Google C++ Style Guide

Table of Contents

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.

Self-contained Headers

Header files should be self-contained and end in `.h`. Files that are meant for textual inclusion, but are not headers, should end in `.inc`. Separate `-inl.h` headers are disallowed.

All header files should be self-contained. In other words, users and refactoring tools should not have to adhere to special conditions in order to include the header. Specifically, a header should have header guards, should include all other headers it needs, and should not require any particular symbols to be defined.

There are rare cases where a file is not meant to be self-contained, but instead is meant to be textually included at a specific point in the code. Examples are files that need to be included multiple times or platform-specific extensions that essentially are part of other headers. Such files should use the file extension `.inc`.

If a template or inline function is declared in a `.h` file, define it in that same file. The definitions of these constructs must be included into every `.cc` file that uses them, or the program may fail to link in some build configurations. Do not move these definitions to separate `-inl.h` files.

As an exception, a function template that is explicitly instantiated for all relevant sets of template arguments, or that is a private member of a class, may be defined in the only `.cc` file that instantiates the template.

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_`.

To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file `foo/src/bar/baz.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

Forward Declarations

You may forward declare ordinary classes in order to avoid unnecessary `#include`s.

Definition:

A "forward declaration" is a declaration of a class, function, or template without an associated definition. #include lines can often be replaced with forward declarations of whatever symbols are actually used by the client code.

Pros:

- Unnecessary #includes force the compiler to open more files and process more input.
- They can also force your code to be recompiled more often, due to changes in the header.

Cons:

- It can be difficult to determine the correct form of a forward declaration in the presence of features like templates, typedefs, default parameters, and using declarations.
- It can be difficult to determine whether a forward declaration or a full #include is needed for a given piece of code, particularly when implicit conversion operations are involved. In extreme cases, replacing an #include with a forward declaration can silently change the meaning of code.
- Forward declaring multiple symbols from a header can be more verbose than simply #includeing the header.
- Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs; for example, widening a parameter type, or adding a template parameter with a default value.
- Forward declaring symbols from namespace std:: usually yields undefined behavior.
- Structuring code to enable forward declarations (e.g. using pointer members instead of object members) can make the code slower and more complex.
- The practical efficiency benefits of forward declarations are unproven.

Decision:

- When using a function declared in a header file, always #include that header.
- When using a class template, prefer to #include its header file.
- When using an ordinary class, relying on a forward declaration is OK, but be wary of situations where a forward declaration may be insufficient or incorrect; when in doubt, just #include the appropriate header.
- Do not replace data members with pointers just to avoid an #include.

Please see Names and Order of Includes for rules about when to #include a header.

Inline Functions

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

Definition:

You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism.

Pros:

Inlining a function can generate more efficient object code, as long as the inlined function is small. Feel free to inline accessors and mutators, and other short, performance-critical functions.

Cons:

Overuse of inlining can actually make programs slower. Depending on a function's size, inlining it can cause the code size to increase or decrease. Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size. On modern processors smaller code usually runs faster due to better use of the instruction cache.

Decision:

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements (unless, in the common case, the loop or switch statement is never executed).

It is important to know that functions are not always inlined even if they are declared as such; for example, virtual and recursive functions are not normally inlined. Usually recursive functions should not be inline. The main reason for making a virtual function inline is to place its definition in the class, either for convenience or to document its behavior, e.g., for accessors and mutators.

Function Parameter Ordering

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

Parameters to C/C++ functions 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 non-`const` pointers. 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.

Names and Order of Includes

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

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts `.` (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`*`.cc` or *`dir/foo_test`*`.cc`, whose main purpose is to implement or test the stuff in *`dir2/foo2`*`.h`, order your includes as follows:

1. *`dir2/foo2`*`.h`.
2. C system files.
3. C++ system files.
4. Other libraries' `.h` files.
5. Your project's `.h` files.

With the preferred ordering, if *`dir2/foo2`*`.h` omits any necessary includes, the build of *`dir/foo`*`.cc` or *`dir/foo`*`_test.cc` will break. Thus, this rule ensures that build breaks show up first for the people working on these files, not for innocent people in other packages.

dir/foo`.cc` and *dir2/foo2*`.h` are usually in the same directory (e.g. `base/basictypes_test.cc` and `base/basictypes.h`), but may sometimes be in different directories too.

Within each section the includes should be ordered alphabetically. Note that older code might not conform to this rule and should be fixed when convenient.

You should include all the headers that define the symbols you rely upon (except in cases of forward declaration). If you rely on symbols from `bar.h`, don't count on the fact that you included `foo.h` which (currently) includes `bar.h`: include `bar.h` yourself, unless `foo.h` explicitly demonstrates its intent to provide you the symbols of `bar.h`. However, any includes present in the related header do not need to be included again in the related `cc` (i.e., `foo.cc` can rely on `foo.h`'s includes).

For example, the includes in `google-awesome-project/src/foo/internal/fooserver.cc` might look like this:

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

Exception:

Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. Of course, keep your system-specific code small and localized. Example:

```
#include "foo/public/fooserver.h"

#include "base/port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

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*. Do not use inline namespaces.

Definition:

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

Pros:

Namespaces provide a (hierarchical) axis of naming, in addition to the (also hierarchical) name axis provided by classes.

For example, if two different projects have a class `Foo` in the global scope, these symbols may collide at compile time or at runtime. If each project places their code in a namespace, `project1::Foo` and `project2::Foo` are now distinct symbols that do not collide.

Inline namespaces automatically place their names in the enclosing scope. Consider the following snippet, for example:

```
namespace X {
inline namespace Y {
  void foo();
}
}
```

The expressions `X::Y::foo()` and `X::foo()` are interchangeable. Inline namespaces are primarily intended for ABI compatibility across versions.

Cons:

Namespaces can be confusing, because they provide an additional (hierarchical) axis of naming, in addition to the (also hierarchical) name axis provided by classes.

Inline namespaces, in particular, can be confusing because names aren't actually restricted to the namespace where they are declared. They are only useful as part of some larger versioning policy.

Use of unnamed namespaces in header files can easily cause violations of the C++ One Definition Rule (ODR).

Decision:

Use namespaces according to the policy described below. Terminate namespaces with comments as shown in the given examples.

Unnamed Namespaces

- Unnamed namespaces are allowed and even encouraged in `.cc` files, to avoid link time naming conflicts:

```
namespace {                               // This is in a .c

// The content of a namespace is not indented.
//
// This function is guaranteed not to generate a collidi
// with other symbols at link time, and is only visible
// callers in this .cc file.
bool UpdateInternals(Frobber* f, int newval) {
  ...
}

}  // namespace
```

  However, file-scope declarations that are associated with a particular class may be declared in that class as types, static data members or static member functions rather than as members of an unnamed namespace.

- Do not use unnamed namespaces in `.h` files.

Named Namespaces

Named namespaces should be used as follows:

- Namespaces wrap the entire source file after includes, gflags definitions/declarations, and forward declarations of classes from other namespaces:

```
// In the .h file
namespace mynamespace {

// All declarations are within the namespace scope.
// Notice the lack of indentation.
class MyClass {
 public:
  ...
  void Foo();
};

}  // namespace mynamespace
```

```
// In the .cc file
namespace mynamespace {

// Definition of functions is within scope of the namesp
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

The typical `.cc` file might have more complex detail, including the need to reference classes in other namespaces.

```
#include "a.h"

DEFINE_bool(someflag, false, "dummy flag");

class C;  // Forward declaration of class C in the globa
namespace a { class A; }  // Forward declaration of a::A

namespace b {

...code for b...         // Code goes against the left m

}  // namespace b
```

- Do not declare anything in namespace `std`, not even forward declarations of standard library classes. Declaring entities in namespace `std` is undefined behavior, i.e., not portable. To declare entities from the standard library, include the appropriate header file.

