed description of all the power (and problems) of templates. Nico, an “ordinary” application programmer and member of the C++ Standard Committee Library Working Group, was interested in understanding all the techniques of templates in a way that he could use and benefit from them. Doug, a template library developer turned compiler implementer and language designer, was interested in collecting, categorizing, and evaluating the myriad techniques used to build template libraries. In addition, we all wanted to share this knowledge with you, the reader, and the whole community to help to avoid further misunderstanding, confusion, or apprehension.

As a consequence, you will see both conceptual introductions with day-to-day examples and detailed descriptions of the exact behavior of templates. Starting from the basic principles of templates and working up to the “art of template programming,” you will discover (or rediscover) techniques such as static polymorphism, type traits, metaprogramming, and expression templates. You will also gain a deeper understanding of the C++ standard library, in which almost all code involves templates.

We learned a lot and we had much fun while writing this book. We hope you will have the same experience while reading it. Enjoy!
This page intentionally left blank
Acknowledgments for the Second Edition

Writing a book is hard. Maintaining a book is even harder. It took us more than five years—spread over the past decade—to come up with this second edition, and it couldn’t have been done without the support and patience of a lot of people.

First, we’d like to thank everyone in the C++ community and on the C++ standardization committee. In addition to all the work to add new language and library features, they spent many, many hours explaining and discussing their work with us, and they did so with patience and enthusiasm.

Part of this community also includes the programmers who gave feedback for errors and possible improvement for the first edition over the past 15 years. There are simply too many to list them all, but you know who you are and we’re truly grateful to you for taking the time to write up your thoughts and observations. Please accept our apologies if our answers were sometimes less than prompt.

We’d also like to thank everyone who reviewed drafts of this book and provided us with valuable feedback and clarifications. These reviews brought the book to a significantly higher level of quality, and it again proved that good things need the input of many “wise guys.” For this reason, huge thanks to Steve Dewhurst, Howard Hinnant, Mikael Kilpeläinen, Dietmar Kühl, Daniel Krügler, Nevin Lieber, Andreas Neiser, Eric Niebler, Richard Smith, Andrew Sutton, Hubert Tong, and Ville Voutilainen.

Of course, thanks to all the people who supported us from Addison-Wesley/Pearson. These days, you can no longer take professional support for book authors for granted. But they were patient, nagged us when appropriate, and were of great help when knowledge and professionalism were necessary. So, many thanks to Peter Gordon, Kim Boedigheimer, Greg Doench, Julie Nahil, Dana Wilson, and Carol Lallier.

A special thanks goes to the LaTeX community for a great text system and to Frank Mittelbach for solving our LaTeX issues (it was almost always our fault).
David’s Acknowledgments for the Second Edition

This second edition was a long time in the waiting, and as we put the finishing touches to it, I am grateful for the people in my life who made it possible despite many obligations vying for attention. First, I’m indebted to my wife (Karina) and daughters (Alessandra and Cassandra), for agreeing to let me take significant time out of the “family schedule” to complete this edition, particularly in the last year of work. My parents have always shown interest in my goals, and whenever I visit them, they do not forget this particular project.

Clearly, this is a technical book, and its contents reflect knowledge and experience about a programming topic. However, that is not enough to pull off completing this kind of work. I’m therefore extremely grateful to Nico for having taken upon himself the “management” and “production” aspects of this edition (in addition to all of his technical contributions). If this book is useful to you and you run into Nico some day, be sure to tell him thanks for keeping us all going. I’m also thankful to Doug for having agreed to come on board several years ago and to keep going even as demands on his own schedule made for tough going.

Many programmers in our C++ community have shared nuggets of insight over the years, and I am grateful to all of them. However, I owe special thanks to Richard Smith, who has been efficiently answering my e-mails with arcane technical issues for years now. In the same vein, thanks to my colleagues John Spicer, Mike Miller, and Mike Herrick, for sharing their knowledge and creating an encouraging work environment that allows us to learn ever more.

Nico’s Acknowledgments for the Second Edition

First, I want to thank the two hard-core experts, David and Doug, because, as an application programmer and library expert, I asked so many silly questions and learned so much. I now feel like becoming a core expert (only until the next issue, of course). It was fun, guys.

All my other thanks go to Jutta Eckstein. Jutta has the wonderful ability to force and support people in their ideals, ideas, and goals. While most people experience this only occasionally when meeting her or working with her in our IT industry, I have the honor to benefit from her in my day-to-day life. After all these years, I still hope this will last forever.

Doug’s Acknowledgments for the Second Edition

My heartfelt thanks go to my wonderful and supportive wife, Amy, and our two little girls, Molly and Tessa. Their love and companionship bring me daily joy and the confidence to tackle the greatest challenges in life and work. I’d also like to thank my parents, who instilled in me a great love of learning and encouraged me throughout these years.

It was a pleasure to work with both David and Nico, who are so different in personality yet complement each other so well. David brings a great clarity to technical writing, honing in on descriptions that are precise and illuminating. Nico, beyond his exceptional organizational skills that kept two coauthors from wandering off into the weeds, brings a unique ability to take apart a complex technical discussion and make it simpler, more accessible, and far, far clearer.
Acknowledgments for the First Edition

This book presents ideas, concepts, solutions, and examples from many sources. We’d like to thank all the people and companies who helped and supported us during the past few years.

First, we’d like to thank all the reviewers and everyone else who gave us their opinion on early manuscripts. These people endow the book with a quality it would never have had without their input. The reviewers for this book were Kyle Blaney, Thomas Gschwind, Dennis Mancl, Patrick Mc Killen, and Jan Christiaan van Winkel. Special thanks to Dietmar Kühl, who meticulously reviewed and edited the whole book. His feedback was an incredible contribution to the quality of this book.

We’d also like to thank all the people and companies who gave us the opportunity to test our examples on different platforms with different compilers. Many thanks to the Edison Design Group for their great compiler and their support. It was a big help during the standardization process and the writing of this book. Many thanks also go to all the developers of the free GNU and egcs compilers (Jason Merrill was especially responsive) and to Microsoft for an evaluation version of Visual C++ (Jonathan Caves, Herb Sutter, and Jason Shirk were our contacts there).

Much of the existing “C++ wisdom” was collectively created by the online C++ community. Most of it comes from the moderated Usenet groups comp.lang.c++.moderated and comp.std.c++. We are therefore especially indebted to the active moderators of those groups, who keep the discussions useful and constructive. We also much appreciate all those who over the years have taken the time to describe and explain their ideas for us all to share.

The Addison-Wesley team did another great job. We are most indebted to Debbie Lafferty (our editor) for her gentle prodding, good advice, and relentless hard work in support of this book. Thanks also go to Tyrrell Albaugh, Bunny Ames, Melanie Buck, Jacquelyn Doucette, Chanda Leary-Coutu, Catherine Ohala, and Marty Rabinowitz. We’re grateful as well to Marina Lang, who first sponsored this book within Addison-Wesley. Susan Winer contributed an early round of editing that helped shape our later work.
Nico’s Acknowledgments for the First Edition

My first personal thanks go with a lot of kisses to my family: Ulli, Lucas, Anica, and Frederic supported this book with a lot of patience, consideration, and encouragement.

In addition, I want to thank David. His expertise turned out to be incredible, but his patience was even better (sometimes I ask really silly questions). It is a lot of fun to work with him.

David’s Acknowledgments for the First Edition

My wife, Karina, has been instrumental in this book coming to a conclusion, and I am immensely grateful for the role that she plays in my life. Writing “in your spare time” quickly becomes erratic when many other activities vie for your schedule. Karina helped me to manage that schedule, taught me to say “no” in order to make the time needed to make regular progress in the writing process, and above all was amazingly supportive of this project. I thank God every day for her friendship and love.

I’m also tremendously grateful to have been able to work with Nico. Besides his directly visible contributions to the text, his experience and discipline moved us from my pitiful doodling to a well-organized production.

John “Mr. Template” Spicer and Steve “Mr. Overload” Adamczyk are wonderful friends and colleagues, but in my opinion they are (together) also the ultimate authority regarding the core C++ language. They clarified many of the trickier issues described in this book, and should you find an error in the description of a C++ language element, it is almost certainly attributable to my failing to consult with them.

Finally, I want to express my appreciation to those who were supportive of this project without necessarily contributing to it directly (the power of cheer cannot be understated). First, my parents: Their love for me and their encouragement made all the difference. And then there are the numerous friends inquiring: “How is the book going?” They, too, were a source of encouragement: Michael Beckmann, Brett and Julie Beene, Jarran Carr, Simon Chang, Ho and Sarah Cho, Christophe De Dinechin, Ewa Deelman, Neil Eberle, Sassan Hazeghi, Vikram Kumar, Jim and Lindsay Long, R.J. Morgan, Mike Puritano, Ragu Raghavendra, Jim and Phuong Sharp, Gregg Vaughn, and John Wiegley.
About This Book

The first edition of this book was published almost 15 years ago. We had set out to write the definitive
guide to C++ templates, with the expectation that it would be useful to practicing C++ programmers.
That project was successful: It’s been tremendously gratifying to hear from readers who found our
material helpful, to see our book time and again being recommended as a work of reference, and to
be universally well reviewed.

That first edition has aged well, with most material remaining entirely relevant to the modern C++
programmer, but there is no denying that the evolution of the language—culminating in the “Modern
C++” standards, C++11, C++14, and C++17—has raised the need for a revision of the material in
the first edition.

So with this second edition, our high-level goal has remained unchanged: to provide the definitive
guide to C++ templates, including both a solid reference and an accessible tutorial. This time, how-
ever, we work with the “Modern C++” language, which is a significantly bigger beast (still!) than the
language available at the time of the prior edition.

We’re also acutely aware that C++ programming resources have changed (for the better) since the
first edition was published. For example, several books have appeared that develop specific template-
based applications in great depth. More important, far more information about C++ templates and
template-based techniques is easily available online, as are examples of advanced uses of these tech-
niques. So in this edition, we have decided to emphasize a breadth of techniques that can be used in
a variety of applications.

Some of the techniques we presented in the first edition have become obsolete because the C++
language now offers more direct ways of achieving the same effects. Those techniques have been
dropped (or relegated to minor notes), and instead you’ll find new techniques that show the state-of-
the-art uses of the new language.

We’ve now lived over 20 years with C++ templates, but the C++ programmers’ community still
regularly finds new fundamental insights into the way they can fit in our software development needs.
Our goal with this book is to share that knowledge but also to fully equip the reader to develop new
understanding and, perhaps, discover the next major C++ technique.
What You Should Know Before Reading This Book

To get the most from this book, you should already know C++. We describe the details of a particular language feature, not the fundamentals of the language itself. You should be familiar with the concepts of classes and inheritance, and you should be able to write C++ programs using components such as IOstreams and containers from the C++ standard library. You should also be familiar with the basic features of “Modern C++”, such as auto, decltype, move semantics, and lambdas. Nevertheless, we review more subtle issues as the need arises, even when such issues aren’t directly related to templates. This ensures that the text is accessible to experts and intermediate programmers alike.

