


Google C++ Style Guide

Revision 3.180

*Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray*

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: . You may toggle all summaries with the big arrow button:



Toggle all summaries


Table of Contents

Header Files	The #define Guard Header File Dependencies Inline Functions The -inl.h Files Function Parameter Ordering Names and Order of Includes
Scoping	Namespaces Nested Classes Nonmember, Static Member, and Global Functions Local Variables Static and Global Variables
Classes	Doing Work in Constructors Default Constructors Explicit Constructors Copy Constructors Structs vs. Classes Inheritance Multiple Inheritance Interfaces Operator Overloading Access Control Declaration Order Write Short Functions
Google-Specific Magic	Smart Pointers cpplint
Other C++ Features	Reference Arguments Function Overloading Default Arguments Variable-Length Arrays and alloca() Friends Exceptions Run-Time Type Information (RTTI) Casting Streams Preincrement and Predecrement Use of const Integer Types 64-bit Portability Preprocessor Macros 0 and NULL sizeof Boost C++0x
Naming	General Naming Rules File Names Type Names Variable Names Constant Names Function Names Namespace Names Enumerator Names Macro Names Exceptions to Naming Rules
Comments	Comment Style File Comments Class Comments Function Comments Variable Comments Implementation Comments

	Punctuation, Spelling and Grammar TODO Comments Deprecation Comments
Formatting	Line Length Non-ASCII Characters Spaces vs. Tabs Function Declarations and Definitions Function Calls Conditionals Loops and Switch Statements Pointer and Reference Expressions Boolean Expressions Return Values Variable and Array Initialization Preprocessor Directives Class Format Constructor Initializer Lists Namespace Formatting Horizontal Whitespace Vertical Whitespace
Exceptions to the Rules	Existing Non-conformant Code Windows Code

Important Note

Displaying Hidden Details in this Guide

 This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Background

C++ is the main development language used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term *Style* is a bit of a misnomer, since these conventions cover far more than just source file formatting.

One way in which we keep the code base manageable is by enforcing *consistency*. It is very important that any programmer be able to look at another's code and quickly understand it. Maintaining a uniform style and following conventions means that we can more easily use "pattern-matching" to infer what various symbols are and what invariants are true about them. Creating common, required idioms and patterns makes code much easier to understand. In some cases there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency.

Another issue this guide addresses is that of C++ feature bloat. C++ is a huge language with many advanced features. In some cases we constrain, or even ban, use of certain features. We do this to keep code simple and to avoid the various common errors and problems that these features can cause. This guide lists these features and explains why their use is restricted.

Open-source projects developed by Google conform to the requirements in this guide.

Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language.

Header Files

In general, every `.cc` file should have an associated `.h` file. There are some common exceptions, such as unittests and small `.cc` files containing just a `main()` function.

Correct use of header files can make a huge difference to the readability, size and performance of your code.

The following rules will guide you through the various pitfalls of using header files.

The `#define` Guard

- ▶ All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `<PROJECT>_<PATH>_<FILE>_H_`.

Header File Dependencies

- ▶ Don't use an `#include` when a forward declaration would suffice.

Inline Functions

- ▶ Define functions inline only when they are small, say, 10 lines or less.

The `-inl.h` Files

- ▶ You may use file names with a `-inl.h` suffix to define complex inline functions when needed.

Function Parameter Ordering

- ▶ When defining a function, parameter order is: inputs, then outputs.

Names and Order of Includes

- ▶ Use standard order for readability and to avoid hidden dependencies: C library, C++ library, other libraries' `.h`, your project's `.h`.

Scoping

Namespaces

- ▶ Unnamed namespaces in `.cc` files are encouraged. With named namespaces, choose the name based on the project, and possibly its path. Do not use a *using-directive*.

Nested Classes

- ▶ Although you may use public nested classes when they are part of an interface, consider a [namespace](#) to keep declarations out of the global scope.

Nonmember, Static Member, and Global Functions

- ▶ Prefer nonmember functions within a namespace or static member functions to global functions; use completely global functions rarely.

Local Variables

- ▶ Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

Static and Global Variables

- ▶ Static or global variables of class type are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction.

Classes

Classes are the fundamental unit of code in C++. Naturally, we use them extensively. This section lists the main dos and don'ts you should follow when writing a class.

