**C++ Coding Convention Document**

**C++ Version[](https://google.github.io/styleguide/cppguide.html#C++_Version)**

Currently, code should target C++17, i.e., should not use C++2x features. Consider portability to other environments before using features from C++14 and C++17 in your project.

**Header Files[](https://google.github.io/styleguide/cppguide.html#Header_Files)**

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

**Forward Declarations[](https://google.github.io/styleguide/cppguide.html#Forward_Declarations)**

Avoid using forward declarations where possible. Instead, #include the headers you need.

**Names and Order of Includes[](https://google.github.io/styleguide/cppguide.html#Names_and_Order_of_Includes)**

Include headers in the following order: Related header, C system headers, C++ standard library headers, other libraries' headers, your project's headers.

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory aliases . (the current directory) or .. (the parent directory). For example, google-awesome-project/src/base/logging.h should be included as:

#include "base/logging.h"

In *dir/foo*.cpp or *dir/foo\_test*.cpp, whose main purpose is to implement or test the stuff in *dir2/foo2*.h, order your includes as follows:

1. *dir2/foo2*.h.
2. A blank line
3. C system headers (more precisely: headers in angle brackets with the .h extension), e.g. <unistd.h>, <stdlib.h>.
4. A blank line
5. C++ standard library headers (without file extension), e.g. <algorithm>, <cstddef>.
6. A blank line
7. Other libraries' .h files.
8. Your project's .h files.

Separate each non-empty group with one blank line.

**Scoping[](https://google.github.io/styleguide/cppguide.html#Scoping)**

**Static Variables[](https://google.github.io/styleguide/cppguide.html#Unnamed_Namespaces_and_Static_Variables)**

When definitions in a .cpp file do not need to be referenced outside that file, declare them static. Do not use either of these constructs in .h files.

Functions and variables can also be given internal linkage by declaring them static. This means that anything you're declaring can't be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

Use of internal linkage in .cpp files is encouraged for all code that does not need to be referenced elsewhere. Do not use internal linkage in .h files.

**Nonmember, Static Member, and Global Functions[](https://google.github.io/styleguide/cppguide.html#Nonmember,_Static_Member,_and_Global_Functions)**

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Do not use a class simply to group static functions. Static methods of a class should generally be closely related to instances of the class or the class's static data.

Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Do not create classes only to group static member functions; this is no different than just giving the function names a common prefix, and such grouping is usually unnecessary anyway.

If you define a nonmember function and it is only needed in its .cpp file, use [internal linkage](https://google.github.io/styleguide/cppguide.html#Unnamed_Namespaces_and_Static_Variables) to limit its scope.

**Local Variables[](https://google.github.io/styleguide/cppguide.html#Local_Variables)**

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

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.:

int i;

i = f(); // Bad -- initialization separate from declaration.

int j = g(); // Good -- declaration has initialization.

std::vector<int> v;

v.push\_back(1); // Prefer initializing using brace initialization.

v.push\_back(2);

std::vector<int> v = {1, 2}; // Good -- v starts initialized.

Variables needed for if, while and for statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

while (const char\* p = strchr(str, '/')) str = p + 1;

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

// Inefficient implementation:

for (int i = 0; i < 1000000; ++i) {

Foo f; // My ctor and dtor get called 1000000 times each.

f.DoSomething(i);

}

It may be more efficient to declare such a variable used in a loop outside that loop:

Foo f; // My ctor and dtor get called once each.

for (int i = 0; i < 1000000; ++i) {

f.DoSomething(i);

}

**Static and Global Variables[](https://google.github.io/styleguide/cppguide.html#Static_and_Global_Variables)**

Objects with [static storage duration](http://en.cppreference.com/w/cpp/language/storage_duration#Storage_duration) are forbidden unless they are [trivially destructible](http://en.cppreference.com/w/cpp/types/is_destructible). Informally this means that the destructor does not do anything, even taking member and base destructors into account. More formally it means that the type has no user-defined or virtual destructor and that all bases and non-static members are trivially destructible. Static function-local variables may use dynamic initialization.

**Decision on initialization**

Initialization is a more complex topic. This is because we must not only consider whether class constructors execute, but we must also consider the evaluation of the initializer:

int n = 5; // fine

int m = f(); // ? (depends on f)

Foo x; // ? (depends on Foo::Foo)

Bar y = g(); // ? (depends on g and on Bar::Bar)

All but the first statement expose us to indeterminate initialization ordering.