We deal primarily with the C++ language revisions standardized in 2011, 2014, and 2017. However, at the time of this writing, the ink is barely dry on the C++17 revision, and we expect that most of our readers will not be intimately familiar with its details. All revisions had a significant impact on the behavior and usage of templates. We therefore provide short introductions to those new features that have the greatest bearing on our subject matter. However, our goal is neither to introduce the modern C++ standards nor to provide an exhaustive description of the changes from the prior versions of this standard ([C++98] and [C++03]). Instead, we focus on templates as designed and used in C++, using the modern C++ standards ([C++11], [C++14], and [C++17]) as our basis, and we occasionally call out cases where the modern C++ standards enable or encourage different techniques than the prior standards.

Overall Structure of the Book

Our goal is to provide the information necessary to start using templates and benefit from their power, as well as to provide information that will enable experienced programmers to push the limits of the state-of-the-art. To achieve this, we decided to organize our text in parts:

- Part I introduces the basic concepts underlying templates. It is written in a tutorial style.
- Part II presents the language details and is a handy reference to template-related constructs.
- Part III explains fundamental design and coding techniques supported by C++ templates. They range from near-trivial ideas to sophisticated idioms.

Each of these parts consists of several chapters. In addition, we provide a few appendixes that cover material not exclusively related to templates (e.g., an overview of overload resolution in C++). An additional appendix covers concepts, which is a fundamental extension to templates that has been included in the draft for a future standard (C++20, presumably).

The chapters of Part I are meant to be read in sequence. For example, Chapter 3 builds on the material covered in Chapter 2. In the other parts, however, the connection between chapters is considerably looser. Cross references will help readers jump through the different topics.

Last, we provide a rather complete index that encourages additional ways to read this book out of sequence.
How to Read This Book

If you are a C++ programmer who wants to learn or review the concepts of templates, carefully read Part I, The Basics. Even if you’re quite familiar with templates already, it may help to skim through this part quickly to familiarize yourself with the style and terminology that we use. This part also covers some of the logistical aspects of organizing your source code when it contains templates.

Depending on your preferred learning method, you may decide to absorb the many details of templates in Part II, or instead you could read about practical coding techniques in Part III (and refer back to Part II for the more subtle language issues). The latter approach is probably particularly useful if you bought this book with concrete day-to-day challenges in mind.

The appendices contain much useful information that is often referred to in the main text. We have also tried to make them interesting in their own right.

In our experience, the best way to learn something new is to look at examples. Therefore, you’ll find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files at the Web site of this book at http://www.tmplbook.com.

Some Remarks About Programming Style

C++ programmers use different programming styles, and so do we: The usual questions about where to put whitespace, delimiters (braces, parentheses), and so forth came up. We tried to be consistent in general, although we occasionally make concessions to the topic at hand. For example, in tutorial sections, we may prefer generous use of whitespace and concrete names to help visualize code, whereas in more advanced discussions, a more compact style could be more appropriate.

We do want to draw your attention to one slightly uncommon decision regarding the declaration of types, parameters, and variables. Clearly, several styles are possible:

```cpp
void foo (const int &x);
void foo (const int& x);
void foo (int const &x);
void foo (int const& x);
```

Although it is a bit less common, we decided to use the order `int const` rather than `const int` for “constant integer.” We have two reasons for using this order. First, it provides for an easier answer to the question, “What is constant?” It’s always what is in front of the `const` qualifier. Indeed, although

```cpp
const int N = 100;
```

is equivalent to

```cpp
int const N = 100;
```

there is no equivalent form for

```cpp
int* const bookmark; // the pointer cannot change, but the value pointed to can
```
that would place the const qualifier before the pointer operator *. In this example, it is the pointer itself that is constant, not the int to which it points.

Our second reason has to do with a syntactical substitution principle that is very common when dealing with templates. Consider the following two type declarations using the typedef keyword:

```cpp
typedef char* CHARS;
typedef CHARS const CPTR; // constant pointer to chars
```

or using the using keyword:

```cpp
using CHARS = char*;
using CPTR = CHARS const; // constant pointer to chars
```

The meaning of the second declaration is preserved when we textually replace CHARS with what it stands for:

```cpp
typedef char* const CPTR; // constant pointer to chars
```

or:

```cpp
using CPTR = char* const; // constant pointer to chars
```

However, if we write const before the type it qualifies, this principle doesn’t apply. Consider the alternative to our first two type definitions presented earlier:

```cpp
typedef char* CHARS;
typedef const CHARS CPTR; // constant pointer to chars
```

Textually replacing CHARS results in a type with a different meaning:

```cpp
typedef const char* CPTR; // pointer to constant chars
```

The same observation applies to the volatile specifier, of course.

Regarding whitespaces, we decided to put the space between the ampersand and the parameter name:

```cpp
void foo (int const& x);
```

By doing this, we emphasize the separation between the parameter type and the parameter name. This is admittedly more confusing for declarations such as

```cpp
char* a, b;
```

where, according to the rules inherited from C, a is a pointer but b is an ordinary char. To avoid such confusion, we simply avoid declaring multiple entities in this way.

This is primarily a book about language features. However, many techniques, features, and helper templates now appear in the C++ standard library. To connect these two, we therefore demonstrate

---

1 Note that in C++, a type definition defines a “type alias” rather than a new type (see Section 2.8 on page 38). For example:

```cpp
typedef int Length; // define Length as an alias for int
int i = 42;
Length l = 88;
i = l; // OK
l = i; // OK
```
template techniques by illustrating how they are used to implement certain library components, and we use standard library utilities to build our own more complex examples. Hence, we use not only headers such as `<iostream>` and `<string>` (which contain templates but are not particularly relevant to define other templates) but also `<cstdlib>`, `<utility>`, `<functional>`, and `<type_traits>` (which do provide building blocks for more complex templates).

In addition, we provide a reference, Appendix D, about the major template utilities provided by the C++ standard library, including a detailed description of all the standard type traits. These are commonly used at the core of sophisticated template programming.

The C++11, C++14, and C++17 Standards

The original C++ standard was published in 1998 and subsequently amended by a technical corrigendum in 2003, which provided minor corrections and clarifications to the original standard. This “old C++ standard” is known as C++98 or C++03.

The C++11 standard was the first major revision of C++ driven by the ISO C++ standardization committee, bringing a wealth of new features to the language. A number of these new features interact with templates and are described in this book, including:

- Variadic templates
- Alias templates
- Move semantics, rvalue references, and perfect forwarding
- Standard type traits

C++14 and C++17 followed, both introducing some new language features, although the changes brought about by these standards were not quite as dramatic as those of C++11. New features interacting with templates and described in this book include but are not limited to:

- Variable templates (C++14)
- Generic Lambdas (C++14)
- Class template argument deduction (C++17)
- Compile-time if (C++17)
- Fold expressions (C++17)

We even describe concepts (template interfaces), which are currently slated for inclusion in the forthcoming C++20 standard.

At the time of this writing, the C++11 and C++14 standards are broadly supported by the major compilers, and C++17 is largely supported also. Still, compilers differ greatly in their support of the different language features. Several will compile most of the code in this book, but a few compilers may not be able to handle some of our examples. However, we expect that this problem will soon be resolved as programmers everywhere demand standard support from their vendors.

---

2 The committee now aims at issuing a new standard roughly every 3 years. Clearly, that leaves less time for massive additions, but it brings the changes more quickly to the broader programming community. The development of larger features, then, spans time and might cover multiple standards.
Even so, the C++ programming language is likely to continue to evolve as time passes. The experts of the C++ community (regardless of whether they participate in the C++ Standardization Committee) are discussing various ways to improve the language, and already several candidate improvements affect templates. Chapter 17 presents some trends in this area.

Example Code and Additional Information

You can access all example programs and find more information about this book from its Web site, which has the following URL:

http://www.tmplbook.com

Feedback

We welcome your constructive input—both the negative and the positive. We worked very hard to bring you what we hope you’ll find to be an excellent book. However, at some point we had to stop writing, reviewing, and tweaking so we could “release the product.” You may therefore find errors, inconsistencies, and presentations that could be improved, or topics that are missing altogether. Your feedback gives us a chance to inform all readers through the book’s Web site and to improve any subsequent editions.

The best way to reach us is by email. You will find the email address at the Web site of this book:

http://www.tmplbook.com

Please, be sure to check the book’s Web site for the currently known errata before submitting reports. Many thanks.
Chapter 4

Variadic Templates

Since C++11, templates can have parameters that accept a variable number of template arguments. This feature allows the use of templates in places where you have to pass an arbitrary number of arguments of arbitrary types. A typical application is to pass an arbitrary number of parameters of arbitrary type through a class or framework. Another application is to provide generic code to process any number of parameters of any type.

4.1 Variadic Templates

Template parameters can be defined to accept an unbounded number of template arguments. Templates with this ability are called variadic templates.

4.1.1 Variadic Templates by Example

For example, you can use the following code to call print() for a variable number of arguments of different types:

```cpp
#include <iostream>

void print ()
{
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    std::cout << firstArg << '\n';  // print first argument
    print(args...);  // call print() for remaining arguments
}
```
If one or more arguments are passed, the function template is used, which by specifying the first argument separately allows printing of the first argument before recursively calling print() for the remaining arguments. These remaining arguments named `args` are a function parameter pack:

```cpp
void print (T firstArg, Types... args)
```

using different “Types” specified by a template parameter pack:

```cpp
template<typename T, typename... Types>
```

To end the recursion, the nontemplate overload of `print()` is provided, which is issued when the parameter pack is empty.

For example, a call such as

```cpp
std::string s("world");
print (7.5, "hello", s);
```

would output the following:

```
7.5
hello
world
```

The reason is that the call first expands to

```cpp
print<double, char const*, std::string> (7.5, "hello", s);
```

with

- `firstArg` having the value 7.5 so that type `T` is a `double` and
- `args` being a variadic template argument having the values "hello" of type `char const*` and "world" of type `std::string`.

After printing 7.5 as `firstArg`, it calls `print()` again for the remaining arguments, which then expands to:

```cpp
print<char const*, std::string> ("hello", s);
```

with

- `firstArg` having the value "hello" so that type `T` is a `char const*` here and
- `args` being a variadic template argument having the value of type `std::string`.