- You may not use a *using-directive* to make all names from a namespace available.

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- You may use a *using-declaration* anywhere in a `.cc` file, and in functions, methods or classes in `.h` files.

```
// OK in .cc files.
// Must be in a function, method or class in .h files.
using ::foo::bar;
```

- Namespace aliases are allowed anywhere in a `.cc` file, anywhere inside the named namespace that wraps an entire `.h` file, and in functions and methods.

```
// Shorten access to some commonly used names in .cc fil
namespace fbz = ::foo::bar::baz;

// Shorten access to some commonly used names (in a .h f
namespace librarian {
// The following alias is available to all files includi
// this header (in namespace librarian):
// alias names should therefore be chosen consistently
// within a project.
namespace pd_s = ::pipeline_diagnostics::sidetable;

inline void my_inline_function() {
  // namespace alias local to a function (or method).
  namespace fbz = ::foo::bar::baz;
  ...
}
}  // namespace librarian
```

Note that an alias in a .h file is visible to everyone #including that file, so public headers (those available outside a project) and headers transitively #included by them, should avoid defining aliases, as part of the general goal of keeping public APIs as small as possible.

- Do not use inline namespaces.

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.

Definition:

A class can define another class within it; this is also called a *member class*.

```
class Foo {

 private:
  // Bar is a member class, nested within Foo.
  class Bar {
    ...
  };

};
```

Pros:

This is useful when the nested (or member) class is only used by the enclosing class; making it a member puts it in the enclosing class scope rather than polluting the outer scope with the class name. Nested classes can be forward declared within the enclosing class and then defined in the `.cc` file to avoid including the nested class definition in the enclosing class declaration, since the nested class definition is usually only relevant to the implementation.

Cons:

Nested classes can be forward-declared only within the definition of the enclosing class. Thus, any header file manipulating a `Foo::Bar*` pointer will have to include the full class declaration for `Foo`.

Decision:

Do not make nested classes public unless they are actually part of the interface, e.g., a class that holds a set of options for some method.

Nonmember, Static Member, and Global Functions

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

Pros:

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

Cons:

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.

Decision:

Sometimes it is useful, or even necessary, 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. Rather than creating classes only to group static member functions which do not share static data, use namespaces instead.

Functions defined in the same compilation unit as production classes may introduce unnecessary coupling and link-time dependencies when directly called from other compilation units; static member functions are particularly susceptible to this. Consider extracting a new class, or placing the functions in a namespace possibly in a separate library.

If you must define a nonmember function and it is only needed in its `.cc` file, use an unnamed namespace or `static` linkage (eg `static int Foo() {...}`) to limit its scope.

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 declarat
```

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

```
vector<int> v;
v.push_back(1);  // Prefer initializing using brace initializ
v.push_back(2);
```

```
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

Static or global variables of class type are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such variables are allowed if they are `constexpr`: they have no dynamic initialization or destruction.

Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only ints, chars, floats, or pointers, or arrays/structs of POD.

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find. Therefore in addition to banning globals of class type, we do not allow static POD variables to be initialized with the result of a function, unless that function (such as getenv(), or getpid()) does not itself depend on any other globals. (This prohibition does not apply to a static variable within function scope, since its initialization order is well-defined and does not occur until control passes through its declaration.)

Likewise, global and static variables are destroyed when the program terminates, regardless of whether the termination is by returning from `main()` or by calling `exit()`. The order in which destructors are called is defined to be the reverse of the order in which the constructors were called. Since constructor order is indeterminate, so is destructor order. For example, at program-end time a static variable might have been destroyed, but code still running — perhaps in another thread — tries to access it and fails. Or the destructor for a static `string` variable might be run prior to the destructor for another variable that contains a reference to that string.

One way to alleviate the destructor problem is to terminate the program by calling `quick_exit()` instead of `exit()`. The difference is that `quick_exit()` does not invoke destructors and does not invoke any handlers that were registered by calling `atexit()`. If you have a handler that needs to run when a program terminates via `quick_exit()` (flushing logs, for example), you can register it using `at_quick_exit()`. (If you have a handler that needs to run at both `exit()` and `quick_exit()`, you need to register it in both places.)

As a result we only allow static variables to contain POD data. This rule completely disallows `vector` (use C arrays instead), or `string` (use `const char []`).

If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your main() function or from pthread_once(). Note that this must be a raw pointer, not a "smart" pointer, since the smart pointer's destructor will have the order-of-destructor issue that we are trying to avoid.

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

Avoid doing complex initialization in constructors (in particular, initialization that can fail or that requires virtual method calls).

Definition:

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

Pros:

Convenience in typing. No need to worry about whether the class has been initialized or not.

Cons:

The problems with doing work in constructors are:

- There is no easy way for constructors to signal errors, short of using exceptions (which are forbidden).
- If the work fails, we now have an object whose initialization code failed, so it may be an indeterminate state.
- 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.
- If someone creates a global variable of this type (which is against the rules, but still), the constructor code will be called before `main()`, possibly breaking some implicit assumptions in the constructor code. For instance, gflags will not yet have been initialized.

Decision:

Constructors should never call virtual functions or attempt to raise non-fatal failures. If your object requires non-trivial initialization, consider using a factory function or `Init()` method.

Initialization

If your class defines member variables, you must provide an in-class initializer for every member variable or write a constructor (which can be a default constructor). If you do not declare any constructors yourself then the compiler will generate a default constructor for you, which may leave some fields uninitialized or initialized to inappropriate values.

Definition:

The default constructor is called when we `new` a class object with no arguments. It is always called when calling `new[]` (for arrays). In-class member initialization means declaring a member variable using a construction like `int count_ = 17;` or `string name_{"abc"};`, as opposed to just `int count_;` or `string name_;`.

Pros:

A user-defined default constructor is used to initialize an object if no initializer is provided. It can ensure that an object is always in a valid and usable state as soon as it's constructed; it can also ensure that an object is initially created in an obviously "impossible" state, to aid debugging.

In-class member initialization ensures that a member variable will be initialized appropriately without having to duplicate the initialization code in multiple constructors. This can reduce bugs where you add a new member variable, initialize it in one constructor, and forget to put that initialization code in another constructor.

Cons:

Explicitly defining a default constructor is extra work for you, the code writer.

In-class member initialization is potentially confusing if a member variable is initialized as part of its declaration and also initialized in a constructor, since the value in the constructor will override the value in the declaration.

Decision:

Use in-class member initialization for simple initializations, especially when a member variable must be initialized the same way in more than one constructor.

If your class defines member variables that aren't initialized in-class, and if it has no other constructors, you must define a default constructor (one that takes no arguments). It should preferably initialize the object in such a way that its internal state is consistent and valid.

The reason for this is that if you have no other constructors and do not define a default constructor, the compiler will generate one for you. This compiler generated constructor may not initialize your object sensibly.

If your class inherits from an existing class but you add no new member variables, you are not required to have a default constructor.

Explicit Constructors

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

Definition:

Normally, if a constructor can be called with one argument, it can be used as a conversion. For instance, if you define `Foo::Foo(string name)` and then pass a string to a function that expects a `Foo`, the constructor will be called to convert the string into a `Foo` and will pass the `Foo` to your function for you. This can be convenient but is also a source of trouble when things get converted and new objects created without you meaning them to. Declaring a constructor `explicit` prevents it from being invoked implicitly as a conversion.

In addition to single-parameter constructors, this also applies to constructors where every parameter after the first has a default value, e.g., `Foo::Foo(string name, int id = 42)`.

