A Quick Summary of Google C++ Style Guide

[Google C++ Style Guide [external link]](https://google.github.io/styleguide/cppguide.html#Naming)

1. Header Files

1. Every header file should have a guard to avoid multiple copies, except for .inc files.
2. The header include order should be "Related header, C library, C++ library, other libraries' .h, your project's .h.".
3. All templates and inline functions should have their definitions in the header files, not in implementations.
4. Only inline functions with less than 10 lines, but do not inline constructors and destructors.

2. Scoping and Namespace

1. Do not use "using namespace xxx" syntax in headers. For implementation, prefer not use "using namespace xxx" to make all names from a space available, use "using std::string" as an alternative if possible.
2. Prefer placing nonmember functions in a named space to avoid global space pollution. If the nonmember function is only used within its .cc file, declare it as static.
3. When definitions in a .cc file do not need to be referenced outside that file, place them in an unnamed namespace or declare them static.
4. Variables needed for if, while and for statements should normally be declared within those statements, so that such variables are confined to those scopes.
5. Try to define local variable with initialization.

3. Class Declaration and Definition

1. Constructors should never call virtual functions.
2. Terminating the program may be an appropriate error handling response, if an error occurs in the constructor.
3. Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition. As an exception, copy and move constructors should not be explicit.
4. Use a struct only for passive objects that carry data; everything else is a class.

4. Function Declaration and Definition

1. Prefer using return values rather than output parameters. If output-only parameters are used they should appear after input parameters. put all input-only parameters before any output parameters.
2. Prefer small and focused functions. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.
3. Within function input parameter lists all references must be const, while output arguments are pointers.
4. Never use std::auto\_ptr. Instead, use std::unique\_ptr.

5. Naming Rules

1. File names: all lowercase and can include underscores (\_) or dashes (-). For example, my\_useful\_class.cc.
2. Type names: start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass, MyExcitingEnum.
3. Variable names (including function parameters): all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores. For instance, a\_local\_variable, a\_struct\_data\_member.
4. Data members (but not structs): follow variable naming rule, additionally have trailing underscores. For instance, a\_class\_data\_member\_.
5. Constants: named with a leading "k" followed by mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation. For instance, const int kDaysInAWeek = 7; const int kAndroid8\_0\_0 = 24; // Android 8.0.0
6. Functions: regular functions have mixed case; accessors and mutators may be named like variables.
7. Namespace: all lower-case. Top-level namespace names are based on the project name.
8. Enumerate: mixed type like constants. For instance, enum UrlTableErrors.
9. Macros: all upper case letters, include underscores. For instance, #define PI\_ROUNDED 3.0.

6. Comment Style

1. File comments: starts with license boilerplate. If a .h declares multiple abstractions, the file-level comment should broadly describe the contents of the file, and how the abstractions are related.
2. Class comments: describes what it is and how it should be used. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.
3. Function comments: 1) almost every function **declaration** should have comments immediately preceding it that describe what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g. simple accessors for obvious properties of the class). 2) function **definition** comments should be explanatory. Describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative.
4. Class data member comments: describes the purpose, and describes the existence and meaning of sentinel values, such as nullptr or -1, when they are not obvious.
5. Global variable comments: describes what they are, what they are used for, and (if unclear) why it needs to be global.
6. Function argument comments: 1) If the argument is a literal constant, replace it with a named constant. 2) Consider changing the function signature to replace a bool argument with an enum argument. This will make the argument values self-describing. 3) For functions that have several configuration options, consider defining a single class or struct to hold all the options , and pass an instance of that.

// What are these arguments?

const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);

// Put configurations into a struct

ProductOptions options;

options.set\_precision\_decimals(7);

options.set\_use\_cache(ProductOptions::kDontUseCache);

const DecimalNumber product =

CalculateProduct(values, options, /\*completion\_callback=\*/nullptr);

7. Code Formatting

1. Function declaration & definition: 1) Each line of text in your code should be at most 80 characters long. Use only spaces, and indent 2 spaces at a time. 2) Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.
   * Choose good parameter names.
   * A parameter name may be omitted only if the parameter is not used in the function's definition.
   * If you cannot fit the return type and the function name on a single line, break between them. If you break after the return type of a function declaration or definition, do not indent.
   * The open parenthesis is always on the same line as the function name.
   * There is never a space between the function name and the open parenthesis.
   * There is never a space between the parentheses and the parameters.
   * The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
   * The close curly brace is either on the last line by itself or on the same line as the open curly brace.
   * There should be a space between the close parenthesis and the open curly brace.
   * All parameters should be aligned if possible.
   * Default indentation is 2 spaces.
   * Wrapped parameters have a 4 space indent.
2. Conditionals: Prefer no spaces inside parentheses. The if and else keywords belong on separate lines.
3. Pointer and reference: No spaces around period or arrow. Pointer operators do not have trailing spaces. When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name.
4. Preprocessor directives: a preprocessor directive should always be at the beginning of the line.
5. Class: Sections in public, protected and private order, each indented one space; otherwise are 2 space indented.
6. White space: 1) Never put trailing whitespace at the end of a line. 2) No spaces inside empty braces. 3) Binary operators usually have spaces around them, but no spaces separating unary operators and their arguments. 4) Don't put more than one or two blank lines between functions.

// If you have too much text to fit on one line:

ReturnType ReallyLongFunctionName(Type par\_name1, Type par\_name2,

Type par\_name3) {

DoSomething();

...

}

// or if you cannot fit even the first parameter:

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(

Type par\_name1, // 4 space indent

Type par\_name2,

Type par\_name3) {

DoSomething(); // 2 space indent

...

}

// Conditionals

if (condition) { // no spaces inside parentheses

... // 2 space indent.

} else if (...) { // The else goes on the same line as the closing brace.

...

} else {

...

}