After printing "hello" as `firstArg`, it calls `print()` again for the remaining arguments, which then expands to:

```cpp
print<std::string> (s);
```

with

- `firstArg` having the value "world" so that type `T` is a `std::string` now and
- `args` being an empty variadic template argument having no value.

Thus, after printing "world" as `firstArg`, we calls `print()` with no arguments, which results in calling the nontemplate overload of `print()` doing nothing.
4.1 Variadic Templates

4.1.2 Overloading Variadic and Nonvariadic Templates

Note that you can also implement the example above as follows:

```cpp
#include <iostream>

template<typename T>
void print (T arg)
{
    std::cout << arg << 'n'; // print passed argument
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    print(firstArg); // call print() for the first argument
    print(args...); // call print() for remaining arguments
}
```

That is, if two function templates only differ by a trailing parameter pack, the function template without the trailing parameter pack is preferred.\(^\text{1}\) Section \text{C.3.1} on page 688 explains the more general overload resolution rule that applies here.

4.1.3 Operator `sizeof...`

C++11 also introduced a new form of the `sizeof` operator for variadic templates: `sizeof...`. It expands to the number of elements a parameter pack contains. Thus,

```cpp
template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    std::cout << sizeof...(Types) << 'n'; // print number of remaining types
    std::cout << sizeof...(args) << '\n'; // print number of remaining args
    ...
}
```

twice prints the number of remaining arguments after the first argument passed to `print()`. As you can see, you can call `sizeof...` for both template parameter packs and function parameter packs.

This might lead us to think we can skip the function for the end of the recursion by not calling it in case there are no more arguments:

\(^\text{1}\) Initially, in C++11 and C++14 this was an ambiguity, which was fixed later (see [CoreIssue1395]), but all compilers handle it this way in all versions.
However, this approach doesn’t work because in general both branches of all if statements in function templates are instantiated. Whether the instantiated code is useful is a run-time decision, while the instantiation of the call is a compile-time decision. For this reason, if you call the print() function template for one (last) argument, the statement with the call of print(args...) still is instantiated for no argument, and if there is no function print() for no arguments provided, this is an error.

However, note that since C++17, a compile-time if is available, which achieves what was expected here with a slightly different syntax. This will be discussed in Section 8.5 on page 134.

### 4.2 Fold Expressions

Since C++17, there is a feature to compute the result of using a binary operator over all the arguments of a parameter pack (with an optional initial value). For example, the following function returns the sum of all passed arguments:

```cpp
template<typename... T>
auto foldSum (T... s) {
    return (... + s);  // ((s1 + s2) + s3)...
}
```

If the parameter pack is empty, the expression is usually ill-formed (with the exception that for operator && the value is true, for operator || the value is false, and for the comma operator the value for an empty parameter pack is void()).

Table 4.1 lists the possible fold expressions.

<table>
<thead>
<tr>
<th>Fold Expression</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ... op pack )</td>
<td>(((pack1 op pack2) op pack3) ... op packN)</td>
</tr>
<tr>
<td>( pack op ... )</td>
<td>(pack1 op (... (packN-1 op packN)))</td>
</tr>
<tr>
<td>( init op ... op pack )</td>
<td>(((init op pack1) op pack2) ... op packN)</td>
</tr>
<tr>
<td>( pack op ... op init )</td>
<td>(pack1 op (... (packN op init)))</td>
</tr>
</tbody>
</table>

Table 4.1. Fold Expressions (since C++17)

You can use almost all binary operators for fold expressions (see Section 12.4.6 on page 208 for details). For example, you can use a fold expression to traverse a path in a binary tree using operator 

```cpp
->*;
```
4.2 Fold Expressions

basics/foldtraverse.cpp

```cpp
// define binary tree structure and traverse helpers:
struct Node {
  int value;
  Node* left;
  Node* right;
  Node(int i=0) : value(i), left(nullptr), right(nullptr) {
  }
  ...
};
auto left = &Node::left;
auto right = &Node::right;

// traverse tree, using fold expression:
template<typename T, typename... TP>
Node* traverse (T np, TP... paths) {
  return (np ->* ... ->* paths);  // np ->* paths1 ->* paths2 ...
}

int main()
{
  // init binary tree structure:
  Node* root = new Node{0};
  root->left = new Node{1};
  root->left->right = new Node{2};
  ...
  // traverse binary tree:
  Node* node = traverse(root, left, right);
  ...
}
```

Here,

```
(np ->* ... ->* paths)
```

uses a fold expression to traverse the variadic elements of `paths` from `np`.

With such a fold expression using an initializer, we might think about simplifying the variadic template to print all arguments, introduced above:

```cpp
template<typename... Types>
void print (Types const&... args)
{
  (std::cout << ... << args) << \n;
}
```
However, note that in this case no whitespace separates all the elements from the parameter pack. To do that, you need an additional class template, which ensures that any output of any argument is extended by a space:

```cpp
template<typename T>
class AddSpace
{
  private:
    T const& ref;              // refer to argument passed in constructor
  public:
    AddSpace(T const& r): ref(r) {}
    friend std::ostream& operator<< (std::ostream& os, AddSpace<T> s) {
      return os << s.ref << ' ';  // output passed argument and a space
    }
};
```

```cpp
template<typename... Args>
void print (Args... args) {
  ( std::cout << ... << AddSpace(args) ) << 'n';
}
```

Note that the expression `AddSpace(args)` uses class template argument deduction (see Section 2.9 on page 40) to have the effect of `AddSpace<Args>(args)`, which for each argument creates an `AddSpace` object that refers to the passed argument and adds a space when it is used in output expressions.

See Section 12.4.6 on page 207 for details about fold expressions.

### 4.3 Application of Variadic Templates

Variadic templates play an important role when implementing generic libraries, such as the C++ standard library.

One typical application is the forwarding of a variadic number of arguments of arbitrary type. For example, we use this feature when:

- Passing arguments to the constructor of a new heap object owned by a shared pointer:
  ```cpp
  // create shared pointer to complex<float> initialized by 4.2 and 7.7:
  auto sp = std::make_shared<std::complex<float>>(4.2, 7.7);
  ```

- Passing arguments to a thread, which is started by the library:
  ```cpp
  std::thread t (foo, 42, "hello");  // call foo(42,"hello") in a separate thread
  ```
4.4 Variadic Class Templates and Variadic Expressions

- Passing arguments to the constructor of a new element pushed into a vector:
  ```cpp
  std::vector<Customer> v;
  ...
  v.emplace("Tim", "Jovi", 1962); // insert a Customer initialized by three arguments
  ```

Usually, the arguments are “perfectly forwarded” with move semantics (see Section 6.1 on page 91), so that the corresponding declarations are, for example:

```cpp
namespace std {
  template<typename T, typename... Args> shared_ptr<T>
  make_shared(Args&&... args);

  class thread {
    public:
      template<typename F, typename... Args>
      explicit thread(F&& f, Args&&... args);
    ...
  };

  template<typename T, typename Allocator = allocator<T>>
  class vector {
    public:
      template<typename... Args> reference emplace_back(Args&&... args);
    ...
  };
}
```

Note also that the same rules apply to variadic function template parameters as for ordinary parameters. For example, if passed by value, arguments are copied and decay (e.g., arrays become pointers), while if passed by reference, parameters refer to the original parameter and don’t decay:

```cpp
// args are copies with decayed types:
template<typename... Args> void foo (Args... args);
// args are nondecayed references to passed objects:
template<typename... Args> void bar (Args const&... args);
```

### 4.4 Variadic Class Templates and Variadic Expressions

Besides the examples above, parameter packs can appear in additional places, including, for example, expressions, class templates, using declarations, and even deduction guides. Section 12.4.2 on page 202 has a complete list.
4.4.1 Variadic Expressions

You can do more than just forward all the parameters. You can compute with them, which means to compute with all the parameters in a parameter pack.

For example, the following function doubles each parameter of the parameter pack args and passes each doubled argument to print():

```cpp
template<typename... T>
void printDoubled (T const&... args)
{
    print (args + args...);
}
```

If, for example, you call

```cpp
printDoubled(7.5, std::string("hello"), std::complex<float>(4,2));
```

the function has the following effect (except for any constructor side effects):

```cpp
print(7.5 + 7.5,
    std::string("hello") + std::string("hello"),
    std::complex<float>(4,2) + std::complex<float>(4,2);
```

If you just want to add 1 to each argument, note that the dots from the ellipsis may not directly follow a numeric literal:

```cpp
template<typename... T>
void addOne (T const&... args)
{
    print (args + 1...);
    // ERROR: 1... is a literal with too many decimal points
    print (args + 1 ...);  // OK
    print ((args + 1)...);  // OK
}
```

Compile-time expressions can include template parameter packs in the same way. For example, the following function template returns whether the types of all the arguments are the same:

```cpp
template<typename T1, typename... TN>
constexpr bool isHomogeneous (T1, TN...)
{
    return (std::is_same<T1,TN>::value && ...
        // since C++17
}
```

This is an application of fold expressions (see Section 4.2 on page 58): For

```cpp
isHomogeneous(43, -1, "hello")
```

the expression for the return value expands to

```cpp
std::is_same<int,int>::value && std::is_same<int,char const*>::value
```

and yields false, while

```cpp
isHomogeneous("hello", " ", "world", "!")
```

yields true because all passed arguments are deduced to be char const* (note that the argument types decay because the call arguments are passed by value).
4.4 Variadic Class Templates and Variadic Expressions

4.4.2 Variadic Indices

As another example, the following function uses a variadic list of indices to access the corresponding element of the passed first argument:

```cpp
template<typename C, typename... Idx>
void printElems (C const& coll, Idx... idx)
{
    print (coll[idx]...);
}
```

That is, when calling

```cpp
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printElems(coll, 2, 0, 3);
```

the effect is to call

```cpp
print (coll[2], coll[0], coll[3]);
```

You can also declare nontype template parameters to be parameter packs. For example:

```cpp
template<std::size_t... Idx, typename C>
void printIdx (C const& coll)
{
    print (coll[Idx]...);
}
```

allows you to call

```cpp
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printIdx<2, 0, 3>(coll);
```

which has the same effect as the previous example.

4.4.3 Variadic Class Templates

Variadic templates can also be class templates. An important example is a class where an arbitrary number of template parameters specify the types of corresponding members:

```cpp
template<typename... Elements>
class Tuple;
```

```cpp
Tuple<int, std::string, char> t;  // t can hold integer, string, and character
```

This will be discussed in Chapter 25.

Another example is to be able to specify the possible types objects can have:

```cpp
template<typename... Types>
class Variant;
```

```cpp
Variant<int, std::string, char> v;  // v can hold integer, string, or character
```

This will be discussed in Chapter 26.
You can also define a class that as a type represents a list of indices:

// type for arbitrary number of indices:

```cpp
template<std::size_t...>
struct Indices {
};
```

This can be used to define a function that calls `print()` for the elements of a `std::array` or `std::tuple` using the compile-time access with `get<>()` for the given indices:

```cpp
template<typename T, std::size_t... Idx>
void printByIdx(T t, Indices<Idx...>)
{
    print(std::get<Idx>(t)...);
}
```

This template can be used as follows:

```cpp
std::array<std::string, 5> arr = {"Hello", "my", "new", "!", "World"};
printByIdx(arr, Indices<0, 4, 3>());
```

or as follows:

```cpp
auto t = std::make_tuple(12, "monkeys", 2.0);
printByIdx(t, Indices<0, 1, 2>());
```

This is a first step towards meta-programming, which will be discussed in Section 8.1 on page 123 and Chapter 23.

### 4.4.4 Variadic Deduction Guides

Even deduction guides (see Section 2.9 on page 42) can be variadic. For example, the C++ standard library defines the following deduction guide for `std::array`:

```cpp
namespace std {
    template<typename T, typename... U> array(T, U...) -> array<enable_if_t<(is_same_v<T, U1> && ...), T>,
                                                  (1 + sizeof...(U))>;
}
```

An initialization such as

```cpp
std::array<int, 3> a{42, 45, 77};
```

deduces `T` in the guide to the type of the element, and the various `U...` types to the types of the subsequent elements. The total number of elements is therefore `1 + sizeof...(U)`:

```cpp
std::array<int, 3> a{42, 45, 77};
```

The `std::enable_if<>` expression for the first array parameter is a fold expression that (as introduced as `isHomogeneous()` in Section 4.4.1 on page 62) expands to:

```cpp
is_same_v<T, U1> && is_same_v<T, U2> && is_same_v<T, U3> ...
```

If the result is not `true` (i.e., not all the element types are the same), the deduction guide is discarded and the overall deduction fails. This way, the standard library ensures that all elements must have the same type for the deduction guide to succeed.
4.4 Variadic Class Templates and Variadic Expressions

4.4.5 Variadic Base Classes and \texttt{using}

Finally, consider the following example:

```cpp
// basics/varusing.cpp
#include <string>
#include <unordered_set>

class Customer
{
    private:
        std::string name;
    public:
        Customer(std::string const& n) : name(n) {} 
        std::string getName() const { return name; }
};

struct CustomerEq {
    bool operator() (Customer const& c1, Customer const& c2) const {
        return c1.getName() == c2.getName();
    }
};

struct CustomerHash {
    std::size_t operator() (Customer const& c) const {
        return std::hash<std::string>()(c.getName());
    }
};

// define class that combines \texttt{operator()} for variadic base classes:
template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator...; // OK since C++17
};

int main()
{
    // combine hasher and equality for customers in one type:
    using CustomerOP = Overloader<CustomerHash,CustomerEq>;

    std::unordered_set<Customer,CustomerHash,CustomerEq> coll1;
    std::unordered_set<Customer,CustomerOP,CustomerOP> coll2;
    ...
}
```
Here, we first define a class `Customer` and independent function objects to hash and compare `Customer` objects. With

```cpp
template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator()...;  // OK since C++17
};
```

we can define a class derived from a variadic number of base classes that brings in the `operator()` declarations from each of those base classes. With

```cpp
using CustomerOP = Overloader<CustomerHash,CustomerEq>;
```

we use this feature to derive `CustomerOP` from `CustomerHash` and `CustomerEq` and enable both implementations of `operator()` in the derived class.

See Section 26.4 on page 611 for another application of this technique.

### 4.5 Summary

- By using parameter packs, templates can be defined for an arbitrary number of template parameters of arbitrary type.
- To process the parameters, you need recursion and/or a matching nonvariadic function.
- Operator `sizeof...` yields the number of arguments provided for a parameter pack.
- A typical application of variadic templates is forwarding an arbitrary number of arguments of arbitrary type.
- By using fold expressions, you can apply operators to all arguments of a parameter pack.
Index

-> 249
<
parsing 225
>
in template argument list 50
parsing 225
>>
versus >> 28, 226
[ ] 685

A

ABC 759, see abstract base class
about the book xxix
Abrahams, David 515, 547, 573
AbrahamsGurtovoyMeta 750
abstract base class 369
  as concept 377
abstract class 759
ACCU 750
actual parameter 155
Adamczyk, Steve 321, 352
adapter
  iterator 505
add_const 729
add_cv 729
add_lvalue_reference 730
add_pointer 730
addressof 166, 737
add_rvalue_reference 730
add_volatile 729
ADL 217, 218, 219, 759
aggregate 692
  template 43
  trait 711
Alexandrescu, Andrei 266, 397, 463, 547, 573, 601, 628
AlexandrescuAdHocVisitor 750
AlexandrescuDesign 750
AlexandrescuDiscriminatedUnions 750
algorithm specialization 465, 557
alias declaration 38
alias template 39, 312, 446
  as member 178
drawbacks 446
  specialization 338
aligned_storage 733, 734
aligned_union 733
alignment_of 715
allocator 85, 462
angle bracket
  hack 28, 226, 759
angle brackets 4, 760
  parsing 225
anonymous union 246
ANSI 760
apply 592
archetype 655
argument 155, 192, 760
  by value or by reference 20
  conversions 287
  deduction 269, see argument
deduction
  derived class 495
  for function templates 192
  for template template parameters 85, 197
match 682
named 512
nontype arguments 194
type arguments 194
versus parameter 155
argument deduction 7
  for class templates 40
  for function templates 10
argument-dependent lookup 217, 218, 219, 760
argument list
  operator> 50
argument pack 200
array 43
array
  as parameter in templates 71
  as template parameter 186
  conversion to pointer 107, 270
array
deduction guide 64
array
  passing 115
  qualification 453
Array<>
  635
assignment operator
  as template 79
  with type conversion 74
associated class 219
associated namespace 219
AusternSTL 750
auto 12
auto&& 167
auto 294
  and initializer lists 303
  as template parameter 50, 296
deduction 303
return type 11, 296
automatic instantiation 243
avoiding deduction 497
B
back() for vectors 26
back() for vectors 23
baggage 462
Barton, John J. 497, 515, 547
BartonNackman 751
Barton-Nackman trick
  versus CRTP 515
base class
  conversion to 689
  dependent 70, 237, 238
  duplicate 513
  empty 489
  nondependent 236
  parameterized by derived class 495
  variadic 65
Batory, Don S. 516
BCCL 751
bibliography 749
binary compatibility
  with concepts 747
binary right fold 208
bitset 79
Blinn, Frank 516
Blitz++ 751
books 749
bool
contextually convertible to 485
  conversion to 689
BoolConstant 411
bool_constant 699
BoostAny 751
Boost 751
BoostFusion 751
BoostHana 751
BoostIterator 751
Index C

BoostMPL 751
BoostOperators 752
BoostOptional 752
BoostSmartPtr 752
BoostTuple 752
BoostTypeTraits 752
BoostVariant 752
Borland 256, 649
bounded polymorphism 375
bridge pattern 379
Bright, Walter 266
Brown, Walter E. 547
BrownSIunits 752
by value vs. by reference 105, 638

C

C++03 752
C++11 753
C++14 753
C++17 753
C++98 752
CacciolaKrzemienski2013 753
call
default arguments 289
parameter 9
callable 157
callback 157
CargillExceptionSafety 753
category
composite type 702
for values 673
primary type 702
char*
as template argument 49, 354
char_traits 462
Chochlík, Matúš 547
chrono library 534
class 4, 185, 760
associated 219
as template argument 49
definition 668
dependent base 237
name injection 221
nondependent base 236
policy class 394
qualification 456
template see class template
versus struct 151
class template 23, 151, 760
argument deduction 40
as member 74, 178
declaration 24, 177
default argument 36
enable_if 477
friend 75, 209
full specialization 338
overloading 359
parameters 288
partial specialization 347
tag dispatching 479
class type 151, 760
qualification 456
C linkage 183
closure 310, 422
code bloat 348
code layout principles 490
collapsing references 277
collection class 760, see container
common_type 12, 622, 732
compiler 760
compile-time if 134, 263, 474
compiling 255, 651
models 243
complete type 154, 245, 760
complex 6
composite type (category) 702
computation
metaprogramming 538
concept 29, 103, 651, 654, 739, 760
abstract base class 377
binary compatibility 747
definition 742
disable function templates 475
with static_assert 29
conditional 171, 443, 732
evaluation of unused branches 442
implementation 440
ensurehelvetica/sembedded_()
avoiding 497
class template arguments 40
for rvalue references 277
from default arguments 8, 289
function template arguments 7
of forwarding references 278
deduction guide 42, 314
explicit 319
for aggregates 43
guided type 42, 314
variadic 64
default argument see default argument
call argument deduction 8, 289
for template template parameter 197
nontype parameter 48
default argument depending on following arguments 621
for call parameters 180
for class templates 36
for function templates 13
for templates 190
for template template parameters 85
defer evaluation 171
definition 3, 153, 664, 762
of class types 668
of concepts 742
versus declaration 153, 664
definition time 6
dependent base class 237, 762
dependent expression 233
dependent name 215, 217, 762
in using declarations 231
of templates 230
of types 228
dependent type 223, 228
deque 85
derivation 236, 489
derived class as base class argument 495
design 367
design pattern 379
DesignPatternsGoF 753
determining types 448, 460
digraph 227, 762
Dimov, Peter 488
Dionne, Louis 463, 547
directive for explicit instantiation 260
discriminated union 603
disjunction 736
dispatching tag 467, 487
domination rule 515
Dos Reis, Gabriel 214, 321
DosReisMarcusAliasTemplates 754
double as template argument 49, 356
as traits value 391
zero initialization 68
duplicate base class 513
dynamic polymorphism 369, 375
E
EBCDIC 387
EBCO 489, 593, 762
and tuples 593
EDG 266
EDG 754
Edison Design Group 266, 352
Eisenecker, Ulrich 516, 573
EiseneckerBlinnCzarnecki 754
ellipsis 417, 683
EllisStroustrupARM 754
duplicate email to the authors xxxiv
empty() for vectors 23
empty base class optimization 489, 593, 762
EnableIf
placement 472
enable_if 98, 732
and parameter packs 735
class templates 477
disable copy constructor 102
implementation 469
placement 472
entity templated 181
enumeration
  qualification 457
  versus static constant 543
erasure of types 523
error handling 651
error message 143
evaluation deferred 171
exception
  safety 26
  specifications 290
expansion
  restricted 497
explicit
  in deduction guide 319
explicit generic initialization 69
explicit instantiation
  declaration 262
  definition 260
  directive 260, 762
explicit specialization 152, 224, 228, 238, 338, 762
explicit template argument 233
exported templates 266
expression
  dependent 233
  instantiation-dependent 234
  type-dependent 217, 233
  value-dependent 234
expression template 629, 762
  limitations 646
  performance 646
extended Koenig lookup 242
extent 110, 715