The concept we are looking for is called *constant initialization* in the formal language of the C++ standard. It means that the initializing expression is a constant expression, and if the object is initialized by a constructor call, then the constructor must be specified as constexpr, too:

struct Foo { constexpr Foo(int) {} };

int n = 5; // fine, 5 is a constant expression

Foo x(2); // fine, 2 is a constant expression and the chosen constructor is constexpr

Foo a[] = { Foo(1), Foo(2), Foo(3) }; // fine

Constant initialization is always allowed. Constant initialization of static storage duration variables should be marked with constexpr or where possible the [ABSL\_CONST\_INIT](https://github.com/abseil/abseil-cpp/blob/03c1513538584f4a04d666be5eb469e3979febba/absl/base/attributes.h#L540) attribute. Any non-local static storage duration variable that is not so marked should be presumed to have dynamic initialization, and reviewed very carefully.

By contrast, the following initializations are problematic:

// Some declarations used below.

time\_t time(time\_t\*); // not constexpr!

int f(); // not constexpr!

struct Bar { Bar() {} };

// Problematic initializations.

time\_t m = time(nullptr); // initializing expression not a constant expression

Foo y(f()); // ditto

Bar b; // chosen constructor Bar::Bar() not constexpr

Dynamic initialization of nonlocal variables is discouraged, and in general it is forbidden. However, we do permit it if no aspect of the program depends on the sequencing of this initialization with respect to all other initializations. Under those restrictions, the ordering of the initialization does not make an observable difference. For example:

int p = getpid(); // allowed, as long as no other static variable

// uses p in its own initialization

Dynamic initialization of static local variables is allowed (and common).

**Classes[](https://google.github.io/styleguide/cppguide.html#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[](https://google.github.io/styleguide/cppguide.html#Doing_Work_in_Constructors)**

Avoid virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error.

It is possible to perform arbitrary initialization in the body of the constructor.

* No need to worry about whether the class has been initialized or not.
* Objects that are fully initialized by constructor call can be const and may also be easier to use with standard containers or algorithms.
* If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
* There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions.
* If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a bool IsValid() state checking mechanism (or similar) which is easy to forget to call.
* You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

Constructors should never call virtual functions. If appropriate for your code, terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or Init() method. Avoid Init() methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

**Structs vs. Classes[](https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes)**

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

The struct and class keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

structs should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. All fields must be public, and accessed directly rather than through getter/setter methods. The struct must not have invariants that imply relationships between different fields, since direct user access to those fields may break those invariants. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, Initialize(), Reset().

If more functionality or invariants are required, a class is more appropriate. If in doubt, make it a class.

For consistency with STL, you can use struct instead of class for stateless types, such as traits, [template metafunctions](https://google.github.io/styleguide/cppguide.html#Template_metaprogramming), and some functors.

Note that member variables in structs and classes have [different naming rules](https://google.github.io/styleguide/cppguide.html#Variable_Names).

**Inheritance[](https://google.github.io/styleguide/cppguide.html#Inheritance)**

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

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the base class defines. "Interface inheritance" is inheritance from a pure abstract base class (one with no state or defined methods); all other inheritance is "implementation inheritance".

Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type. Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation.

Multiple inheritance is especially problematic, because it often imposes a higher performance overhead (in fact, the performance drop from single inheritance to multiple inheritance can often be greater than the performance drop from ordinary to virtual dispatch), and because it risks leading to "diamond" inheritance patterns, which are prone to ambiguity, confusion, and outright bugs.

All inheritance should be public. If you want to do private inheritance, you should be including an instance of the base class as a member instead.

Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the "is-a" case: Bar subclasses Foo if it can reasonably be said that Bar "is a kind of" Foo.

Limit the use of protected to those member functions that might need to be accessed from subclasses. Note that [data members should be private](https://google.github.io/styleguide/cppguide.html#Access_Control).

Explicitly annotate overrides of virtual functions or virtual destructors with exactly one of an override or (less frequently) final specifier. Do not use virtual when declaring an override. Rationale: A function or destructor marked override or final that is not an override of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is virtual or not.

Multiple inheritance is permitted, but multiple *implementation* inheritance is strongly discouraged.

**Access Control[](https://google.github.io/styleguide/cppguide.html#Access_Control)**

Make classes' data members private, unless they are [constants](https://google.github.io/styleguide/cppguide.html#Constant_Names). This simplifies reasoning about invariants, at the cost of some easy boilerplate in the form of accessors (usually const) if necessary.

For technical reasons, we allow data members of a test fixture class in a cpp file to be protected when using [Google Test](https://github.com/google/googletest)).

**Declaration Order[](https://google.github.io/styleguide/cppguide.html#Declaration_Order)**

Group similar declarations together, placing private parts earlier.

A class definition should usually start with a private: section, followed by protected:, then public: . Omit sections that would be empty.

**Functions[](https://google.github.io/styleguide/cppguide.html#Functions)**

**Output Parameters[](https://google.github.io/styleguide/cppguide.html#Output_Parameters)**

The output of a C++ function is naturally provided via a return value and sometimes via output parameters.

Prefer using return values over output parameters: they improve readability, and often provide the same or better performance. If output-only parameters are used, they should appear after input parameters.

Parameters are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be pointers to non-const.

When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

**Write Short Functions[](https://google.github.io/styleguide/cppguide.html#Write_Short_Functions)**

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

**Reference Arguments[](https://google.github.io/styleguide/cppguide.html#Reference_Arguments)**

All parameters passed by lvalue reference must be labeled const.

In C, if a function needs to modify a variable, the parameter must use a pointer, eg int foo(int \*pval). In C++, the function can alternatively declare a reference parameter: int foo(int &val).

Defining a parameter as reference avoids ugly code like (\*pval)++. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

References can be confusing, as they have value syntax but pointer semantics.

Within function parameter lists all references must be const:

void Foo(const std::string &in, std::string \*out);

In fact it is a very strong convention in Google code that input arguments are values or const references while output arguments are pointers. Input parameters may be const pointers, but we never allow non-const reference parameters except when required by convention, e.g., swap().

However, there are some instances where using const T\* is preferable to const T& for input parameters. For example:

* You want to pass in a null pointer.
* The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as const T&. Using const T\* instead communicates to the reader that the input is somehow treated differently. So if you choose const T\* rather than const T&, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

**Default Arguments[](https://google.github.io/styleguide/cppguide.html#Default_Arguments)**

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value.

**Other C++ Features[](https://google.github.io/styleguide/cppguide.html#Other_C++_Features)**

**Exceptions[](https://google.github.io/styleguide/cppguide.html#Exceptions)**

We will utilize exceptions to test code. These exceptions shall be used primarily to make sure that the correct type is passed to a function call.

**Casting[](https://google.github.io/styleguide/cppguide.html#Casting)**

Use C++-style casts like static\_cast<float>(double\_value), or brace initialization for conversion of arithmetic types like int64 y = int64{1} << 42. Do not use cast formats like int y = (int)x or int y = int(x) (but the latter is okay when invoking a constructor of a class type).

C++ introduced a different cast system from C that distinguishes the types of cast operations.

The problem with C casts is the ambiguity of the operation; sometimes you are doing a *conversion* (e.g., (int)3.5) and sometimes you are doing a *cast* (e.g., (int)"hello"). Brace initialization and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

**Use of const[](https://google.github.io/styleguide/cppguide.html#Use_of_const)**

In APIs, use const whenever it makes sense.

Declared variables and parameters can be preceded by the keyword const to indicate the variables are not changed (e.g., const int foo). Class functions can have the const qualifier to indicate the function does not change the state of the class member variables (e.g., class Foo { int Bar(char c) const; };).

Easier for people to understand how variables are being used. Allows the compiler to do better type checking, and, conceivably, generate better code. Helps people convince themselves of program correctness because they know the functions they call are limited in how they can modify your variables. Helps people know what functions are safe to use without locks in multi-threaded programs.

const is viral: if you pass a const variable to a function, that function must have const in its prototype (or the variable will need a const\_cast). This can be a particular problem when calling library functions.

We strongly recommend using const in APIs (i.e. on function parameters, methods, and non-local variables) wherever it is meaningful and accurate. This provides consistent, mostly compiler-verified documentation of what objects an operation can mutate. Having a consistent and reliable way to distinguish reads from writes is critical to writing thread-safe code, and is useful in many other contexts as well. In particular:

* If a function guarantees that it will not modify an argument passed by reference or by pointer, the corresponding function parameter should be a reference-to-const (const T&) or pointer-to-const (const T\*), respectively.
* For a function parameter passed by value, const has no effect on the caller, thus is not recommended in function declarations.
* Declare methods to be const unless they alter the logical state of the object (or enable the user to modify that state, e.g. by returning a non-const reference, but that's rare), or they can't safely be invoked concurrently.

Using const on local variables is neither encouraged nor discouraged.

All of a class's const operations should be safe to invoke concurrently with each other. If that's not feasible, the class must be clearly documented as "thread-unsafe".

The const keyword should be placed after the type name when a constant is declared.

**0 and nullptr/NULL[](https://google.github.io/styleguide/cppguide.html#0_and_nullptr/NULL)**

Use nullptr for pointers, and '\0' for chars (and not the 0 literal).

For pointers (address values), use nullptr, as this provides type-safety.

For C++03 projects, prefer NULL to 0. While the values are equivalent, NULL looks more like a pointer to the reader, and some C++ compilers provide special definitions of NULL which enable them to give useful warnings. Never use NULL for numeric (integer or floating-point) values.

Use '\0' for the null character. Using the correct type makes the code more readable.

**Naming[](https://google.github.io/styleguide/cppguide.html#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[](https://google.github.io/styleguide/cppguide.html#General_Naming_Rules)**

Optimize for readability using names that would be clear even to people on a different team.

Use names that describe the purpose or intent of the object. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Minimize the use of abbreviations that would likely be unknown to someone outside your project (especially acronyms and initialisms). Do not abbreviate by deleting letters within a word. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia. Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, n may be a fine name within a 5-line function, but within the scope of a class, it's likely too vague.

class MyClass {

public:

int CountFooErrors(const std::vector<Foo>& foos) {

int n = 0; // Clear meaning given limited scope and context

for (const auto& foo : foos) {

...

++n;

}

return n;

}

void DoSomethingImportant() {

std::string fqdn = ...; // Well-known abbreviation for Fully Qualified Domain Name

}

private:

const int kMaxAllowedConnections = ...; // Clear meaning within context

};

class MyClass {

public:

int CountFooErrors(const std::vector<Foo>& foos) {

int total\_number\_of\_foo\_errors = 0; // Overly verbose given limited scope and context

for (int foo\_index = 0; foo\_index < foos.size(); ++foo\_index) { // Use idiomatic `i`

...

++total\_number\_of\_foo\_errors;

}

return total\_number\_of\_foo\_errors;

}

void DoSomethingImportant() {

int cstmr\_id = ...; // Deletes internal letters

}

private:

const int kNum = ...; // Unclear meaning within broad scope

};

Note that certain universally-known abbreviations are OK, such as i for an iteration variable and T for a template parameter.

For the purposes of the naming rules below, a "word" is anything that you would write in English without internal spaces. This includes abbreviations and acronyms; e.g., for "[camel case](https://en.wikipedia.org/wiki/Camel_case)" or "Pascal case," in which the first letter of each word is capitalized, use a name like StartRpc(), not StartRPC().

Template parameters should follow the naming style for their category: type template parameters should follow the rules for [type names](https://google.github.io/styleguide/cppguide.html#Type_Names), and non-type template parameters should follow the rules for [variable names](https://google.github.io/styleguide/cppguide.html#Variable_Names).

**File Names[](https://google.github.io/styleguide/cppguide.html#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 "\_".

Examples of acceptable file names:

* my\_useful\_classcpp
* my-useful-classcpp
* myusefulclasscpp
* myusefulclass\_testcpp // \_unittest and \_regtest are deprecated.

C++ files should end in cpp and header files should end in .h. Files that rely on being textually included at specific points should end in .inc (see also the section on [self-contained headers](https://google.github.io/styleguide/cppguide.html#Self_contained_Headers)).

Do not use filenames that already exist in /usr/include, such as db.h.

In general, make your filenames very specific. For example, use http\_server\_logs.h rather than logs.h. A very common case is to have a pair of files called, e.g., foo\_bar.h and foo\_barcpp, defining a class called FooBar.

**Type Names[](https://google.github.io/styleguide/cppguide.html#Type_Names)**

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

The names of all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

// classes and structs

class UrlTable { ...

class UrlTableTester { ...

struct UrlTableProperties { ...

// typedefs

typedef hash\_map<UrlTableProperties \*, std::string> PropertiesMap;

// using aliases

using PropertiesMap = hash\_map<UrlTableProperties \*, std::string>;

// enums

enum UrlTableErrors { ...

**Variable Names[](https://google.github.io/styleguide/cppguide.html#Variable_Names)**

The names of variables (including function parameters) and data members are all lowercase, with underscores between words.

**Constant Names[](https://google.github.io/styleguide/cppguide.html#Constant_Names)**

Variables declared const, and whose value is fixed for the duration of the program, are 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 example:

const int kDaysInAWeek = 7;

const int kAndroid8\_0\_0 = 24; // Android 8.0.0

All such variables with static storage duration (i.e. statics and globals, see [Storage Duration](http://en.cppreference.com/w/cpp/language/storage_duration#Storage_duration) for details) should be named this way. This convention is optional for variables of other storage classes, e.g. automatic variables, otherwise the usual variable naming rules apply.

**Function Names[](https://google.github.io/styleguide/cppguide.html#Function_Names)**

Regular functions have mixed case; accessors and mutators may be named like variables.

Ordinarily, functions should start with a capital letter and have a capital letter for each new word.

AddTableEntry()

DeleteUrl()

OpenFileOrDie()

(The same naming rule applies to class- and namespace-scope constants that are exposed as part of an API and that are intended to look like functions, because the fact that they're objects rather than functions is an unimportant implementation detail.)

Accessors and mutators (get and set functions) may be named like variables. These often correspond to actual member variables, but this is not required. For example, int count() and void set\_count(int count).

**Enumerator Names[](https://google.github.io/styleguide/cppguide.html#Enumerator_Names)**

Enumerators (for both scoped and unscoped enums) should be named *either* like [constants](https://google.github.io/styleguide/cppguide.html#Constant_Names) or like [macros](https://google.github.io/styleguide/cppguide.html#Macro_Names): either kEnumName or ENUM\_NAME.

Preferably, the individual enumerators should be named like [constants](https://google.github.io/styleguide/cppguide.html#Constant_Names). However, it is also acceptable to name them like [macros](https://google.github.io/styleguide/cppguide.html#Macro_Names). The enumeration name, UrlTableErrors (and AlternateUrlTableErrors), is a type, and therefore mixed case.

enum UrlTableErrors {

kOk = 0,

kErrorOutOfMemory,

kErrorMalformedInput,

};

enum AlternateUrlTableErrors {

OK = 0,

OUT\_OF\_MEMORY = 1,

MALFORMED\_INPUT = 2,

};

**Exceptions to Naming Rules[](https://google.github.io/styleguide/cppguide.html#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.

bigopen()

function name, follows form of open()

uint

typedef

bigpos

struct or class, follows form of pos

sparse\_hash\_map

STL-like entity; follows STL naming conventions

LONGLONG\_MAX

a constant, as in INT\_MAX

**Comments[](https://google.github.io/styleguide/cppguide.html#Comments)**

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[](https://google.github.io/styleguide/cppguide.html#Comment_Style)**

Use either the // or /\* \*/ syntax, as long as you are consistent.

You can use either the // or the /\* \*/ syntax; however, // is *much* more common. Be consistent with how you comment and what style you use where.

**File Contents**

If a .h declares multiple abstractions, the file-level comment should broadly describe the contents of the file, and how the abstractions are related. A 1 or 2 sentence file-level comment may be sufficient. The detailed documentation about individual abstractions belongs with those abstractions, not at the file level.

Do not duplicate comments in both the .h and the cpp. Duplicated comments diverge.

**Function Declarations**

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). These comments should open with descriptive verbs in the indicative mood ("Opens the file") rather than verbs in the imperative ("Open the file"). The comment describes the function; it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

* What the inputs and outputs are.
* For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
* If the function allocates memory that the caller must free.
* Whether any of the arguments can be a null pointer.
* If there are any performance implications of how a function is used.
* If the function is re-entrant. What are its synchronization assumptions?

Here is an example:

// Returns an iterator for this table. It is the client's

// responsibility to delete the iterator when it is done with it,

// and it must not use the iterator once the GargantuanTable object

// on which the iterator was created has been deleted.

//

// The iterator is initially positioned at the beginning of the table.

//

// This method is equivalent to:

// Iterator\* iter = table->NewIterator();

// iter->Seek("");

// return iter;

// If you are going to immediately seek to another place in the

// returned iterator, it will be faster to use NewIterator()

// and avoid the extra seek.

Iterator\* GetIterator() const;

However, do not be unnecessarily verbose or state the completely obvious.

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like "destroys this object" are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

**Function Definitions**

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might 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. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should *not* just repeat the comments given with the function declaration, in the .h file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

**Class Data Members**

The purpose of each class data member (also called an instance variable or member variable) must be clear. If there are any invariants (special values, relationships between members, lifetime requirements) not clearly expressed by the type and name, they must be commented. However, if the type and name suffice (int num\_events\_;), no comment is needed.

In particular, add comments to describe the existence and meaning of sentinel values, such as nullptr or -1, when they are not obvious. For example:

private:

// Used to bounds-check table accesses. -1 means

// that we don't yet know how many entries the table has.

int num\_total\_entries\_;

**Global Variables**

All global variables should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global. For example:

// The total number of tests cases that we run through in this regression test.

const int kNumTestCases = 6;

**Punctuation, Spelling, and Grammar[](https://google.github.io/styleguide/cppguide.html#Punctuation,_Spelling_and_Grammar)**

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

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

**Formatting[](https://google.github.io/styleguide/cppguide.html#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](https://raw.githubusercontent.com/google/styleguide/gh-pages/google-c-style.el).

**Line Length[](https://google.github.io/styleguide/cppguide.html#Line_Length)**

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

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

Those who favor this rule argue that it is rude to force them to resize their windows and there is no need for anything longer. Some folks are used to having several code windows side-by-side, and thus don't have room to widen their windows in any case. People set up their work environment assuming a particular maximum window width, and 80 columns has been the traditional standard. Why change it?

Proponents of change argue that a wider line can make code more readable. The 80-column limit is an hidebound throwback to 1960s mainframes; modern equipment has wide screens that can easily show longer lines.

80 characters is the maximum.

A line may exceed 80 characters if it is

* a comment line which is not feasible to split without harming readability, ease of cut and paste or auto-linking -- e.g. if a line contains an example command or a literal URL longer than 80 characters.
* a raw-string literal with content that exceeds 80 characters. Except for test code, such literals should appear near the top of a file.
* an include statement.
* a [header guard](https://google.github.io/styleguide/cppguide.html#The__define_Guard)
* a using-declaration

**Spaces vs. Tabs[](https://google.github.io/styleguide/cppguide.html#Spaces_vs._Tabs)**

Use only spaces, and indent 4 spaces at a time.

We use spaces for indentation. Do not use tabs in your code unless you set your editor to emit spaces when you hit the tab key.

**Function Declarations and Definitions[](https://google.github.io/styleguide/cppguide.html#Function_Declarations_and_Definitions)**

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](https://google.github.io/styleguide/cppguide.html#Function_Calls).

Functions look like this:

ReturnType ClassName::FunctionName(Type par\_name1, Type par\_name2) {

DoSomething();

...

}

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

ReturnType ClassName::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

...

}

Some points to note:

* 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.

Unused parameters that are obvious from context may be omitted:

class Foo {

public:

Foo(const Foo&) = delete;

Foo& operator=(const Foo&) = delete;

};

Unused parameters that might not be obvious should comment out the variable name in the function definition:

class Shape {

public:

virtual void Rotate(double radians) = 0;

};

class Circle : public Shape {

public:

void Rotate(double radians) override;

};

void Circle::Rotate(double /\*radians\*/) {}

// Bad - if someone wants to implement later, it's not clear what the

// variable means.

void Circle::Rotate(double) {}

Attributes, and macros that expand to attributes, appear at the very beginning of the function declaration or definition, before the return type:

ABSL\_MUST\_USE\_RESULT bool IsOk();

**Function Calls[](https://google.github.io/styleguide/cppguide.html#Function_Calls)**

Either write the call all on a single line, wrap the arguments at the parenthesis, or start the arguments on a new line indented by four spaces and continue at that 4 space indent. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

Function calls have the following format:

bool result = DoSomething(argument1, argument2, argument3);

If the arguments do not all fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open paren or before the close paren:

bool result = DoSomething(averyveryveryverylongargument1,

argument2, argument3);

Arguments may optionally all be placed on subsequent lines with a four space indent:

if (...) {

...

...

if (...) {

bool result = DoSomething(

argument1, argument2, // 4 space indent

argument3, argument4);

...

}

Put multiple arguments on a single line to reduce the number of lines necessary for calling a function unless there is a specific readability problem. Some find that formatting with strictly one argument on each line is more readable and simplifies editing of the arguments. However, we prioritize for the reader over the ease of editing arguments, and most readability problems are better addressed with the following techniques.

If having multiple arguments in a single line decreases readability due to the complexity or confusing nature of the expressions that make up some arguments, try creating variables that capture those arguments in a descriptive name:

int my\_heuristic = scores[x] \* y + bases[x];

bool result = DoSomething(my\_heuristic, x, y, z);

Or put the confusing argument on its own line with an explanatory comment:

bool result = DoSomething(scores[x] \* y + bases[x], // Score heuristic.

x, y, z);

If there is still a case where one argument is significantly more readable on its own line, then put it on its own line. The decision should be specific to the argument which is made more readable rather than a general policy.

Sometimes arguments form a structure that is important for readability. In those cases, feel free to format the arguments according to that structure:

// Transform the widget by a 3x3 matrix.

my\_widget.Transform(x1, x2, x3,

y1, y2, y3,

z1, z2, z3);

**Conditionals[](https://google.github.io/styleguide/cppguide.html#Conditionals)**

Prefer no spaces inside parentheses. The if and else keywords belong on separate lines.

There are two acceptable formats for a basic conditional statement. One includes spaces between the parentheses and the condition, and one does not.

The most common form is without spaces. Either is fine, but *be consistent*. If you are modifying a file, use the format that is already present. If you are writing new code, use the format that the other files in that directory or project use. If in doubt and you have no personal preference, do not add the spaces.

if (condition) { // no spaces inside parentheses

... // 2 space indent.

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

...

} else {

...

}

Note that in all cases you must have a space between the if and the open parenthesis. You must also have a space between the close parenthesis and the curly brace, if you're using one.

if(condition) { // Bad - space missing after IF.

if (condition){ // Bad - space missing before {.

if(condition){ // Doubly bad.

if (condition) { // Good - proper space after IF and before {.

Short conditional statements may be written on one line if this enhances readability. You may use this only when the line is brief and the statement does not use the else clause.

if (x == kFoo) return new Foo();

if (x == kBar) return new Bar();

This is not allowed when the if statement has an else:

// Not allowed - IF statement on one line when there is an ELSE clause

if (x) DoThis();

else DoThat();

In general, curly braces are not required for single-line statements, but they are allowed if you like them; conditional or loop statements with complex conditions or statements may be more readable with curly braces. Some projects require that an if must always have an accompanying brace.

if (condition)

DoSomething(); // 2 space indent.

if (condition) {

DoSomething(); // 2 space indent.