Pros:

Avoids undesirable conversions.

Cons:

None.

Decision:

We require all constructors that are callable with a single argument to be explicit. Always put `explicit` in front of such constructors in the class definition: `explicit Foo(string name);`

Copy and move constructors are exceptions: they should not be `explicit`. Classes that are intended to be transparent wrappers around other classes are also exceptions. Such exceptions should be clearly marked with comments.

Finally, constructors that take only a `std::initializer_list` may be non-explicit. This permits construction of your type from a braced initializer list, as in an assignment-style initialization, function argument, or return statement. For example:

```
MyType m = {1, 2};
MyType MakeMyType() { return {1, 2}; }
TakeMyType({1, 2});
```

Copyable and Movable Types

Support copying and/or moving if it makes sense for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.

Definition:

A copyable type allows its objects to be initialized or assigned from any other object of the same type, without changing the value of the source. For user-defined types, the copy behavior is defined by the copy constructor and the copy-assignment operator. `string` is an example of a copyable type.

A movable type is one that can be initialized and assigned from temporaries (all copyable types are therefore movable). `std::unique_ptr<int>` is an example of a movable but not copyable type. For user-defined types, the move behavior is defined by the move constructor and the move-assignment operator.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g. when passing objects by value.

Pros:

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing pointers or references, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand and maintain. Such objects can be used with generic APIs that require pass-by-value, such as most containers.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like `Clone()`, `CopyFrom()` or `Swap()`, because they can be generated by the compiler, either implicitly or with `= default`. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialization and assignment steps, and they're eligible for optimizations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

Cons:

Many types do not need to be copyable, and providing copy operations for them can be confusing, nonsensical, or outright incorrect. Copy/assigment operations for base class types are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion, particularly for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

Decision:

Make your type copyable/movable if it will be useful, and if it makes sense in the context of the rest of the API. As a rule of thumb, if the behavior (including computational complexity) of a copy isn't immediately obvious to users of your type, your type shouldn't be copyable. If you choose to make it copyable, define both copy operations (constructor and assignment). If your type is copyable and a move operation is more efficient than a copy, define both move operations (constructor and assignment). If your type is not copyable, but the correctness of a move is obvious to users of the type and its fields support it, you may make the type move-only by defining both of the move operations.

Prefer to define copy and move operations with `= default`. Defining non-default move operations currently requires a style exception. Remember to review the correctness of any defaulted operations as you would any other code.

Due to the risk of slicing, avoid providing an assignment operator or public copy/move constructor for a class that's intended to be derived from (and avoid deriving from a class with such members). If your base class needs to be copyable, provide a public virtual `Clone()` method, and a protected copy constructor that derived classes can use to implement it.

If you do not want to support copy/move operations on your type, explicitly disable them using `= delete` or whatever other mechanism your project uses.

Delegating and Inheriting Constructors

Use delegating and inheriting constructors when they reduce code duplication.

Definition:

Delegating and inheriting constructors are two different features, both introduced in C++11, for reducing code duplication in constructors. Delegating constructors allow one of a class's constructors to forward work to one of the class's other constructors, using a special variant of the initialization list syntax. For example:

```
X::X(const string& name) : name_(name) {
  ...
}

X::X() : X("") { }
```

Inheriting constructors allow a derived class to have its base class's constructors available directly, just as with any of the base class's other member functions, instead of having to redeclare them. This is especially useful if the base has multiple constructors. For example:

```
class Base {
 public:
  Base();
  Base(int n);
  Base(const string& s);
  ...
};

class Derived : public Base {
 public:
  using Base::Base;  // Base's constructors are redeclared he
};
```

This is especially useful when `Derived`'s constructors don't have to do anything more than calling `Base`'s constructors.

Pros:

Delegating and inheriting constructors reduce verbosity and boilerplate, which can improve readability.

Delegating constructors are familiar to Java programmers.

Cons:

It's possible to approximate the behavior of delegating constructors by using a helper function.

Inheriting constructors may be confusing if a derived class introduces new member variables, since the base class constructor doesn't know about them.

Decision:

Use delegating and inheriting constructors when they reduce boilerplate and improve readability. Be cautious about inheriting constructors when your derived class has new member variables. Inheriting constructors may still be appropriate in that case if you can use in-class member initialization for the derived class's member variables.

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. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, `Initialize()`, `Reset()`, `Validate()`.

If more functionality is 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 functors and traits.

Note that member variables in structs and classes have different naming rules.

Inheritance

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

Definition:

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the parent base class defines. In practice, inheritance is used in two major ways in C++: implementation inheritance, in which actual code is inherited by the child, and interface inheritance, in which only method names are inherited.

Pros:

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.

Cons:

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. The base class may also define some data members, so that specifies physical layout of the base class.

Decision:

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`.

Make your destructor `virtual` if necessary. If your class has virtual methods, its destructor should be virtual.

Limit the use of `protected` to those member functions that might need to be accessed from subclasses. Note that data members should be private.

Explicitly annotate overrides of virtual functions or virtual destructors with an `override` or (less frequently) `final` specifier. Older (pre-C++11) code will use the `virtual` keyword as an inferior alternative annotation. For clarity, use exactly one of `override`, `final`, or `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

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.

Definition:

Multiple inheritance allows a sub-class to have more than one base class. We distinguish between base classes that are *pure interfaces* and those that have an *implementation*.

Pros:

Multiple implementation inheritance may let you re-use even more code than single inheritance (see Inheritance).

Cons:

Only very rarely is multiple *implementation* inheritance actually useful. When multiple implementation inheritance seems like the solution, you can usually find a different, more explicit, and cleaner solution.

Decision:

Multiple inheritance is allowed only when all superclasses, with the possible exception of the first one, are pure interfaces. In order to ensure that they remain pure interfaces, they must end with the `Interface` suffix.

Note:

There is an exception to this rule on Windows.

Interfaces

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

Definition:

A class is a pure interface if it meets the following requirements:

- It has only public pure virtual ("`= 0`") methods and static methods (but see below for destructor).
- It may not have non-static data members.
- It need not have any constructors defined. If a constructor is provided, it must take no arguments and it must be protected.
- If it is a subclass, it may only be derived from classes that satisfy these conditions and are tagged with the `Interface` suffix.

An interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, the interface must also declare a virtual destructor (in an exception to the first rule, this should not be pure). See Stroustrup, *The C++*

Programming Language, 3rd edition, section 12.4 for details.

Pros:

Tagging a class with the `Interface` suffix lets others know that they must not add implemented methods or non static data members. This is particularly important in the case of multiple inheritance. Additionally, the interface concept is already well-understood by Java programmers.

Cons:

The `Interface` suffix lengthens the class name, which can make it harder to read and understand. Also, the interface property may be considered an implementation detail that shouldn't be exposed to clients.

Decision:

A class may end with `Interface` only if it meets the above requirements. We do not require the converse, however: classes that meet the above requirements are not required to end with `Interface`.

Operator Overloading

Do not overload operators except in rare, special circumstances. Do not create user-defined literals.

Definition:

A class can define that operators such as `+` and `/` operate on the class as if it were a built-in type. An overload of `operator""` allows the built-in literal syntax to be used to create objects of class types.

Pros:

Operator overloading can make code appear more intuitive because a class will behave in the same way as built-in types (such as `int`). Overloaded operators are more playful names for functions that are less-colorfully named, such as `Equals()` or `Add()`.

For some template functions to work correctly, you may need to define operators.

User-defined literals are a very concise notation for creating objects of user-defined types.