Doing Work in Constructors

- ▶ In general, constructors should merely set member variables to their initial values. Any complex initialization should go in an explicit `Init()` method.

Default Constructors

- ▶ You must define a default constructor if your class defines member variables and has no other constructors. Otherwise the compiler will do it for you, badly.

Explicit Constructors

- ▶ Use the C++ keyword `explicit` for constructors with one argument.

Copy Constructors

- ▶ Provide a copy constructor and assignment operator only when necessary. Otherwise, disable them with `DISALLOW_COPY_AND_ASSIGN`.

Structs vs. Classes

- ▶ Use a `struct` only for passive objects that carry data; everything else is a `class`.

Inheritance

- ▶ Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

Multiple Inheritance

- ▶ Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be [pure interface](#) classes tagged with the `Interface` suffix.

Interfaces

- ▶ Classes that satisfy certain conditions are allowed, but not required, to end with an `Interface` suffix.

Operator Overloading

- ▶ Do not overload operators except in rare, special circumstances.

Access Control

- ▶ Make data members `private`, and provide access to them through accessor functions as needed (for technical reasons, we allow data members of a test fixture class to be `protected` when using [Google Test](#)). Typically a variable would be called `foo_` and the accessor function `foo()`. You may also want a mutator function `set_foo()`. Exception: `static const` data members (typically called `kFoo`) need not be `private`.

Declaration Order

- ▶ Use the specified order of declarations within a class: `public:` before `private:`, methods before data members (variables), etc.

Write Short Functions

- ▶ Prefer small and focused functions.

Google-Specific Magic

There are various tricks and utilities that we use to make C++ code more robust, and various ways we use C++ that may differ from what you see elsewhere.

Smart Pointers

- ▶ If you actually need pointer semantics, `scoped_ptr` is great. You should only use `std::tr1::shared_ptr` under very specific conditions, such as when objects need to be held by STL containers. You should never use `auto_ptr`.

cpplint

- ▶ Use `cpplint.py` to detect style errors.

Other C++ Features

Reference Arguments

- ▶ All parameters passed by reference must be labeled `const`.

Function Overloading

- ▶ Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

Default Arguments

- ▶ We do not allow default function parameters, except in a few uncommon situations explained below.

Variable-Length Arrays and `alloca()`

- ▶ We do not allow variable-length arrays or `alloca()`.

Friends

- ▶ We allow use of `friend` classes and functions, within reason.

Exceptions

- ▶ We do not use C++ exceptions.

Run-Time Type Information (RTTI)

- ▶ We do not use Run Time Type Information (RTTI).

Casting

- ▶ Use C++ casts like `static_cast<>()`. Do not use other cast formats like `int y = (int)x;` or `int y = int(x);`.

Streams

- ▶ Use streams only for logging.

Preincrement and Predecrement

- ▶ Use prefix form (`++i`) of the increment and decrement operators with iterators and other template objects.

Use of `const`

- ▶ We strongly recommend that you use `const` whenever it makes sense to do so.

Integer Types

- ▶ Of the built-in C++ integer types, the only one used is `int`. If a program needs a variable of a different size, use a precise-width integer type from `<stdint.h>`, such as `int16_t`.

64-bit Portability

- ▶ Code should be 64-bit and 32-bit friendly. Bear in mind problems of printing, comparisons, and structure alignment.

Preprocessor Macros

- ▶ Be very cautious with macros. Prefer inline functions, enums, and `const` variables to macros.

0 and NULL

- ▶ Use `0` for integers, `0.0` for reals, `NULL` for pointers, and `'\0'` for chars.

sizeof

- ▶ Use `sizeof(varname)` instead of `sizeof(type)` whenever possible.

Boost

- ▶ Use only approved libraries from the Boost library collection.

C++0x

- ▶ Use only approved libraries and language extensions from C++0x. Currently, none are approved.

Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

- ▶ Function names, variable names, and filenames should be descriptive; eschew abbreviation. Types and variables should be nouns, while functions should be "command" verbs.

File Names

- ▶ Filenames should be all lowercase and can include underscores (`_`) or dashes (`-`). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer `"_"`.

