C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter

Andrei Alexandrescu

Addison-Wesley

Boston

Contents

Preface xi

Organizational and Policy Issues 1

0. Don't sweat the small stuff. (Or: Know what not to standardize.) 2
1. Compile cleanly at high warning levels. 4
2. Use an automated build system. 7
3. Use a version control system. 8
4. Invest in code reviews. 9

Design Style 11

5. Give one entity one cohesive responsibility. 12
6. Correctness, simplicity, and clarity come first. 13
7. Know when and how to code for scalability. 14
8. Don't optimize prematurely. 16
9. Don't pessimize prematurely. 18
10. Minimize global and shared data. 19
11. Hide information. 20
12. Know when and how to code for concurrency. 21
13. Ensure resources are owned by objects. Use explicit RAII and smart pointers. 24

Coding Style 27

14. Prefer compile- and link-time errors to run-time errors. 28
15. Use **const** proactively. 30
16. Avoid macros. 32

17. Avoid magic numbers. 34
18. Declare variables as locally as possible. 35
19. Always initialize variables. 36
20. Avoid long functions. Avoid deep nesting. 38
21. Avoid initialization dependencies across compilation units. 39
22. Minimize definitional dependencies. Avoid cyclic dependencies. 40
23. Make header files self-sufficient. 42
24. Always write internal #include guards. Never write external #include guards. 43

Functions and Operators 45

25. Take parameters appropriately by value, (smart) pointer, or reference. 46
26. Preserve natural semantics for overloaded operators. 47
27. Prefer the canonical forms of arithmetic and assignment operators. 48
28. Prefer the canonical form of ++ and --. Prefer calling the prefix forms. 50
29. Consider overloading to avoid implicit type conversions. 51
30. Avoid overloading &&, ||, or , (comma) . 52
31. Don't write code that depends on the order of evaluation of function arguments. 54

Class Design and Inheritance 55

32. Be clear what kind of class you're writing. 56
33. Prefer minimal classes to monolithic classes. 57
34. Prefer composition to inheritance. 58
35. Avoid inheriting from classes that were not designed to be base classes. 60
36. Prefer providing abstract interfaces. 62
37. Public inheritance is substitutability. Inherit, not to reuse, but to be reused. 64
38. Practice safe overriding. 66
39. Consider making virtual functions nonpublic, and public functions nonvirtual. 68
40. Avoid providing implicit conversions. 70
41. Make data members private, except in behaviorless aggregates (C-style structs). 72
42. Don't give away your internals. 74
43. Pimpl judiciously. 76
44. Prefer writing nonmember nonfriend functions. 79
45. Always provide new and delete together. 80
46. If you provide any class-specific new, provide all of the standard forms (plain, in-place, and nothrow). 82

Construction, Destruction, and Copying **85**
47. Define and initialize member variables in the same order. 86
48. Prefer initialization to assignment in constructors. 87
49. Avoid calling virtual functions in constructors and destructors. 88
50. Make base class destructors public and virtual, or protected and nonvirtual. 90
51. Destructors, deallocation, and swap never fail. 92
52. Copy and destroy consistently. 94
53. Explicitly enable or disable copying. 95
54. Avoid slicing. Consider Clone instead of copying in base classes. 96
55. Prefer the canonical form of assignment. 99
56. Whenever it makes sense, provide a no-fail swap (and provide it correctly). 100

Namespaces and Modules **103**
57. Keep a type and its nonmember function interface in the same namespace. 104
58. Keep types and functions in separate namespaces unless they're specifically intended to work together. 106
59. Don't write namespace usings in a header file or before an #include. 108
60. Avoid allocating and deallocating memory in different modules. 111
61. Don't define entities with linkage in a header file. 112
62. Don't allow exceptions to propagate across module boundaries. 114
63. Use sufficiently portable types in a module's interface. 116

Templates and Genericity **119**
64. Blend static and dynamic polymorphism judiciously. 120
65. Customize intentionally and explicitly. 122
66. Don't specialize function templates. 126
67. Don't write unintentionally nongeneric code. 128

Error Handling and Exceptions **129**
68. Assert liberally to document internal assumptions and invariants. 130
69. Establish a rational error handling policy, and follow it strictly. 132
70. Distinguish between errors and non-errors. 134
71. Design and write error-safe code. 137
72. Prefer to use exceptions to report errors. 140
73. Throw by value, catch by reference. 144
74. Report, handle, and translate errors appropriately. 145
75. Avoid exception specifications. 146

STL: Containers 149

76. Use vector by default. Otherwise, choose an appropriate container. 150
77. Use vector and string instead of arrays. 152
78. Use vector (and string::c_str) to exchange data with non-C++ APIs. 153
79. Store only values and smart pointers in containers. 154
80. Prefer push_back to other ways of expanding a sequence. 155
81. Prefer range operations to single-element operations. 156
82. Use the accepted idioms to really shrink capacity and really erase elements. 157

STL: Algorithms 159

83. Use a checked STL implementation. 160
84. Prefer algorithm calls to handwritten loops. 162
85. Use the right STL search algorithm. 165
86. Use the right STL sort algorithm. 166
87. Make predicates pure functions. 168
88. Prefer function objects over functions as algorithm and comparer arguments. 170
89. Write function objects correctly. 172

Type Safety 173

90. Avoid type switching; prefer polymorphism. 174
91. Rely on types, not on representations. 176
92. Avoid using reinterpret_cast. 177
93. Avoid using static_cast on pointers. 178
94. Avoid casting away const. 179
95. Don't use C-style casts. 180
96. Don't memcpy or memcmp non-PODs. 182
97. Don't use unions to reinterpret representation. 183
98. Don't use varargs (ellipsis). 184
99. Don't use invalid objects. Don't use unsafe functions. 185
100. Don't treat arrays polymorphically. 186

Bibliography 187

Summary of Summaries 195

Index 209