Cons:

While operator overloading can make code more intuitive, it has several drawbacks:

- It can fool our intuition into thinking that expensive operations are cheap, built-in operations.
- It is much harder to find the call sites for overloaded operators. Searching for `Equals()` is much easier than searching for relevant invocations of `==`.
- Some operators work on pointers too, making it easy to introduce bugs. `Foo + 4` may do one thing, while `&Foo + 4` does something totally different. The compiler does not complain for either of these, making this very hard to debug.
- User-defined literals allow creating new syntactic forms that are unfamiliar even to experienced C++ programmers.

Overloading also has surprising ramifications. For instance, if a class overloads unary `operator&`, it cannot safely be forward-declared.

Decision:

In general, do not overload operators. You can define ordinary functions like

`Equals()` if you need them. Likewise, avoid the dangerous unary `operator&` at all costs, if there's any possibility the class might be forward-declared.

Do not overload `operator""`, i.e. do not introduce user-defined literals.

However, there may be rare cases where you need to overload an operator to interoperate with templates or "standard" C++ classes (such as `operator<<(ostream&, const T&)` for logging). These are acceptable if fully justified, but you should try to avoid these whenever possible. In particular, do not overload `operator==` or `operator<` just so that your class can be used as a key in an STL container; instead, you should create equality and comparison functor types when declaring the container.

Some of the STL algorithms do require you to overload `operator==`, and you may do so in these cases, provided you document why.

See also Copyable and Movable Types and Function Overloading.

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`.

The definitions of accessors are usually inlined in the header file.

See also Inheritance and Function Names.

Declaration Order

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

Your class definition should start with its `public:` section, followed by its `protected:` section and then its `private:` section. If any of these sections are empty, omit them.

Within each section, the declarations generally should be in the following order:

- Typedefs and Enums
- Constants (`static const` data members)
- Constructors
- Destructor
- Methods, including static methods
- Data Members (except `static const` data members)

Friend declarations should always be in the private section. If copying and assignment are disabled with a macro such as `DISALLOW_COPY_AND_ASSIGN`, it should be at the end of the `private:` section, and should be the last thing in the class. See Copyable and Movable Types.

Method definitions in the corresponding `.cc` file should be the same as the declaration order, as much as possible.

Do not put large method definitions inline in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined inline. See Inline

Functions for more details.

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.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

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.

Ownership and Smart Pointers

Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

Definition:

"Ownership" is a bookkeeping technique for managing dynamically allocated memory (and other resources). The owner of a dynamically allocated object is an object or function that is responsible for ensuring that it is deleted when no longer needed. Ownership can sometimes be shared, in which case the last owner is typically responsible for deleting it. Even when ownership is not shared, it can be transferred from one piece of code to another.

"Smart" pointers are classes that act like pointers, e.g. by overloading the `*` and `->` operators. Some smart pointer types can be used to automate ownership bookkeeping, to ensure these responsibilities are met. `std::unique_ptr` is a smart pointer type introduced in C++11, which expresses exclusive ownership of a dynamically allocated object; the object is deleted when the `std::unique_ptr` goes out of scope. It cannot be copied, but can be *moved* to represent ownership transfer. `std::shared_ptr` is a smart pointer type that expresses shared ownership of a dynamically allocated object. `std::shared_ptr`s can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed.

Pros:

- It's virtually impossible to manage dynamically allocated memory without some sort of ownership logic.
- Transferring ownership of an object can be cheaper than copying it (if copying it is even possible).
- Transferring ownership can be simpler than 'borrowing' a pointer or reference, because it reduces the need to coordinate the lifetime of the object between the two users.
- Smart pointers can improve readability by making ownership logic explicit, self-documenting, and unambiguous.
- Smart pointers can eliminate manual ownership bookkeeping, simplifying the code and ruling out large classes of errors.
- For const objects, shared ownership can be a simple and efficient alternative to deep copying.

Cons:

- Ownership must be represented and transferred via pointers (whether smart or plain). Pointer semantics are more complicated than value semantics, especially in APIs: you have to worry not just about ownership, but also aliasing, lifetime, and mutability, among other issues.
- The performance costs of value semantics are often overestimated, so the performance benefits of ownership transfer might not justify the readability and complexity costs.
- APIs that transfer ownership force their clients into a single memory management model.
- Code using smart pointers is less explicit about where the resource releases take place.
- `std::unique_ptr` expresses ownership transfer using C++11's move semantics, which are relatively new and may confuse some programmers.
- Shared ownership can be a tempting alternative to careful ownership design, obfuscating the design of a system.
- Shared ownership requires explicit bookkeeping at run-time, which can be costly.
- In some cases (e.g. cyclic references), objects with shared ownership may never be deleted.
- Smart pointers are not perfect substitutes for plain pointers.

Decision:

If dynamic allocation is necessary, prefer to keep ownership with the code that allocated it. If other code needs access to the object, consider passing it a copy, or passing a pointer or reference without transferring ownership. Prefer to use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Do not design your code to use shared ownership without a very good reason. One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e. `std::shared_ptr<const Foo>`). If you do use shared ownership, prefer to use `std::shared_ptr`.

Do not use `scoped_ptr` in new code unless you need to be compatible with older versions of C++. Never use `std::auto_ptr`. Instead, use `std::unique_ptr`.

cpplint

Use `cpplint.py` to detect style errors.

`cpplint.py` is a tool that reads a source file and identifies many style errors. It is not perfect, and has both false positives and false negatives, but it is still a valuable tool. False positives can be ignored by putting `// NOLINT` at the end of the line or `// NOLINTNEXTLINE` in the previous line.

Some projects have instructions on how to run `cpplint.py` from their project tools. If the project you are contributing to does not, you can download `cpplint.py` separately.

Other C++ Features

Reference Arguments

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

Definition:

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)`.

Pros:

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.

Cons:

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

Decision:

Within function parameter lists all references must be `const`:

```
void Foo(const string &in, 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.

Rvalue References

Use rvalue references only to define move constructors and move assignment operators. Do not use `std::forward`.

Definition:

Rvalue references are a type of reference that can only bind to temporary objects. The syntax is similar to traditional reference syntax. For example, `void f(string&& s);` declares a function whose argument is an rvalue reference to a string.

Pros:

- Defining a move constructor (a constructor taking an rvalue reference to the class type) makes it possible to move a value instead of copying it. If `v1` is a `vector<string>`, for example, then `auto v2(std::move(v1))` will probably just result in some simple pointer manipulation instead of copying a large amount of data. In some cases this can result in a major performance improvement.
- Rvalue references make it possible to write a generic function wrapper that forwards its arguments to another function, and works whether or not its arguments are temporary objects.
- Rvalue references make it possible to implement types that are movable but not copyable, which can be useful for types that have no sensible definition of copying but where you might still want to pass them as function arguments, put them in containers, etc.
- `std::move` is necessary to make effective use of some standard-library types, such as `std::unique_ptr`.

Cons:

- Rvalue references are a relatively new feature (introduced as part of C++11), and not yet widely understood. Rules like reference collapsing, and automatic synthesis of move constructors, are complicated.

Decision:

Use rvalue references only to define move constructors and move assignment operators, as described in Copyable and Movable Types. Do not use `std::forward` utility function. You may use `std::move` to express moving a value from one object to another rather than copying it.

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.

Definition:

You may write a function that takes a `const string&` and overload it with another that takes `const char*`.