Type Names

- ▶ Type names start with a capital letter and have a capital letter for each new word, with no underscores: `MyExcitingClass`, `MyExcitingEnum`.

Variable Names

- ▶ Variable names are all lowercase, with underscores between words. Class member variables have trailing underscores. For instance:
`my_exciting_local_variable`, `my_exciting_member_variable_`.

Constant Names

- ▶ Use a `k` followed by mixed case: `kDaysInAWeek`.

Function Names

- ▶ Regular functions have mixed case; accessors and mutators match the name of the variable: `MyExcitingFunction()`, `MyExcitingMethod()`,
`my_exciting_member_variable()`, `set_my_exciting_member_variable()`.

Namespace Names

- ▶ Namespace names are all lower-case, and based on project names and possibly their directory structure: `google_awesome_project`.

Enumerator Names

- ▶ Enumerators should be named *either* like [constants](#) or like [macros](#): either `kEnumName` or `ENUM_NAME`.

Macro Names

- ▶ You're not really going to [define a macro](#), are you? If you do, they're like this:
`MY_MACRO_THAT_SCARES_SMALL_CHILDREN`.

Exceptions to Naming Rules

- ▶ If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme.

Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

Comment Style

- ▶ Use either the `//` or `/* */` syntax, as long as you are consistent.

File Comments

- ▶ Start each file with a copyright notice, followed by a description of the contents of the file.

Class Comments

- ▶ Every class definition should have an accompanying comment that describes what it is for and how it should be used.

Function Comments

- ▶ Declaration comments describe use of the function; comments at the definition of a function describe operation.

Variable Comments

- ▶ In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

Implementation Comments

- ▶ In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

Punctuation, Spelling and Grammar

- ▶ Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

TODO Comments

- ▶ Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

Deprecation Comments

- ▶ Mark deprecated interface points with `DEPRECATED` comments.

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

To help you format code correctly, we've created a [settings file for emacs](#).

Line Length

- ▶ Each line of text in your code should be at most 80 characters long.

Non-ASCII Characters

- ▶ Non-ASCII characters should be rare, and must use UTF-8 formatting.

Spaces vs. Tabs

- ▶ Use only spaces, and indent 2 spaces at a time.

Function Declarations and Definitions

- ▶ Return type on the same line as function name, parameters on the same line if they fit.

Function Calls

- ▶ On one line if it fits; otherwise, wrap arguments at the parenthesis.

Conditionals

- ▶ Prefer no spaces inside parentheses. The `else` keyword belongs on a new line.

Loops and Switch Statements

- ▶ Switch statements may use braces for blocks. Empty loop bodies should use `{}` or `continue`.

Pointer and Reference Expressions

- ▶ No spaces around period or arrow. Pointer operators do not have trailing spaces.

Boolean Expressions

- ▶ When you have a boolean expression that is longer than the [standard line length](#), be consistent in how you break up the lines.

Return Values

- ▶ Do not needlessly surround the `return` expression with parentheses.

Variable and Array Initialization

- ▶ Your choice of `=` or `()`.

Preprocessor Directives

- ▶ Preprocessor directives should not be indented but should instead start at the beginning of the line.

Class Format

- ▶ Sections in `public`, `protected` and `private` order, each indented one space.

Constructor Initializer Lists

- ▶ Constructor initializer lists can be all on one line or with subsequent lines indented four spaces.

Namespace Formatting

- ▶ The contents of namespaces are not indented.

Horizontal Whitespace

- ▶ Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.

Vertical Whitespace

- ▶ Minimize use of vertical whitespace.

Exceptions to the Rules

The coding conventions described above are mandatory. However, like all good rules, these sometimes have exceptions, which we discuss here.

Existing Non-conformant Code

- ▶ You may diverge from the rules when dealing with code that does not conform to this style guide.

Windows Code

▶ Windows programmers have developed their own set of coding conventions, mainly derived from the conventions in Windows headers and other Microsoft code. We want to make it easy for anyone to understand your code, so we have a single set of guidelines for everyone writing C++ on any platform.

Parting Words

Use common sense and *BE CONSISTENT*.

If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their `if` clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity throws readers out of their rhythm when they go to read it. Try to avoid this.

OK, enough writing about writing code; the code itself is much more interesting. Have fun!

Revision 3.180

*Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray*