C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter
Andrei Alexandrescu

C++ In-Depth Series • Bjarne Stroustrup

C++ Coding Standards

C++ Coding Standards: 101 Rules, Guidelines, and Best Practices

Table of Contents

Contents

Preface

Organizational and Policy Issues

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

Design Style

5. Give one entity one cohesive responsibility.
6. Correctness, simplicity, and clarity come first.
7. Know when and how to code for scalability.
8. Dont optimize prematurely.
9. Dont pessimize prematurely.
10. Minimize global and shared data.
11. Hide information.
12. Know when and how to code for concurrency.
13. Ensure resources are owned by objects. Use explicit RAII and smart pointers.

Coding Style

14. Prefer compile- and link-time errors to run-time errors.

Pearson

Table of Contents

15. Use const proactively.
16. Avoid macros.
17. Avoid magic numbers.
18. Declare variables as locally as possible.
19. Always initialize variables.
20. Avoid long functions. Avoid deep nesting.
21. Avoid initialization dependencies across compilation units.
22. Minimize definitional dependencies. Avoid cyclic dependencies.
23. Make header files self-sufficient.
24. Always write internal #include guards. Never write external #include guards.

Functions and Operators

25. Take parameters appropriately by value, (smart) pointer, or reference.
26. Preserve natural semantics for overloaded operators.
27. Prefer the canonical forms of arithmetic and assignment operators.
28. Prefer the canonical form of ++ and --. Prefer calling the prefix forms.
29. Consider overloading to avoid implicit type conversions.
30. Avoid overloading &&, ||, or , (comma)
31. Dont write code that depends on the order of evaluation of function arguments.

Class Design and Inheritance

32. Be clear what kind of class youre writing.
33. Prefer minimal classes to monolithic classes.
34. Prefer composition to inheritance.

Table of Contents

35. Avoid inheriting from classes that were not designed to be base classes.
36. Prefer providing abstract interfaces.
37. Public inheritance is substitutability. Inherit, not to reuse, but to be reused.
38. Practice safe overriding.
39. Consider making virtual functions nonpublic, and public functions nonvirtual.
40. Avoid providing implicit conversions.
41. Make data members private, except in behaviorless aggregates (C-style structs).
42. Dont give away your internals.
43. Pimpl judiciously.
44. Prefer writing nonmember nonfriend functions.
45. Always provide new and delete together.
46. If you provide any class-specific new, provide all of the standard forms (plain, in-place, and nothrow).

Construction, Destruction, and Copying

47. Define and initialize member variables in the same order.
48. Prefer initialization to assignment in constructors.
49. Avoid calling virtual functions in constructors and destructors.
50. Make base class destructors public and virtual, or protected and nonvirtual.
51. Destructors, deallocation, and swap never fail.
52. Copy and destroy consistently.
53. Explicitly enable or disable copying.
54. Avoid slicing. Consider Clone instead of copying in base classes.

Pearson

Table of Contents

55. Prefer the canonical form of assignment.
56. Whenever it makes sense, provide a no-fail swap (and provide it correctly).

Namespaces and Modules

57. Keep a type and its nonmember function interface in the same namespace.
58. Keep types and functions in separate namespaces unless theyre specifically intended to work together.
59. Dont write namespace usings in a header file or before an #include.
60. Avoid allocating and deallocating memory in different modules.
61. Dont define entities with linkage in a header file.
62. Dont allow exceptions to propagate across module boundaries.
63. Use sufficiently portable types in a modules interface.

Templates and Genericity

64. Blend static and dynamic polymorphism judiciously.
65. Customize intentionally and explicitly.
66. Dont specialize function templates.
67. Dont write unintentionally nongeneric code.

Error Handling and Exceptions

68. Assert liberally to document internal assumptions and invariants.
69. Establish a rational error handling policy, and follow it strictly.
70. Distinguish between errors and non-errors.
71. Design and write error-safe code.
72. Prefer to use exceptions to report errors.
73. Throw by value, catch by reference.

Table of Contents

74. Report, handle, and translate errors appropriately.
75. Avoid exception specifications.

STL: Containers

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

STL: Algorithms

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

Type Safety

90. Avoid type switching; prefer polymorphism.
91. Rely on types, not on representations.
92. Avoid using reinterpret_cast.
93. Avoid using static_cast on pointers.

Table of Contents

94. Avoid casting away const.
95. Dont use C-style casts.
96. Dont memcpy or memcmp non-PODs.
97. Dont use unions to reinterpret representation.
98. Dont use varargs (ellipsis).
99. Dont use invalid objects. Dont use unsafe functions.
100.Dont treat arrays polymorphically.

Bibliography

Summary of Summaries

Index

Pearson