F
facade 501
FalseType 411
false_type 413, 699
  implementation 411
file organization 137
final 493
fixed traits 386
float
  as template argument 49, 356
  as traits value 391
  zero initialization 68
fold expression 58, 207
  deduction guide 64
formal parameter 155
forums 749
forward declaration 244
forwarding
  metafunction 407, 447, 452, 552
  perfect 91, 280
forwarding reference 93, 111, 763
friend 185, 209
  class 209
  class templates 75
  function 211
  function versus function template 499
  name injection 221, 241, 498, 763
  template 213
full specialization 338, 763
  of class templates 338
  of function templates 342
  of member function template 78
  of member templates 344
function 517, 519
  as template parameter 186
  dispatch table 257
  for types 401
  qualification 454
  signature 328
  spilled inline 257
  surrogate 694
  template  see function template
  function call wrapper 162
  function object 157, 763
    and overloading 694
  function object type 157
  function parameter pack 204
  function pointer 517
  FunctionPtr 519
Index G

function template 3, 151, 763
  argument deduction 10
  arguments 192
  as member 74, 178
  declaration 177
  default call argument 180
  default template argument 13
  friend 211
  full specialization 342
  inline 20, 140
  nontype parameters 48
  overloading 15, 326
  partial specialization 356
  versus friend function 499
functor 763
  and overloading 694
fundamental type
  qualification 448
future template features 353

G

generated specialization 152
generation
  metaprogramming 538
generic lambda 80, 309
  for SFINAE 421
generic programming 380
get() for tuples 598
Gibbons, Bill 241, 515
glossary 759
gvalue 674, 763
greedy instantiation 256
Gregor, Doug 214
GregorJarviPowellVariadicTemplates 754
guard for header files 667
guided type 42, 314
Gurtovoy, Aleksey 547, 573

H

Hartinger, Roland 358
HasDereference 654
has_unique_object_representations 714
has_virtual_destructor 714
header file 137, 763
  guard 667
  order 141
  precompiled 141
  std.hpp 142
Henney, Kevlin 528
HenneyValuedConversions 754
heterogeneous collection 376
Hewlett-Packard 241, 352
higher-order genericity 214
Hinnant, Howard 488, 547
hybrid metaprogramming 532

I

identifier 216
if
  compile-time 263
  constexpr 134, 263, 474
  IfThenElseT<> 440, 541
  immediate context 285
  implicit instantiation 243
  INCITS 763
include file 764, see header file
  #include order 141
  inclusion model 139, 254
  incomplete type 154, 171, 764
    using traits 734
  index list 570, 586
  index sequence 570, 586
indirect call 764
inheritance 236, 489
  domination rule 515
  duplicate base class 513
initialization
  explicit 69
  of fundamental types 68
initializer 764
initializer list 69, 764
  and auto 303
  and overloading 691
  initializer_list
    and overloading 691
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>deduction</td>
<td>274</td>
</tr>
<tr>
<td>injected</td>
<td></td>
</tr>
<tr>
<td>class name</td>
<td>221, 764</td>
</tr>
<tr>
<td>friend name</td>
<td>221, 241, 498</td>
</tr>
<tr>
<td>inline</td>
<td>20, 140</td>
</tr>
<tr>
<td>and full specialization</td>
<td>78</td>
</tr>
<tr>
<td>for variables</td>
<td>178</td>
</tr>
<tr>
<td>inline variable</td>
<td>392</td>
</tr>
<tr>
<td>instance</td>
<td>6, 764</td>
</tr>
<tr>
<td>instantiated specialization</td>
<td>152</td>
</tr>
<tr>
<td>instantiation</td>
<td>5, 6, 152, 243, 764</td>
</tr>
<tr>
<td>automatic</td>
<td>243</td>
</tr>
<tr>
<td>costs</td>
<td>539</td>
</tr>
<tr>
<td>current</td>
<td>223</td>
</tr>
<tr>
<td>explicit definition</td>
<td>260</td>
</tr>
<tr>
<td>explicit directive</td>
<td>260</td>
</tr>
<tr>
<td>greedy</td>
<td>256</td>
</tr>
<tr>
<td>implicit</td>
<td>243</td>
</tr>
<tr>
<td>iterated</td>
<td>259</td>
</tr>
<tr>
<td>lazy</td>
<td>245</td>
</tr>
<tr>
<td>levels</td>
<td>542</td>
</tr>
<tr>
<td>manual</td>
<td>260</td>
</tr>
<tr>
<td>mechanisms</td>
<td>243</td>
</tr>
<tr>
<td>model</td>
<td>249</td>
</tr>
<tr>
<td>on-demand</td>
<td>243</td>
</tr>
<tr>
<td>point</td>
<td>250</td>
</tr>
<tr>
<td>queried</td>
<td>257</td>
</tr>
<tr>
<td>recursive</td>
<td>542</td>
</tr>
<tr>
<td>shallow</td>
<td>652</td>
</tr>
<tr>
<td>virtual</td>
<td>246</td>
</tr>
<tr>
<td>instantiation-dependent expression</td>
<td>234</td>
</tr>
<tr>
<td>instantiation-safe template</td>
<td>482</td>
</tr>
<tr>
<td>instantiation time</td>
<td>6</td>
</tr>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>277</td>
</tr>
<tr>
<td>parsing literals</td>
<td>599</td>
</tr>
<tr>
<td>zero initialization</td>
<td>68</td>
</tr>
<tr>
<td>integer_sequence</td>
<td>586</td>
</tr>
<tr>
<td>integral_constant</td>
<td>566</td>
</tr>
<tr>
<td>integral_constant</td>
<td>698</td>
</tr>
<tr>
<td>Internet resources</td>
<td>749</td>
</tr>
<tr>
<td>intrusive</td>
<td>375</td>
</tr>
<tr>
<td>invasive</td>
<td>375</td>
</tr>
<tr>
<td>invoke()</td>
<td>160</td>
</tr>
<tr>
<td>trait</td>
<td>716</td>
</tr>
<tr>
<td>invoke_result</td>
<td>163, 717</td>
</tr>
<tr>
<td>is_abstract</td>
<td>714</td>
</tr>
<tr>
<td>is_aggregate</td>
<td>711</td>
</tr>
<tr>
<td>is_arithmetic</td>
<td>707</td>
</tr>
<tr>
<td>is_array</td>
<td>110, 704</td>
</tr>
<tr>
<td>IsArrayT&lt;&gt;</td>
<td>453</td>
</tr>
<tr>
<td>is_assignable</td>
<td>722</td>
</tr>
<tr>
<td>is_base_of</td>
<td>726</td>
</tr>
<tr>
<td>is_callable</td>
<td>716</td>
</tr>
<tr>
<td>is_class</td>
<td>705</td>
</tr>
<tr>
<td>IsClassT&lt;&gt;</td>
<td>456</td>
</tr>
<tr>
<td>is_compound</td>
<td>707</td>
</tr>
<tr>
<td>is_const</td>
<td>709</td>
</tr>
<tr>
<td>is_constructible</td>
<td>719</td>
</tr>
<tr>
<td>is_convertible</td>
<td>727</td>
</tr>
<tr>
<td>implementation</td>
<td>428</td>
</tr>
<tr>
<td>IsConvertibleT</td>
<td>428</td>
</tr>
<tr>
<td>is_copy_assignable</td>
<td>722</td>
</tr>
<tr>
<td>is_copy_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_default_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_destructible</td>
<td>724</td>
</tr>
<tr>
<td>is_empty</td>
<td>714</td>
</tr>
<tr>
<td>is_enum</td>
<td>705</td>
</tr>
<tr>
<td>IsEnumT&lt;&gt;</td>
<td>457</td>
</tr>
<tr>
<td>is_final</td>
<td>714</td>
</tr>
<tr>
<td>is_floating_point</td>
<td>703</td>
</tr>
<tr>
<td>is_function</td>
<td>706</td>
</tr>
<tr>
<td>is_fundamental</td>
<td>707</td>
</tr>
<tr>
<td>IsFundamentalT&lt;&gt;</td>
<td>448</td>
</tr>
<tr>
<td>is_integral</td>
<td>703</td>
</tr>
<tr>
<td>is_invocable</td>
<td>716</td>
</tr>
<tr>
<td>is_invocable_r</td>
<td>716</td>
</tr>
<tr>
<td>is_literal_type</td>
<td>713</td>
</tr>
<tr>
<td>is_lvalue_reference</td>
<td>705</td>
</tr>
<tr>
<td>IsLValueReferenceT&lt;&gt;</td>
<td>452</td>
</tr>
<tr>
<td>is_member_function_pointer</td>
<td>705</td>
</tr>
<tr>
<td>is_member_object_pointer</td>
<td>705</td>
</tr>
<tr>
<td>is_member_pointer</td>
<td>706</td>
</tr>
<tr>
<td>is_move_assignable</td>
<td>723</td>
</tr>
<tr>
<td>is_move_constructible</td>
<td>721</td>
</tr>
<tr>
<td>is_nothrow_assignable</td>
<td>722</td>
</tr>
<tr>
<td>is_nothrow_constructible</td>
<td>719</td>
</tr>
<tr>
<td>is_nothrow_copy_assignable</td>
<td>722</td>
</tr>
<tr>
<td>Index</td>
<td>779</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>is_nothrow_copy_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_nothrow_default_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_nothrow_destructible</td>
<td>724</td>
</tr>
<tr>
<td>is_nothrow_invocable</td>
<td>716</td>
</tr>
<tr>
<td>is_nothrow_invocable_r</td>
<td>716</td>
</tr>
<tr>
<td>is_nothrow_move_assignable</td>
<td>723</td>
</tr>
<tr>
<td>is_nothrow_move_constructible</td>
<td>721</td>
</tr>
<tr>
<td>is_nothrow_swappable</td>
<td>725</td>
</tr>
<tr>
<td>is_nothrow_swappable_with</td>
<td>724</td>
</tr>
<tr>
<td>is_null_pointer</td>
<td>704</td>
</tr>
<tr>
<td>ISO</td>
<td>764</td>
</tr>
<tr>
<td>is_object</td>
<td>707</td>
</tr>
<tr>
<td>isocpp.org</td>
<td>750</td>
</tr>
<tr>
<td>is_pod</td>
<td>713</td>
</tr>
<tr>
<td>is_pointer</td>
<td>704</td>
</tr>
<tr>
<td>IsPointerT&lt;&gt;</td>
<td>451</td>
</tr>
<tr>
<td>is_polymorphic</td>
<td>714</td>
</tr>
<tr>
<td>IsPtrMemT&lt;&gt;</td>
<td>454</td>
</tr>
<tr>
<td>is_reference</td>
<td>706</td>
</tr>
<tr>
<td>IsReferenceT&lt;&gt;</td>
<td>452</td>
</tr>
<tr>
<td>is_rvalue_reference</td>
<td>705</td>
</tr>
<tr>
<td>IsRValueReferenceT&lt;&gt;</td>
<td>452</td>
</tr>
<tr>
<td>is_same</td>
<td>726</td>
</tr>
<tr>
<td>implementation</td>
<td>410</td>
</tr>
<tr>
<td>IsSameT</td>
<td>410</td>
</tr>
<tr>
<td>is_scalar</td>
<td>707</td>
</tr>
<tr>
<td>is_signed</td>
<td>709</td>
</tr>
<tr>
<td>is_standard_layout</td>
<td>712</td>
</tr>
<tr>
<td>is_swappable</td>
<td>725</td>
</tr>
<tr>
<td>is_swappable_with</td>
<td>724</td>
</tr>
<tr>
<td>is_trivial</td>
<td>712</td>
</tr>
<tr>
<td>is_trivially_assignable</td>
<td>722</td>
</tr>
<tr>
<td>is_trivially_constructible</td>
<td>719</td>
</tr>
<tr>
<td>is_trivially_copyable</td>
<td>712</td>
</tr>
<tr>
<td>is_trivially_copy_assignable</td>
<td>722</td>
</tr>
<tr>
<td>is_trivially_copy_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_trivially_default_constructible</td>
<td>720</td>
</tr>
<tr>
<td>is_trivially_destructible</td>
<td>724</td>
</tr>
<tr>
<td>is_trivially_move_assignable</td>
<td>723</td>
</tr>
<tr>
<td>is_trivially_move_constructible</td>
<td>721</td>
</tr>
<tr>
<td>is_union</td>
<td>706</td>
</tr>
<tr>
<td>is_unsigned</td>
<td>709</td>
</tr>
<tr>
<td>is_void</td>
<td>703</td>
</tr>
<tr>
<td>is_volatile</td>
<td>711</td>
</tr>
<tr>
<td>ItaniumABI</td>
<td>754</td>
</tr>
<tr>
<td>iterated instantiation</td>
<td>259</td>
</tr>
<tr>
<td>iterator</td>
<td>380, 765</td>
</tr>
<tr>
<td>iterator adapter</td>
<td>505</td>
</tr>
<tr>
<td>iterator_traits</td>
<td>399, 462</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>Järvi, Jaakko</td>
<td>214, 321, 488, 649</td>
</tr>
<tr>
<td>JosuttisLaunder</td>
<td>754</td>
</tr>
<tr>
<td>JosuttisStdLib</td>
<td>755</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>Karlsson, Bjorn</td>
<td>485</td>
</tr>
<tr>
<td>KarlssonSafeBool</td>
<td>755</td>
</tr>
<tr>
<td>Klarer, Rober</td>
<td>662</td>
</tr>
<tr>
<td>Koenig, Andrew</td>
<td>217, 218, 242</td>
</tr>
<tr>
<td>Koenig lookup</td>
<td>218, 242</td>
</tr>
<tr>
<td>KoenigMooAcc</td>
<td>755</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>lambda</td>
<td></td>
</tr>
<tr>
<td>as callable</td>
<td>160</td>
</tr>
<tr>
<td>as functor</td>
<td>160</td>
</tr>
<tr>
<td>closure</td>
<td>310</td>
</tr>
<tr>
<td>generic</td>
<td>80, 309, 618</td>
</tr>
<tr>
<td>primary type category</td>
<td>702</td>
</tr>
<tr>
<td>LambdaLib</td>
<td>755</td>
</tr>
<tr>
<td>launder()</td>
<td>617</td>
</tr>
<tr>
<td>layout</td>
<td></td>
</tr>
<tr>
<td>principles</td>
<td>490</td>
</tr>
<tr>
<td>lazy instantiation</td>
<td>245</td>
</tr>
<tr>
<td>left fold</td>
<td>208</td>
</tr>
<tr>
<td>levels of instantiation</td>
<td>542</td>
</tr>
<tr>
<td>lexing</td>
<td>224</td>
</tr>
<tr>
<td>LibIssue181</td>
<td>755</td>
</tr>
<tr>
<td>limit</td>
<td></td>
</tr>
<tr>
<td>levels of instantiation</td>
<td>542</td>
</tr>
<tr>
<td>linkable entity</td>
<td>154, 256, 765</td>
</tr>
<tr>
<td>linkage</td>
<td>182, 183</td>
</tr>
<tr>
<td>linkage</td>
<td>255</td>
</tr>
<tr>
<td>linker</td>
<td>765</td>
</tr>
<tr>
<td>linking</td>
<td>765</td>
</tr>
<tr>
<td>LippmanObjMod</td>
<td>755</td>
</tr>
<tr>
<td>LISP cons cells</td>
<td>571</td>
</tr>
<tr>
<td>list</td>
<td>380</td>
</tr>
<tr>
<td>literal operator</td>
<td>277</td>
</tr>
<tr>
<td>parsing</td>
<td>277, 599</td>
</tr>
<tr>
<td>literal-operator-id</td>
<td>216</td>
</tr>
<tr>
<td>literal type</td>
<td>391</td>
</tr>
<tr>
<td>trait</td>
<td>713</td>
</tr>
<tr>
<td>lookup</td>
<td>argument-dependent</td>
</tr>
<tr>
<td>for names</td>
<td>217</td>
</tr>
<tr>
<td>Koenig lookup</td>
<td>218, 242</td>
</tr>
<tr>
<td>ordinary</td>
<td>218, 249</td>
</tr>
<tr>
<td>qualified</td>
<td>216</td>
</tr>
<tr>
<td>two-phase</td>
<td>249</td>
</tr>
<tr>
<td>unqualified</td>
<td>216</td>
</tr>
<tr>
<td>loop</td>
<td>split</td>
</tr>
<tr>
<td>Lumsdaine, Andrew</td>
<td>488</td>
</tr>
<tr>
<td>lvalue</td>
<td>674, 765</td>
</tr>
<tr>
<td>before C++11</td>
<td>673</td>
</tr>
<tr>
<td>lvalue reference</td>
<td>105</td>
</tr>
</tbody>
</table>