}

However, if one part of an if-else statement uses curly braces, the other part must too:

// Not allowed - curly on IF but not ELSE

if (condition) {

foo;

} else

bar;

// Not allowed - curly on ELSE but not IF

if (condition)

foo;

else {

bar;

}

// Curly braces around both IF and ELSE required because

// one of the clauses used braces.

if (condition) {

foo;

} else {

bar;

}

**Loops and Switch Statements[](https://google.github.io/styleguide/cppguide.html#Loops_and_Switch_Statements)**

Switch statements may use braces for blocks. Annotate non-trivial fall-through between cases. Braces are optional for single-statement loops. Empty loop bodies should use either empty braces or continue.

case blocks in switch statements can have curly braces or not, depending on your preference. If you do include curly braces they should be placed as shown below.

If not conditional on an enumerated value, switch statements should always have a default case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, treat this as an error. For example:

switch (var) {

case 0: { // 2 space indent

... // 4 space indent

break;

}

case 1: {

...

break;

}

default: {

assert(false);

}

}

Fall-through from one case label to another must be annotated using the ABSL\_FALLTHROUGH\_INTENDED; macro (defined in absl/base/macros.h). ABSL\_FALLTHROUGH\_INTENDED; should be placed at a point of execution where a fall-through to the next case label occurs. A common exception is consecutive case labels without intervening code, in which case no annotation is needed.

switch (x) {

case 41: // No annotation needed here.

case 43:

if (dont\_be\_picky) {

// Use this instead of or along with annotations in comments.

ABSL\_FALLTHROUGH\_INTENDED;

} else {

CloseButNoCigar();

break;

}

case 42:

DoSomethingSpecial();

ABSL\_FALLTHROUGH\_INTENDED;

default:

DoSomethingGeneric();

break;

}

Braces are optional for single-statement loops.

for (int i = 0; i < kSomeNumber; ++i)

printf("I love you\n");

for (int i = 0; i < kSomeNumber; ++i) {

printf("I take it back\n");

}

Empty loop bodies should use either an empty pair of braces or continue with no braces, rather than a single semicolon.

while (condition) {

// Repeat test until it returns false.

}

for (int i = 0; i < kSomeNumber; ++i) {} // Good - one newline is also OK.

while (condition) continue; // Good - continue indicates no logic.

while (condition); // Bad - looks like part of do/while loop.

**Pointer and Reference Expressions[](https://google.github.io/styleguide/cppguide.html#Pointer_and_Reference_Expressions)**

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

The following are examples of correctly-formatted pointer and reference expressions:

x = \*p;

p = &x;

x = r.y;

x = r->y;

Note that:

* There are no spaces around the period or arrow when accessing a member.
* Pointer operators have no space after the \* or &.

When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name:

// These are fine, space preceding.

char \*c;

const std::string &str;

// These are fine, space following.

char\* c;

const std::string& str;

You should do this consistently within a single file, so, when modifying an existing file, use the style in that file.

It is allowed (if unusual) to declare multiple variables in the same declaration, but it is disallowed if any of those have pointer or reference decorations. Such declarations are easily misread.

// Fine if helpful for readability.

int x, y;

int x, \*y; // Disallowed - no & or \* in multiple declaration

char \* c; // Bad - spaces on both sides of \*

const std::string & str; // Bad - spaces on both sides of &

**Boolean Expressions[](https://google.github.io/styleguide/cppguide.html#Boolean_Expressions)**

When you have a boolean expression that is longer than the [standard line length](https://google.github.io/styleguide/cppguide.html#Line_Length), be consistent in how you break up the lines.

In this example, the logical AND operator is always at the end of the lines:

if (this\_one\_thing > this\_other\_thing &&

a\_third\_thing == a\_fourth\_thing &&

yet\_another && last\_one) {

...

}

Note that when the code wraps in this example, both of the && logical AND operators are at the end of the line. This is more common in Google code, though wrapping all operators at the beginning of the line is also allowed. Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately. Also note that you should always use the punctuation operators, such as && and ~, rather than the word operators, such as and and compl.

**Return Values[](https://google.github.io/styleguide/cppguide.html#Return_Values)**

Do not needlessly surround the return expression with parentheses.

Use parentheses in return expr; only where you would use them in x = expr;.

return result; // No parentheses in the simple case.

// Parentheses OK to make a complex expression more readable.

return (some\_long\_condition &&

another\_condition);

return (value); // You wouldn't write var = (value);

return(result); // return is not a function!

**Variable and Array Initialization[](https://google.github.io/styleguide/cppguide.html#Variable_and_Array_Initialization)**

Your choice of =, (), or {}.