```
class MyClass {
 public:
  void Analyze(const string &text);
  void Analyze(const char *text, size_t textlen);
};
```

Pros:

Overloading can make code more intuitive by allowing an identically-named function to take different arguments. It may be necessary for templatized code, and it can be convenient for Visitors.

Cons:

If a function is overloaded by the argument types alone, a reader may have to understand C++'s complex matching rules in order to tell what's going on. Also many people are confused by the semantics of inheritance if a derived class overrides only some of the variants of a function.

Decision:

If you want to overload a function, consider qualifying the name with some information about the arguments, e.g., `AppendString()`, `AppendInt()` rather than just `Append()`.

Default Arguments

We do not allow default function parameters, except in limited situations as explained below. Simulate them with function overloading instead, if appropriate.

Pros:

Often you have a function that uses default values, but occasionally you want to override the defaults. Default parameters allow an easy way to do this without having to define many functions for the rare exceptions. Compared to overloading the function, default arguments have a cleaner syntax, with less boilerplate and a clearer distinction between 'required' and 'optional' arguments.

Cons:

Function pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding a default argument to an existing function changes its type, which can cause problems with code taking its address. Adding function overloads avoids these problems. In addition, default parameters may result in bulkier code since they are replicated at every call-site -- as opposed to overloaded functions, where "the default" appears only in the function definition.

Decision:

While the cons above are not that onerous, they still outweigh the (small) benefits of default arguments over function overloading. So except as described below, we require all arguments to be explicitly specified.

One specific exception is when the function is a static function (or in an unnamed namespace) in a .cc file. In this case, the cons don't apply since the function's use is so localized.

In addition, default function parameters are allowed in constructors. Most of the cons listed above don't apply to constructors because it's impossible to take their address.

Another specific exception is when default arguments are used to simulate variable-length argument lists.

```
// Support up to 4 params by using a default empty AlphaNum.
string StrCat(const AlphaNum &a,
              const AlphaNum &b = gEmptyAlphaNum,
              const AlphaNum &c = gEmptyAlphaNum,
              const AlphaNum &d = gEmptyAlphaNum);
```

Variable-Length Arrays and alloca()

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

Pros:

Variable-length arrays have natural-looking syntax. Both variable-length arrays and `alloca()` are very efficient.

Cons:

Variable-length arrays and alloca are not part of Standard C++. More importantly, they allocate a data-dependent amount of stack space that can trigger difficult-to-find memory overwriting bugs: "It ran fine on my machine, but dies mysteriously in production".

Decision:

Use a safe allocator instead, such as `std::vector` or `std::unique_ptr<T[]>`.

Friends

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

Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class. A common use of `friend` is to have a `FooBuilder` class be a friend of `Foo` so that it can construct the inner state of `Foo` correctly, without exposing this state to the world. In some cases it may be useful to make a unittest class a friend of the class it tests.

Friends extend, but do not break, the encapsulation boundary of a class. In some cases this is better than making a member public when you want to give only one other class access to it. However, most classes should interact with other classes solely through their public members.

Exceptions

We do not use C++ exceptions.

Pros:

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an `Init()` method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons:

- When you add a `throw` statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not

clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

Decision:

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

This prohibition also applies to the exception-related features added in C++11, such as `noexcept`, `std::exception_ptr`, and `std::nested_exception`.

There is an exception to this rule (no pun intended) for Windows code.

Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI).

Definition:

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of `typeid` or `dynamic_cast`.

Cons:

Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.

Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

Pros:

The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.

RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.

RTTI is useful when considering multiple abstract objects. Consider

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == NULL)
    return false;
  ...
}
```

Decision:

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

- Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
- If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object. Usually one can use a `static_cast` as an alternative in such situations.

Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.

Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

Casting

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

Definition:

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

Pros:

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"`); C++ casts avoid this. Additionally C++ casts are more visible when searching for them.

Cons:

The syntax is nasty.

Decision:

Do not use C-style casts. Instead, use these C++-style casts.

- Use `static_cast` as the equivalent of a C-style cast that does value conversion, or when you need to explicitly up-cast a pointer from a class to its superclass.
- Use `const_cast` to remove the `const` qualifier (see const).
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

See the RTTI section for guidance on the use of `dynamic_cast`.

Streams

Use streams only for logging.

Definition:

Streams are a replacement for `printf()` and `scanf()`.

Pros:

With streams, you do not need to know the type of the object you are printing. You do not have problems with format strings not matching the argument list. (Though with gcc, you do not have that problem with `printf` either.) Streams have automatic constructors and destructors that open and close the relevant files.

Cons:

Streams make it difficult to do functionality like `pread()`. Some formatting (particularly the common format string idiom `%.*s`) is difficult if not impossible to do efficiently using streams without using `printf`-like hacks. Streams do not support operator reordering (the `%1$s` directive), which is helpful for internationalization.

Decision:

Do not use streams, except where required by a logging interface. Use `printf`-like routines instead.

There are various pros and cons to using streams, but in this case, as in many other cases, consistency trumps the debate. Do not use streams in your code.

Extended Discussion

There has been debate on this issue, so this explains the reasoning in greater depth. Recall the Only One Way guiding principle: we want to make sure that whenever we do a certain type of I/O, the code looks the same in all those places. Because of this, we do not want to allow users to decide between using streams or using `printf` plus Read/Write/etc. Instead, we should settle on one or the other. We made an exception for logging because it is a pretty specialized application, and for historical reasons.

Proponents of streams have argued that streams are the obvious choice of the two, but the issue is not actually so clear. For every advantage of streams they point out, there is an equivalent disadvantage. The biggest advantage is that you do not need to know the type of the object to be printing. This is a fair point. But, there is a downside: you can easily use the wrong type, and the compiler will not warn you. It is easy to make this kind of mistake without knowing when using streams.

```
cout << this;   // Prints the address
cout << *this;  // Prints the contents
```

The compiler does not generate an error because << has been overloaded. We discourage overloading for just this reason.

Some say `printf` formatting is ugly and hard to read, but streams are often no better. Consider the following two fragments, both with the same typo. Which is easier to discover?

```
cerr << "Error connecting to '" << foo->bar()->hostname.first
     << ":" << foo->bar()->hostname.second << ": " << strerro

fprintf(stderr, "Error connecting to '%s:%u: %s",
        foo->bar()->hostname.first, foo->bar()->hostname.seco
        strerror(errno));
```

And so on and so forth for any issue you might bring up. (You could argue, "Things would be better with the right wrappers," but if it is true for one scheme, is it not also true for the other? Also, remember the goal is to make the language smaller, not add yet more machinery that someone has to learn.)

Either path would yield different advantages and disadvantages, and there is not a clearly superior solution. The simplicity doctrine mandates we settle on one of them though, and the majority decision was on `printf` + `read`/`write`.

Preincrement and Predecrement

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

Definition:

When a variable is incremented (`++i` or `i++`) or decremented (`--i` or `i--`) and the value of the expression is not used, one must decide whether to preincrement (decrement) or postincrement (decrement).

Pros:

When the return value is ignored, the "pre" form (`++i`) is never less efficient than the "post" form (`i++`), and is often more efficient. This is because post-increment (or decrement) requires a copy of `i` to be made, which is the value of the expression. If `i` is an iterator or other non-scalar type, copying `i` could be expensive. Since the two types of increment behave the same when the value is ignored, why not just always pre-increment?

Cons:

The tradition developed, in C, of using post-increment when the expression value is not used, especially in `for` loops. Some find post-increment easier to read, since the "subject" (`i`) precedes the "verb" (`++`), just like in English.

Decision:

For simple scalar (non-object) values there is no reason to prefer one form and we allow either. For iterators and other template types, use pre-increment.

Use of const

Use `const` whenever it makes sense. With C++11, `constexpr` is a better choice for some uses of const.

Definition:

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; };`).

Pros:

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.

Cons:

`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.