**M**

| Maddock, John | 662 |
| make_pair() | 120 |
| make_signed | 729 |
| avoid undefined behavior | 442 |
| make_unsigned | 729 |
| avoid undefined behavior | 442 |
| manual instantiation | 260 |
| Marcus, Mat | 214 |
| match | 682 |
| best | 681 |
| perfect | 682 |
| materialization | 676 |
| Maurer, Jens | 214, 267, 321, 322 |
| max_align_t and type traits | 702 |
| max_element() | 380 |
| maximum | |
| levels of instantiation | 542 |
| munch | 226 |
| member | alias template | 178 |
| as base class | 492 |
| class template | 74, 178, 765 |
| function | see member function |
| function template | 74, 178 |
| initialization | 69 |
| of current instantiation | 240 |
| of unknown specialization | 229, 240 |
| template | see member template |
| type check | 431 |
| member function | 181 |
| as template | 74, 178 |
| implementation | 26 |
| template | 151, 765 |
| virtual | 182 |
| member function template specializations | 78 |
| member template | 74, 178, 765 |
| full specialization | 344 |
| generic lambda | 80 |
| versus template template parameter | 398 |
| virtual | 182 |
| Merrill, Jason | 214, 321 |
| metaprogramming | 123, 529, 549 |
| chrono library | 534 |
| constexpr | 529 |
| dimensions | 537 |
| for unit types | 534 |
| hybrid | 532 |
| on types | 531 |
| on values | 529 |
| unrolling loops | 533 |
| Metaware | 241, 545 |
| Meyers, Scott | 516 |
| MeyersCounting | 755 |
| MeyersEffective | 755 |
| MeyersMoreEffective | 755 |
| mixin | 203, 508 |
Index N

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>curious</td>
<td>510</td>
</tr>
<tr>
<td>modules</td>
<td>366</td>
</tr>
<tr>
<td>MoonFlavors</td>
<td>756</td>
</tr>
<tr>
<td>motivation of templates</td>
<td>1</td>
</tr>
<tr>
<td>move constructor</td>
<td></td>
</tr>
<tr>
<td>as template</td>
<td>79</td>
</tr>
<tr>
<td>detect noexcept</td>
<td>443</td>
</tr>
<tr>
<td>move semantics</td>
<td></td>
</tr>
<tr>
<td>perfect forwarding</td>
<td>91</td>
</tr>
<tr>
<td>MTL</td>
<td>756</td>
</tr>
<tr>
<td>MusserWangDynaVeri</td>
<td>756</td>
</tr>
<tr>
<td>Myers, Nathan</td>
<td>397, 462, 515</td>
</tr>
<tr>
<td>MyersTraits</td>
<td>756</td>
</tr>
<tr>
<td>Nackman, Lee R.</td>
<td>497, 515, 547</td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>class name injection</td>
<td>221</td>
</tr>
<tr>
<td>dependent</td>
<td>215, 217</td>
</tr>
<tr>
<td>dependent of templates</td>
<td>230</td>
</tr>
<tr>
<td>dependent of types</td>
<td>228</td>
</tr>
<tr>
<td>friend name injection</td>
<td>221, 241, 498</td>
</tr>
<tr>
<td>lookup</td>
<td>215, 217</td>
</tr>
<tr>
<td>nondependent</td>
<td>217</td>
</tr>
<tr>
<td>qualified</td>
<td>215, 216</td>
</tr>
<tr>
<td>two-phase lookup</td>
<td>249</td>
</tr>
<tr>
<td>unqualified</td>
<td>216</td>
</tr>
<tr>
<td>name()</td>
<td></td>
</tr>
<tr>
<td>of std::type_info</td>
<td>138</td>
</tr>
<tr>
<td>named template argument</td>
<td>358, 512</td>
</tr>
<tr>
<td>namespace</td>
<td></td>
</tr>
<tr>
<td>associated</td>
<td>219</td>
</tr>
<tr>
<td>scope</td>
<td>177</td>
</tr>
<tr>
<td>template</td>
<td>231</td>
</tr>
<tr>
<td>unnamed</td>
<td>666</td>
</tr>
<tr>
<td>narrowing nontype argument for templates</td>
<td>194</td>
</tr>
<tr>
<td>Naumann, Axel</td>
<td>547</td>
</tr>
<tr>
<td>negation</td>
<td>736</td>
</tr>
<tr>
<td>nested class</td>
<td>181</td>
</tr>
<tr>
<td>as template</td>
<td>74, 178</td>
</tr>
<tr>
<td>NewMat</td>
<td>756</td>
</tr>
<tr>
<td>NewShorterOED</td>
<td>756</td>
</tr>
<tr>
<td>Niebler, Eric</td>
<td>649</td>
</tr>
<tr>
<td>NIHCL</td>
<td>383</td>
</tr>
<tr>
<td>noexcept</td>
<td>290, 415</td>
</tr>
<tr>
<td>in declval</td>
<td>415</td>
</tr>
<tr>
<td>traits</td>
<td>443</td>
</tr>
<tr>
<td>nondeduced parameter</td>
<td>10</td>
</tr>
<tr>
<td>nondependent base class</td>
<td>236</td>
</tr>
<tr>
<td>name</td>
<td>217, 765</td>
</tr>
<tr>
<td>noninvasive</td>
<td>376</td>
</tr>
<tr>
<td>nonintrusive</td>
<td>376</td>
</tr>
<tr>
<td>nonreference</td>
<td></td>
</tr>
<tr>
<td>versus reference</td>
<td>115, 270, 638, 687</td>
</tr>
<tr>
<td>nontemplate</td>
<td></td>
</tr>
<tr>
<td>overloading with template</td>
<td>332</td>
</tr>
<tr>
<td>nontype argument</td>
<td></td>
</tr>
<tr>
<td>for templates</td>
<td>194</td>
</tr>
<tr>
<td>nontype parameter</td>
<td>45, 186</td>
</tr>
<tr>
<td>restrictions</td>
<td>49</td>
</tr>
<tr>
<td>nullptr type category</td>
<td>704</td>
</tr>
<tr>
<td>numeric</td>
<td></td>
</tr>
<tr>
<td>trait</td>
<td>707</td>
</tr>
<tr>
<td>numeric_limits</td>
<td>462</td>
</tr>
<tr>
<td>ODR</td>
<td>154, 663, 765</td>
</tr>
<tr>
<td>on-demand instantiation</td>
<td>243</td>
</tr>
<tr>
<td>one-definition rule</td>
<td>154, 663, 765</td>
</tr>
<tr>
<td>operator[ ]</td>
<td></td>
</tr>
<tr>
<td>at compile time</td>
<td>599</td>
</tr>
<tr>
<td>operator&gt;</td>
<td></td>
</tr>
<tr>
<td>in template argument list</td>
<td>50</td>
</tr>
<tr>
<td>operator&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>parsing</td>
<td>277, 599</td>
</tr>
<tr>
<td>operator-function-id</td>
<td>216</td>
</tr>
<tr>
<td>optimization</td>
<td></td>
</tr>
<tr>
<td>for copying strings</td>
<td>107</td>
</tr>
<tr>
<td>for empty base class</td>
<td>489</td>
</tr>
<tr>
<td>oracle</td>
<td>662</td>
</tr>
<tr>
<td>ordering</td>
<td></td>
</tr>
<tr>
<td>partial</td>
<td>330</td>
</tr>
</tbody>
</table>
### P

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rules</td>
<td>331</td>
</tr>
<tr>
<td>order of header files</td>
<td>141</td>
</tr>
<tr>
<td>ordinary lookup</td>
<td>218, 249</td>
</tr>
<tr>
<td>overloading</td>
<td>15, 323, 681</td>
</tr>
<tr>
<td>class templates</td>
<td>359</td>
</tr>
<tr>
<td>for string literals</td>
<td>71</td>
</tr>
<tr>
<td>initializer lists</td>
<td>691</td>
</tr>
<tr>
<td>nonreference versus reference</td>
<td>687</td>
</tr>
<tr>
<td>of function templates</td>
<td>326</td>
</tr>
<tr>
<td>partial ordering</td>
<td>330</td>
</tr>
<tr>
<td>reference versus nonreference</td>
<td>687</td>
</tr>
<tr>
<td>templates and nontemplates</td>
<td>332</td>
</tr>
<tr>
<td>OverloadingProperties</td>
<td>754</td>
</tr>
<tr>
<td>overload resolution</td>
<td>681, 766</td>
</tr>
<tr>
<td>for variadic templates</td>
<td>335</td>
</tr>
<tr>
<td>shall not participate</td>
<td>131</td>
</tr>
<tr>
<td>PCH</td>
<td>141</td>
</tr>
<tr>
<td>Pennello, Tom</td>
<td>241</td>
</tr>
<tr>
<td>perfect forwarding</td>
<td>91, 280</td>
</tr>
<tr>
<td>of return values</td>
<td>300</td>
</tr>
<tr>
<td>temporary</td>
<td>167</td>
</tr>
<tr>
<td>perfect match</td>
<td>682, 686</td>
</tr>
<tr>
<td>perfect returning</td>
<td>162</td>
</tr>
<tr>
<td>placeholder class type</td>
<td>314</td>
</tr>
<tr>
<td>placeholder type</td>
<td>422</td>
</tr>
<tr>
<td>void</td>
<td>6</td>
</tr>
<tr>
<td>void*</td>
<td>186</td>
</tr>
<tr>
<td>parameterization clause</td>
<td>177</td>
</tr>
<tr>
<td>parameterized class</td>
<td>766</td>
</tr>
<tr>
<td>parameterized function</td>
<td>669, 766</td>
</tr>
<tr>
<td>parameterized traits</td>
<td>394</td>
</tr>
<tr>
<td>parameter pack</td>
<td>56, 204, 454, 549</td>
</tr>
<tr>
<td>and enable_if</td>
<td>735</td>
</tr>
<tr>
<td>deduction</td>
<td>275</td>
</tr>
<tr>
<td>expansion</td>
<td>202</td>
</tr>
<tr>
<td>fold expression</td>
<td>207</td>
</tr>
<tr>
<td>function</td>
<td>204</td>
</tr>
<tr>
<td>slicing</td>
<td>365</td>
</tr>
<tr>
<td>template</td>
<td>188, 200</td>
</tr>
<tr>
<td>versus C-style varargs</td>
<td>409</td>
</tr>
<tr>
<td>with deduced type</td>
<td>298, 569</td>
</tr>
<tr>
<td>parsing</td>
<td>224</td>
</tr>
<tr>
<td>maximum munch</td>
<td>226</td>
</tr>
<tr>
<td>of angle brackets</td>
<td>225</td>
</tr>
<tr>
<td>partial ordering</td>
<td>330</td>
</tr>
<tr>
<td>of overloading</td>
<td>330</td>
</tr>
<tr>
<td>partial specialization</td>
<td>33, 152, 638, 766</td>
</tr>
<tr>
<td>additional parameters</td>
<td>453</td>
</tr>
<tr>
<td>for code selection</td>
<td>127</td>
</tr>
<tr>
<td>for function templates</td>
<td>356</td>
</tr>
<tr>
<td>of class templates</td>
<td>347</td>
</tr>
<tr>
<td>participate in overload resolution</td>
<td>131</td>
</tr>
<tr>
<td>pass-by-reference</td>
<td>20, 108</td>
</tr>
<tr>
<td>pass-by-value</td>
<td>20, 106</td>
</tr>
<tr>
<td>pattern</td>
<td>379</td>
</tr>
<tr>
<td>CRTP</td>
<td>495, 606</td>
</tr>
<tr>
<td>pack expansion</td>
<td>202</td>
</tr>
<tr>
<td>auto</td>
<td>50, 296</td>
</tr>
<tr>
<td>by value or by reference</td>
<td>20</td>
</tr>
<tr>
<td>const</td>
<td>186</td>
</tr>
<tr>
<td>constexpr expr</td>
<td>356</td>
</tr>
<tr>
<td>ellipsis</td>
<td>417, 683</td>
</tr>
<tr>
<td>for base class</td>
<td>70, 238</td>
</tr>
<tr>
<td>for call</td>
<td>9</td>
</tr>
<tr>
<td>formal</td>
<td>155</td>
</tr>
<tr>
<td>function</td>
<td>186</td>
</tr>
<tr>
<td>match</td>
<td>682</td>
</tr>
<tr>
<td>nontype</td>
<td>45, 186</td>
</tr>
<tr>
<td>of class templates</td>
<td>288</td>
</tr>
<tr>
<td>reference</td>
<td>167, 187</td>
</tr>
<tr>
<td>reference versus nonreference</td>
<td>115, 270, 638</td>
</tr>
<tr>
<td>string</td>
<td>54</td>
</tr>
<tr>
<td>template template parameter</td>
<td>83, 187</td>
</tr>
<tr>
<td>type</td>
<td>185</td>
</tr>
<tr>
<td>versus argument</td>
<td>155</td>
</tr>
<tr>
<td>decltype(auto)</td>
<td>301</td>
</tr>
</tbody>
</table>

Ensure Helvetica is embedded.
Placement new and launder() 617
POD 151, 766
Trait 713
POI 250, 668, 766
Pointer
conversion 689
conversion to void* 689
iterator traits 400
qualification 451
zero initialization 68
Pointer-to-member
qualification 454
Point of instantiation 250, 668, 766
Policy class 394, 395, 766
versus traits 397
Policy traits 458
Polymorphic object 667
Polymorphism 369, 767
bound 375
dynamic 369, 375
static 372, 376
unbounded 376
POOMA 756
pop_back() for vectors 26
pop_back() for vectors 23
Powell, Gary 214
Practice 137
Precompiled header 141, 767
Predicate traits 410
Prelinker 259
Preprocessor
Guard 667
Primary template 152, 184, 348, 767
Primary type (category) 702
Prime numbers 545
Promotion 683
Property
traits 458
pvalue 674, 767
ptdiff_t
versus size_t 685
ptdiff_t and type traits 702
push_back() for vectors 23
Q
qualification
of array types 453
of class types 456
of enumeration types 457
of function types 454
of fundamental types 448
of pointer-to-member types 454
of pointer types 451
of reference types 452
Qualified-id 216
Qualified lookup 216
Qualified name 215, 216, 767
Queried instantiation 257
Quora 749
R
Rank 715
Ratio library class 534
Read-only parameter types 458
Recursive instantiation 542
Ref() 112
Reference
and SFINAE 432
as template argument 270
as template parameter 167, 187
Binding 679
Collapsing 277
Forwarding 111
Lvalue 105
Qualification 452
Rvalue 105
versus nonreference 115, 270, 638, 687
Reference counting 767
Reference_wrapper 112
Reflection
check for data members 434
check for members functions 435
check for type members 431
Future 363
Metaprogramming 538
Remove_all_extents 731
EnsureHelvetica/sembdeded_()
remove_const 728
remove_cv 728
remove_extent 731
remove_pointer 730
remove_reference 118, 729
remove_volatile 728
requires 103, 740
  clause 476, 740
  expression 742
restricted template expansion 497
result_of 163, 717
result type traits 413
return perfectly 162
return type
  auto 11, 296
  decay 166
  decltype(auto) 301
  deduction 296
  trailing 282
return value
  by value vs. by reference 117
  perfect forwarding 300
right fold 208
run-time analysis oracles 662
rvalue 674, 767
  before C++11 673
rvalue reference 105
  const 110
  deduction 277
  perfect match 687
  value category 92