You may choose between =, (), and {}; the following are all correct:

int x = 3;

int x(3);

int x{3};

std::string name = "Some Name";

std::string name("Some Name");

std::string name{"Some Name"};

Be careful when using a braced initialization list {...} on a type with an std::initializer\_list constructor. A nonempty *braced-init-list* prefers the std::initializer\_list constructor whenever possible. Note that empty braces {} are special, and will call a default constructor if available. To force the non-std::initializer\_list constructor, use parentheses instead of braces.

std::vector<int> v(100, 1); // A vector containing 100 items: All 1s.

std::vector<int> v{100, 1}; // A vector containing 2 items: 100 and 1.

Also, the brace form prevents narrowing of integral types. This can prevent some types of programming errors.

int pi(3.14); // OK -- pi == 3.

int pi{3.14}; // Compile error: narrowing conversion.

**Class Format[](https://google.github.io/styleguide/cppguide.html#Class_Format)**

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

The basic format for a class definition is:

class MyClass : public OtherClass {

public: // Note the 1 space indent!

MyClass(); // Regular 2 space indent.

explicit MyClass(int var);

~MyClass() {}

void SomeFunction();

void SomeFunctionThatDoesNothing() {

}

void set\_some\_var(int var) { some\_var\_ = var; }

int some\_var() const { return some\_var\_; }

private:

bool SomeInternalFunction();

int some\_var\_;

int some\_other\_var\_;

};

Things to note:

* Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
* The public:, protected:, and private: keywords should be indented one space.
* Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
* Do not leave a blank line after these keywords.
* The public section should be first, followed by the protected and finally the private section.
* See [Declaration Order](https://google.github.io/styleguide/cppguide.html#Declaration_Order) for rules on ordering declarations within each of these sections.

**Constructor Initializer Lists[](https://google.github.io/styleguide/cppguide.html#Constructor_Initializer_Lists)**

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

**Horizontal Whitespace[](https://google.github.io/styleguide/cppguide.html#Horizontal_Whitespace)**

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

**General**

void f(bool b) { // Open braces should always have a space before them.

...

int i = 0; // Semicolons usually have no space before them.

// Spaces inside braces for braced-init-list are optional. If you use them,

// put them on both sides!

int x[] = { 0 };

int x[] = {0};

// Spaces around the colon in inheritance and initializer lists.

class Foo : public Bar {

public:

// For inline function implementations, put spaces between the braces

// and the implementation itself.

Foo(int b) : Bar(), baz\_(b) {} // No spaces inside empty braces.

void Reset() { baz\_ = 0; } // Spaces separating braces from implementation.

...

Adding trailing whitespace can cause extra work for others editing the same file, when they merge, as can removing existing trailing whitespace. So: Don't introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation (preferably when no-one else is working on the file).

**Loops and Conditionals**

if (b) { // Space after the keyword in conditions and loops.

} else { // Spaces around else.

}

while (test) {} // There is usually no space inside parentheses.

switch (i) {

for (int i = 0; i < 5; ++i) {

// Loops and conditions may have spaces inside parentheses, but this

// is rare. Be consistent.

switch ( i ) {

if ( test ) {

for ( int i = 0; i < 5; ++i ) {

// For loops always have a space after the semicolon. They may have a space

// before the semicolon, but this is rare.

for ( ; i < 5 ; ++i) {

...

// Range-based for loops always have a space before and after the colon.

for (auto x : counts) {

...

}

switch (i) {

case 1: // No space before colon in a switch case.

...

case 2: break; // Use a space after a colon if there's code after it.

**Operators**

// Assignment operators always have spaces around them.

x = 0;

// Other binary operators usually have spaces around them, but it's

// OK to remove spaces around factors. Parentheses should have no

// internal padding.

v = w \* x + y / z;

v = w\*x + y/z;

v = w \* (x + z);

// No spaces separating unary operators and their arguments.

x = -5;

++x;

if (x && !y)

...

**Templates and Casts**

// No spaces inside the angle brackets (< and >), before

// <, or between >( in a cast

std::vector<std::string> x;

y = static\_cast<char\*>(x);

// Spaces between type and pointer are OK, but be consistent.

std::vector<char \*> x;

**Vertical Whitespace[](https://google.github.io/styleguide/cppguide.html#Vertical_Whitespace)**

Minimize use of vertical whitespace.

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be sparing with your use of blank lines. A blank line within a block of code serves like a paragraph break in prose: visually separating two thoughts.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Use whitespace purposefully to provide separation in that flow.

Some rules of thumb to help when blank lines may be useful:

* Blank lines at the beginning or end of a function do not help readability.
* Blank lines inside a chain of if-else blocks may well help readability.
* A blank line before a comment line usually helps readability — the introduction of a new comment suggests the start of a new thought, and the blank line makes it clear that the comment goes with the following thing instead of the preceding.