Decision:

`const` variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. Therefore we strongly recommend that you use `const` whenever it makes sense to do so:

- If a function does not modify an argument passed by reference or by pointer, that argument should be `const`.
- Declare methods to be `const` whenever possible. Accessors should almost always be `const`. Other methods should be const if they do not modify any data members, do not call any non-`const` methods, and do not return a non-`const` pointer or non-`const` reference to a data member.
- Consider making data members `const` whenever they do not need to be modified after construction.

The `mutable` keyword is allowed but is unsafe when used with threads, so thread safety should be carefully considered first.

Where to put the const

Some people favor the form `int const *foo` to `const int* foo`. They argue that this is more readable because it's more consistent: it keeps the rule that `const` always follows the object it's describing. However, this consistency argument doesn't apply in codebases with few deeply-nested pointer expressions since most `const` expressions have only one `const`, and it applies to the underlying value. In such cases, there's no consistency to maintain. Putting the `const` first is arguably more readable, since it follows English in putting the "adjective" (`const`) before the "noun" (`int`).

That said, while we encourage putting `const` first, we do not require it. But be consistent with the code around you!

Use of constexpr

In C++11, use `constexpr` to define true constants or to ensure constant initialization.

Definition:

Some variables can be declared `constexpr` to indicate the variables are true constants, i.e. fixed at compilation/link time. Some functions and constructors can be declared `constexpr` which enables them to be used in defining a `constexpr` variable.

Pros:

Use of `constexpr` enables definition of constants with floating-point expressions rather than just literals; definition of constants of user-defined types; and definition of constants with function calls.

Cons:

Prematurely marking something as constexpr may cause migration problems if later on it has to be downgraded. Current restrictions on what is allowed in constexpr functions and constructors may invite obscure workarounds in these definitions.

Decision:

`constexpr` definitions enable a more robust specification of the constant parts of an interface. Use `constexpr` to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with `constexpr`. Do not use `constexpr` to force inlining.

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`. If your variable represents a value that could ever be greater than or equal to 2^31 (2GiB), use a 64-bit type such as `int64_t`. Keep in mind that even if your value won't ever be too large for an `int`, it may be used in intermediate calculations which may require a larger type. When in doubt, choose a larger type.

Definition:

C++ does not specify the sizes of its integer types. Typically people assume that `short` is 16 bits, `int` is 32 bits, `long` is 32 bits and `long long` is 64 bits.

Pros:

Uniformity of declaration.

Cons:

The sizes of integral types in C++ can vary based on compiler and architecture.

Decision:

`<stdint.h>` defines types like `int16_t`, `uint32_t`, `int64_t`, etc. You should always use those in preference to `short`, `unsigned long long` and the like, when you need a guarantee on the size of an integer. Of the C integer types, only `int` should be used. When appropriate, you are welcome to use standard types like `size_t` and `ptrdiff_t`.

We use `int` very often, for integers we know are not going to be too big, e.g., loop counters. Use plain old `int` for such things. You should assume that an `int` is at least 32 bits, but don't assume that it has more than 32 bits. If you need a 64-bit integer type, use `int64_t` or `uint64_t`.

For integers we know can be "big", use `int64_t`.

You should not use the unsigned integer types such as `uint32_t`, unless there is a valid reason such as representing a bit pattern rather than a number, or you need defined overflow modulo 2^N. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.

If your code is a container that returns a size, be sure to use a type that will accommodate any possible usage of your container. When in doubt, use a larger type rather than a smaller type.

Use care when converting integer types. Integer conversions and promotions can cause non-intuitive behavior.

On Unsigned Integers

Some people, including some textbook authors, recommend using unsigned types to represent numbers that are never negative. This is intended as a form of self-documentation. However, in C, the advantages of such documentation are outweighed by the real bugs it can introduce. Consider:

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

This code will never terminate! Sometimes gcc will notice this bug and warn you, but often it will not. Equally bad bugs can occur when comparing signed and unsigned variables. Basically, C's type-promotion scheme causes unsigned types to behave differently than one might expect.

So, document that a variable is non-negative using assertions. Don't use an unsigned type.

64-bit Portability

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

- `printf()` specifiers for some types are not cleanly portable between 32-bit and 64-bit systems. C99 defines some portable format specifiers. Unfortunately, MSVC 7.1 does not understand some of these specifiers and the standard is missing a few, so we have to define our own ugly versions in some cases (in the style of the standard include file `inttypes.h`):

```
// printf macros for size_t, in the style of inttypes.h
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
#define __PRIS_PREFIX
#endif

// Use these macros after a % in a printf format string
// to get correct 32/64 bit behavior, like this:
// size_t size = records.size();
// printf("%"PRIuS"\n", size);

#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

Type	DO NOT use	DO use	Notes
`void *` (or any pointer)	`%lx`	`%p`	
`int64_t`	`%qd`, `%lld`	`%"PRId64"`	
`uint64_t`	`%qu`, `%llu`, `%llx`	`%"PRIu64"`, `%"PRIx64"`	
`size_t`	`%u`	`%"PRIuS"`, `%"PRIxS"`	C99 specifies `%zu`
`ptrdiff_t`	`%d`	`%"PRIdS"`	C99 specifies `%td`

Note that the `PRI*` macros expand to independent strings which are concatenated by the compiler. Hence if you are using a non-constant format string, you need to insert the value of the macro into the format, rather than the name. It is still possible, as usual, to include length specifiers, etc., after the `%` when using the `PRI*` macros. So, e.g. `printf("x = %30"PRIuS"\n", x)` would expand on 32-bit Linux to `printf("x = %30" "u" "\n", x)`, which the compiler will treat as `printf("x = %30u\n", x)`.

- Remember that `sizeof(void *)` != `sizeof(int)`. Use `intptr_t` if you want a pointer-sized integer.
- You may need to be careful with structure alignments, particularly for structures being stored on disk. Any class/structure with a `int64_t`/`uint64_t` member will by default end up being 8-byte aligned on a 64-bit system. If you have such structures being shared on disk between 32-bit and 64-bit code, you will need to ensure that they are packed the same on both architectures. Most compilers offer a way to alter structure alignment. For gcc, you can use `__attribute__((packed))`. MSVC offers `#pragma pack()` and `__declspec(align())`.
- Use the `LL` or `ULL` suffixes as needed to create 64-bit constants. For example:

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

- If you really need different code on 32-bit and 64-bit systems, use `#ifdef _LP64` to choose between the code variants. (But please avoid this if possible, and keep any such changes localized.)

Preprocessor Macros

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

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behavior, especially since macros have global scope.

Luckily, macros are not nearly as necessary in C++ as they are in C. Instead of using a macro to inline performance-critical code, use an inline function. Instead of using a macro to store a constant, use a `const` variable. Instead of using a macro to "abbreviate" a long variable name, use a reference. Instead of using a macro to conditionally compile code ... well, don't do that at all (except, of course, for the `#define` guards to prevent double inclusion of header files). It makes testing much more difficult.

Macros can do things these other techniques cannot, and you do see them in the codebase, especially in the lower-level libraries. And some of their special features (like stringifying, concatenation, and so forth) are not available through the language proper. But before using a macro, consider carefully whether there's a non-macro way to achieve the same result.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a `.h` file.
- `#define` macros right before you use them, and `#undef` them right after.
- Do not just `#undef` an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
- Prefer not using ## to generate function/class/variable names.

0 and nullptr/NULL

Use `0` for integers, `0.0` for reals, `nullptr` (or `NULL`) for pointers, and `'\0'` for chars.

Use `0` for integers and `0.0` for reals. This is not controversial.

For pointers (address values), there is a choice between `0`, `NULL`, and `nullptr`. For projects that allow C++11 features, use `nullptr`. For C++03 projects, we prefer `NULL` because it looks like a pointer. In fact, some C++ compilers provide special definitions of `NULL` which enable them to give useful warnings, particularly in situations where `sizeof(NULL)` is not equal to `sizeof(0)`.

Use `'\0'` for chars. This is the correct type and also makes code more readable.

sizeof

Prefer `sizeof(`*`varname`*`)` to `sizeof(`*`type`*`)`.

Use `sizeof(`*`varname`*`)` when you take the size of a particular variable. `sizeof(`*`varname`*`)` will update appropriately if someone changes the variable type either now or later. You may use `sizeof(type)` for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

```
Struct data;
memset(&data, 0, sizeof(data));
```

```
memset(&data, 0, sizeof(Struct));
```

```
if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count:
  return false;
}
```

auto

Use `auto` to avoid type names that are just clutter. Continue to use manifest type declarations when it helps readability, and never use `auto` for anything but local variables.

Definition:

In C++11, a variable whose type is given as `auto` will be given a type that matches that of the expression used to initialize it. You can use `auto` either to initialize a variable by copying, or to bind a reference.