generic lambdas 421
  partial specialization 420
  reference types 432
SFINAE out 131
shall not participate in overload resolution 131
shallow instantiation 652
Siek, Jeremy 515, 662
signature 328
Silicon Graphics 462
sizeof... 57
sizeof 401
size_t
  versus ptrdiff_t 685
size_t type and type traits 702
small string optimization 107
Smalltalk 376, 383
Smaragdakis, Yannis 516
SmaragdakisBatoryMixins 756
Smith, Richard 214, 321
source file 767
spaces 770
specialization 31, 152, 243, 323, 768
  algorithm 557
  explicit 152, 224, 228, 238, 338
  full 338
  generated 152
  inline 78
  instantiated 152
  of algorithms 465
  of member function template 78
  partial 152, 347, 638
  partial for function templates 356
  unknown 223
special member function 79
  disable 102
  templify 102
Spertus, Mike 321
Spicer, John 321, 352
SpicerSFINAE 756
spilled inlined function 257
split loop 634
SSO 107
Stack<> 23

S

Sankel, David 547
scanning 224
semantic transparency 325
separation model 266
sequence
  of conversions 689
SFINAE 129, 284, 767
  examples 416
  friendly 424
  function overloading 416
Index T

Stackoverflow 749
Standard C++ Foundation 750
standard-layout type 712
standard library
utilities 697
Standard Template Library 241, 380, see
STL
static_assert 654
as concept 29
in templates 6
static constant
versus enumeration 543
static data member template 768
static if 266
static member 28, 181
static polymorphism 372, 376
std. hpp 142
StepanovLeeSTL 757
StepanovNotes 757
STL 241, 380, 649
string
[] 685
and reference template parameters 271
as parameter 54
as template argument 49, 354
literal as template parameters 271
string literal
as parameter in templates 71
parsing 277
passing 115
value category 674
Stroustrup, Bjarne 214, 321
StroustrupC++PL 757
StroustrupDnE 757
StroustrupGlossary 757
struct
definition 668
qualification 456
versus class 151
structured bindings 306
substitution 152, 768
Sun Microsystems 257
surrogate 694
Sutter, Herb 266, 321, 547
SutterExceptional 757
SutterMoreExceptional 757
Sutton, Andrew 214, 547
syntax checking 6

tag dispatching 467, 487
class templates 479
Taligent 240
taxonomy of names 215
template 768
alias 39, 312, 446
argument 155, see template argument
default argument 190
for namespaces 231
friend 213
id see template-id
inline 20, 140
instantiation 152, 243, see
instantiation
instantiation-safe 482
member template 74, 178
metaprogramming 529, 549
motivation 1
name 155, 215
nontype arguments 194
of template 28
overloading with nontemplate 332
parameter 3, 9, 155, 185
primary 152, 184, 348
specialization 31, 152
substitution 152
type arguments 194
typedef 38
union 180
variable 80, 447
variadic 55, 190, 200
.template 79, 231
->template 80, 231
::template 231
template argument 155, 192, 768
array 453
<table>
<thead>
<tr>
<th>Index U</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>closure type</td>
<td>310</td>
</tr>
<tr>
<td>complete</td>
<td>154, 245</td>
</tr>
<tr>
<td>composite type (category)</td>
<td>702</td>
</tr>
<tr>
<td>conversion</td>
<td>74</td>
</tr>
<tr>
<td>definition</td>
<td>xxxii, 38, see type alias</td>
</tr>
<tr>
<td>dependent</td>
<td>223, 228</td>
</tr>
<tr>
<td>dependent name</td>
<td>228</td>
</tr>
<tr>
<td>erasure</td>
<td>523</td>
</tr>
<tr>
<td>for bool_constant</td>
<td>699</td>
</tr>
<tr>
<td>for integral_constant</td>
<td>698</td>
</tr>
<tr>
<td>function</td>
<td>401</td>
</tr>
<tr>
<td>incomplete</td>
<td>154</td>
</tr>
<tr>
<td>metaprogramming</td>
<td>531</td>
</tr>
<tr>
<td>of *this</td>
<td>686</td>
</tr>
<tr>
<td>of container element</td>
<td>401</td>
</tr>
<tr>
<td>parameter</td>
<td>185</td>
</tr>
<tr>
<td>POD</td>
<td>151</td>
</tr>
<tr>
<td>POD trait</td>
<td>713</td>
</tr>
<tr>
<td>predicates</td>
<td>410</td>
</tr>
<tr>
<td>primary type (category)</td>
<td>702</td>
</tr>
<tr>
<td>qualification</td>
<td>448, 460</td>
</tr>
<tr>
<td>read-only parameter</td>
<td>458</td>
</tr>
<tr>
<td>requirement</td>
<td>743</td>
</tr>
<tr>
<td>safety</td>
<td>376</td>
</tr>
<tr>
<td>standard-layout</td>
<td>712</td>
</tr>
<tr>
<td>trivial</td>
<td>712</td>
</tr>
<tr>
<td>utilities in standard library</td>
<td>697</td>
</tr>
<tr>
<td>type alias</td>
<td>769</td>
</tr>
<tr>
<td>type category</td>
<td></td>
</tr>
<tr>
<td>composite</td>
<td>702</td>
</tr>
<tr>
<td>primary</td>
<td>702</td>
</tr>
<tr>
<td>typedef</td>
<td>38</td>
</tr>
<tr>
<td>type-dependent expression</td>
<td>217, 233</td>
</tr>
<tr>
<td>typeid</td>
<td>138</td>
</tr>
<tr>
<td>type_info</td>
<td>138</td>
</tr>
<tr>
<td>typelist</td>
<td>455, 549</td>
</tr>
<tr>
<td>typename</td>
<td>4, 67, 185, 229</td>
</tr>
<tr>
<td>future</td>
<td>354</td>
</tr>
<tr>
<td>type parameter</td>
<td>4</td>
</tr>
<tr>
<td>TypeT&lt;&gt;</td>
<td>460</td>
</tr>
<tr>
<td>type template</td>
<td>769</td>
</tr>
<tr>
<td>type traits</td>
<td>164, see traits as predicates 410</td>
</tr>
<tr>
<td>for incomplete types</td>
<td>734</td>
</tr>
<tr>
<td>standard library</td>
<td>697</td>
</tr>
<tr>
<td>_t version</td>
<td>40, 83</td>
</tr>
<tr>
<td>unexpected behavior</td>
<td>164</td>
</tr>
<tr>
<td>variadic templates, multiple type traits</td>
<td>734</td>
</tr>
<tr>
<td>__type_traits</td>
<td>462</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>UCN</td>
<td>216</td>
</tr>
<tr>
<td>unary fold</td>
<td>208</td>
</tr>
<tr>
<td>unbounded polymorphism</td>
<td>376</td>
</tr>
<tr>
<td>underlying_type</td>
<td>716</td>
</tr>
<tr>
<td>unevaluated operand</td>
<td>133, 667</td>
</tr>
<tr>
<td>union</td>
<td></td>
</tr>
<tr>
<td>anonymous</td>
<td>246</td>
</tr>
<tr>
<td>definition</td>
<td>668</td>
</tr>
<tr>
<td>discriminated</td>
<td>603</td>
</tr>
<tr>
<td>qualification</td>
<td>456</td>
</tr>
<tr>
<td>template</td>
<td>180</td>
</tr>
<tr>
<td>unit types metaprogramming</td>
<td>534</td>
</tr>
<tr>
<td>universal character name</td>
<td>216</td>
</tr>
<tr>
<td>universal reference</td>
<td>93, 111, 769, see forwarding reference</td>
</tr>
<tr>
<td>unknown specialization</td>
<td>223, 228</td>
</tr>
<tr>
<td>member of</td>
<td>229, 240</td>
</tr>
<tr>
<td>unnamed namespace</td>
<td>666</td>
</tr>
<tr>
<td>unqualified-id</td>
<td>216</td>
</tr>
<tr>
<td>unqualified lookup</td>
<td>216</td>
</tr>
<tr>
<td>unqualified name</td>
<td>216</td>
</tr>
<tr>
<td>unrolling loops</td>
<td>533</td>
</tr>
<tr>
<td>Unruh, Erwin</td>
<td>352, 545</td>
</tr>
<tr>
<td>UnruhPrimeOrig</td>
<td>757</td>
</tr>
<tr>
<td>user-defined conversion</td>
<td>195, 769</td>
</tr>
<tr>
<td>using</td>
<td>4, 231</td>
</tr>
<tr>
<td>using declaration</td>
<td>239</td>
</tr>
<tr>
<td>dependent name</td>
<td>231</td>
</tr>
<tr>
<td>variadic expressions</td>
<td>65</td>
</tr>
<tr>
<td>utilities in the standard library</td>
<td>697</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>valarray</td>
<td>648</td>
</tr>
</tbody>
</table>
88

Vali, Faisal 321
value
as parameter 45
for bool_constant 699
for integral_constant 698
functions 401
value category 282, 673, 770
*this 686
before C++11 673
declaytype 678
of string literals 674
of template parameters 187
since C++11 674
value-dependent expression 234
value initialization 68
value metaprogramming 529
value traits 389
value_type
for bool_constant 699
for integral_constant 698
Vandevoorde, David 242, 321, 516, 547, 647
VandevoordeJosuttisTemplates1st 757
VandevoordeSolutions 758
varargs interface 55
variable template constexpr 473
variadic
base classes 65
using 65
variadic template 55, 190, 200
and enable_if 735
deduction guide 64
fold expression 58
multiple type traits 734
overload resolution 335
perfect forwarding 281
variant 603
vector 23
vector 380
Veldhuizen, Todd 546, 647
VeldhuizenMeta95 758
virtual
function dispatch table 257
instantiation 246
member templates 182
parameterized 510
visitor for Variant 617
void
and decltype (auto) 162
as template parameter 6
in templates 361
void*
conversion to 689
template parameter 186
void_t 420, 437
Voutilainen, Ville 267
W
whitespace 770
Willcock, Jeremiah 488
Witt, Thomas 515
wrap function calls 162
X
xvalue 674, 770
Z
zero initialization 68