```
vector<string> v;
...
auto s1 = v[0];  // Makes a copy of v[0].
const auto& s2 = v[0];  // s2 is a reference to v[0].
```

Pros:

C++ type names can sometimes be long and cumbersome, especially when they involve templates or namespaces. In a statement like:

```
sparse_hash_map<string, int>::iterator iter = m.find(val);
```

the return type is hard to read, and obscures the primary purpose of the statement. Changing it to:

```
auto iter = m.find(val);
```

makes it more readable.

Without `auto` we are sometimes forced to write a type name twice in the same expression, adding no value for the reader, as in:

```
diagnostics::ErrorStatus* status = new diagnostics::ErrorStat
```

Using `auto` makes it easier to use intermediate variables when appropriate, by reducing the burden of writing their types explicitly.

Cons:

Sometimes code is clearer when types are manifest, especially when a variable's initialization depends on things that were declared far away. In an expression like:

```
auto i = x.Lookup(key);
```

it may not be obvious what `i`'s type is, if `x` was declared hundreds of lines earlier.

Programmers have to understand the difference between `auto` and `const auto&` or they'll get copies when they didn't mean to.

The interaction between `auto` and C++11 brace-initialization can be confusing. The declarations:

```
auto x(3);  // Note: parentheses.
auto y{3};  // Note: curly braces.
```

mean different things — `x` is an `int`, while `y` is a `std::initializer_list<int>`. The same applies to other normally-invisible proxy types.

If an `auto` variable is used as part of an interface, e.g. as a constant in a header, then a programmer might change its type while only intending to change its value, leading to a more radical API change than intended.

Decision:

`auto` is permitted, for local variables only. Do not use `auto` for file-scope or namespace-scope variables, or for class members. Never initialize an `auto`-typed variable with a braced initializer list.

The `auto` keyword is also used in an unrelated C++11 feature: it's part of the syntax for a new kind of function declaration with a trailing return type. Trailing return types are permitted only in lambda expressions.

Braced Initializer List

You may use braced initializer lists.

In C++03, aggregate types (arrays and structs with no constructor) could be initialized with braced initializer lists.

```
struct Point { int x; int y; };
Point p = {1, 2};
```

In C++11, this syntax was generalized, and any object type can now be created with a braced initializer list, known as a *braced-init-list* in the C++ grammar. Here are a few examples of its use.

```
// Vector takes a braced-init-list of elements.
vector<string> v{"foo", "bar"};

// Basically the same, ignoring some small technicalities.
// You may choose to use either form.
vector<string> v = {"foo", "bar"};

// Usable with 'new' expressions.
auto p = new vector<string>{"foo", "bar"};

// A map can take a list of pairs. Nested braced-init-lists v
map<int, string> m = {{1, "one"}, {2, "2"}};

// A braced-init-list can be implicitly converted to a returr
vector<int> test_function() { return {1, 2, 3}; }
```

```
// Iterate over a braced-init-list.
for (int i : {-1, -2, -3}) {}

// Call a function using a braced-init-list.
void TestFunction2(vector<int> v) {}
TestFunction2({1, 2, 3});
```

A user-defined type can also define a constructor and/or assignment operator that take `std::initializer_list<T>`, which is automatically created from *braced-init-list*:

```
class MyType {
 public:
  // std::initializer_list references the underlying init lis
  // It should be passed by value.
  MyType(std::initializer_list<int> init_list) {
    for (int i : init_list) append(i);
  }
  MyType& operator=(std::initializer_list<int> init_list) {
    clear();
    for (int i : init_list) append(i);
  }
};
MyType m{2, 3, 5, 7};
```

Finally, brace initialization can also call ordinary constructors of data types, even if they do not have `std::initializer_list<T>` constructors.

```
double d{1.23};
// Calls ordinary constructor as long as MyOtherType has no
// std::initializer_list constructor.
class MyOtherType {
 public:
  explicit MyOtherType(string);
  MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// If the constructor is explicit, you can't use the "= {}" f
MyOtherType m{"b"};
```

Never assign a *braced-init-list* to an auto local variable. In the single element case, what this means can be confusing.

```
auto d = {1.23};        // d is a std::initializer_list<doubl
```

```
auto d = double{1.23};  // Good -- d is a double, not a std::
```

See Braced_Initializer_List_Format for formatting.

Lambda expressions

Use lambda expressions where appropriate. Do not use default lambda captures; write all captures explicitly.

Definition:

Lambda expressions are a concise way of creating anonymous function objects. They're often useful when passing functions as arguments. For example:

```
std::sort(v.begin(), v.end(), [](int x, int y) {
  return Weight(x) < Weight(y);
});
```

Lambdas were introduced in C++11 along with a set of utilities for working with function objects, such as the polymorphic wrapper `std::function`.

Pros:

- Lambdas are much more concise than other ways of defining function objects to be passed to STL algorithms, which can be a readability improvement.
- Lambdas, `std::function`, and `std::bind` can be used in combination as a general purpose callback mechanism; they make it easy to write functions that take bound functions as arguments.

Cons:

- Variable capture in lambdas can be tricky, and might be a new source of dangling-pointer bugs.
- It's possible for use of lambdas to get out of hand; very long nested anonymous functions can make code harder to understand.

Decision:

- Use lambda expressions where appropriate, with formatting as described below.
- Do not use default captures; write all lambda captures explicitly. For example, instead of `[=](int x) { return x + n; }` you should write `[n](int x) { return x + n; }` so that readers can see immediately that `n` is being captured (by value).
- Keep unnamed lambdas short. If a lambda body is more than maybe five lines long, prefer to give the lambda a name, or to use a named function instead of a lambda.
- Specify the return type of the lambda explicitly if that will make it more obvious to readers, as with `auto`.

Template metaprogramming

Avoid complicated template programming.

Definition:

Template metaprogramming refers to a family of techniques that exploit the fact that the C++ template instantiation mechanism is Turing complete and can be used to perform arbitrary compile-time computation in the type domain.

Pros:

Template metaprogramming allows extremely flexible interfaces that are type safe and high performance. Facilities like Google Test, `std::tuple`, `std::function`, and Boost.Spirit would be impossible without it.

Cons:

The techniques used in template metaprogramming are often obscure to anyone but language experts. Code that uses templates in complicated ways is often unreadable, and is hard to debug or maintain.

Template metaprogramming often leads to extremely poor compiler time error messages: even if an interface is simple, the complicated implementation details become visible when the user does something wrong.

Template metaprogramming interferes with large scale refactoring by making the job of refactoring tools harder. First, the template code is expanded in multiple contexts, and it's hard to verify that the transformation makes sense in all of them. Second, some refactoring tools work with an AST that only represents the structure of the code after template expansion. It can be difficult to automatically work back to the original source construct that needs to be rewritten.

Decision:

Template metaprogramming sometimes allows cleaner and easier-to-use interfaces than would be possible without it, but it's also often a temptation to be overly clever. It's best used in a small number of low level components where the extra maintenance burden is spread out over a large number of uses.

Think twice before using template metaprogramming or other complicated template techniques; think about whether the average member of your team will be able to understand your code well enough to maintain it after you switch to another project, or whether a non-C++ programmer or someone casually browsing the code base will be able to understand the error messages or trace the flow of a function they want to call. If you're using recursive template instantiations or type lists or metafunctions or expression templates, or relying on SFINAE or on the `sizeof` trick for detecting function overload resolution, then there's a good chance you've gone too far.

If you use template metaprogramming, you should expect to put considerable effort into minimizing and isolating the complexity. You should hide metaprogramming as an implementation detail whenever possible, so that user-facing headers are readable, and you should make sure that tricky code is especially well commented. You should carefully document how the code is used, and you should say something about what the "generated" code looks like. Pay extra attention to the error messages that the compiler emits when users make mistakes. The error messages are part of your user interface, and your code should be tweaked as necessary so that the error messages are understandable and actionable from a user point of view.

Boost

Use only approved libraries from the Boost library collection.

Definition:

The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.

Pros:

Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.

Cons:

Some Boost libraries encourage coding practices which can hamper readability, such as metaprogramming and other advanced template techniques, and an excessively "functional" style of programming.

Decision:

In order to maintain a high level of readability for all contributors who might read and maintain code, we only allow an approved subset of Boost features. Currently, the following libraries are permitted:

- Call Traits from `boost/call_traits.hpp`
- Compressed Pair from `boost/compressed_pair.hpp`
- The Boost Graph Library (BGL) from `boost/graph`, except serialization (`adj_list_serialize.hpp`) and parallel/distributed algorithms and data structures (`boost/graph/parallel/*` and

boost/graph/distributed/*).
- Property Map from boost/property_map, except parallel/distributed property maps (boost/property_map/parallel/*).
- The part of Iterator that deals with defining iterators: boost/iterator/iterator_adaptor.hpp, boost/iterator/iterator_facade.hpp, and boost/function_output_iterator.hpp
- The part of Polygon that deals with Voronoi diagram construction and doesn't depend on the rest of Polygon: boost/polygon/voronoi_builder.hpp, boost/polygon/voronoi_diagram.hpp, and boost/polygon/voronoi_geometry_type.hpp
- Bimap from boost/bimap
- Statistical Distributions and Functions from boost/math/distributions
- Multi-index from boost/multi_index
- Heap from boost/heap
- The flat containers from Container: boost/container/flat_map, and boost/container/flat_set

We are actively considering adding other Boost features to the list, so this list may be expanded in the future.

The following libraries are permitted, but their use is discouraged because they've been superseded by standard libraries in C++11:

- Array from boost/array.hpp: use std::array instead.
- Pointer Container from boost/ptr_container: use containers of std::unique_ptr instead.

C++11

Use libraries and language extensions from C++11 (formerly known as C++0x) when appropriate. Consider portability to other environments before using C++11 features in your project.

Definition:

C++11 contains significant changes both to the language and libraries.

Pros:

C++11 was the official standard until august 2014, and is supported by most C++ compilers. It standardizes some common C++ extensions that we use already, allows shorthands for some operations, and has some performance and safety improvements.

Cons:

The C++11 standard is substantially more complex than its predecessor (1,300 pages versus 800 pages), and is unfamiliar to many developers. The long-term effects of some features on code readability and maintenance are unknown. We cannot predict when its various features will be implemented uniformly by tools that may be of interest, particularly in the case of projects that are forced to use older versions of tools.

As with Boost, some C++11 extensions encourage coding practices that hamper readability—for example by removing checked redundancy (such as type names) that may be helpful to readers, or by encouraging template metaprogramming. Other extensions duplicate functionality available through existing mechanisms, which may lead to confusion and conversion costs.

Decision:

C++11 features may be used unless specified otherwise. In addition to what's described in the rest of the style guide, the following C++11 features may not be used:

- Functions (other than lambda functions) with trailing return types, e.g. writing `auto foo() -> int;` instead of `int foo();`, because of a desire to preserve stylistic consistency with the many existing function declarations.
- Compile-time rational numbers (`<ratio>`), because of concerns that it's tied to a more template-heavy interface style.
- The `<cfenv>` and `<fenv.h>` headers, because many compilers do not support those features reliably.
- Default lambda captures.

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.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
int price_count_reader;    // No abbreviation.
int num_errors;            // "num" is a widespread conventio
int num_dns_connections;   // Most people know what "DNS" sta
```

```
int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this
int pc_reader;             // Lots of things can be abbreviat
int cstmr_id;              // Deletes internal letters.
```

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_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc // _unittest and _regtest are deprecated.`

C++ files should end in `.cc` 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).

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_bar.cc`, defining a class called `FooBar`.

Inline functions must be in a `.h` file. If your inline functions are very short, they should go directly into your `.h` file.

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, typedefs, and enums — 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 *, string> PropertiesMap;

// enums
enum UrlTableErrors { ...
```

Variable Names

The names of variables and data members are 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`, `a_class_data_member_`.

Common Variable names

For example:

```
string table_name;  // OK - uses underscore.
string tablename;   // OK - all lowercase.
string tableName;   // Bad - mixed case.
```

Class Data Members

Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
  ...
 private:
  string table_name_;  // OK - underscore at end.
  string tablename_;   // OK.
  static Pool<TableInfo>* pool_;  // OK.
};
```

Struct Data Members

Data members of structs, both static and non-static, are named like ordinary nonmember variables. They do not have the trailing underscores that data members in classes have.

```
struct UrlTableProperties {
  string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

See Structs vs. Classes for a discussion of when to use a struct versus a class.

Global Variables

There are no special requirements for global variables, which should be rare in any case, but if you use one, consider prefixing it with `g_` or some other marker to easily distinguish it from local variables.

Constant Names

Use a `k` followed by mixed case, e.g., `kDaysInAWeek`, for constants defined globally or within a class.

As a convenience to the reader, compile-time constants of global or class scope follow a different naming convention from other variables. Use a `k` followed by words with uppercase first letters:

```
const int kDaysInAWeek = 7;
```

This convention may optionally be used for compile-time constants of local scope; otherwise the usual variable naming rules apply.

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()`.

Regular Functions

Functions should start with a capital letter and have a capital letter for each new word. No underscores.

If your function crashes upon an error, you should append OrDie to the function name. This only applies to functions which could be used by production code and to errors that are reasonably likely to occur during normal operation.

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

Accessors and Mutators

Accessors and mutators (get and set functions) should match the name of the variable they are getting and setting. This shows an excerpt of a class whose instance variable is `num_entries_`.

```
class MyClass {
 public:
  ...
  int num_entries() const { return num_entries_; }
  void set_num_entries(int num_entries) { num_entries_ = num_

 private:
  int num_entries_;
};
```

You may also use lowercase letters for other very short inlined functions. For example if a function were so cheap you would not cache the value if you were calling it in a loop, then lowercase naming would be acceptable.

Namespace Names

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

See Namespaces for a discussion of namespaces and how to name them.

Enumerator Names

Enumerators should be named *either* like constants or like macros: either `kEnumName` or `ENUM_NAME`.

Preferably, the individual enumerators should be named like constants. However, it is also acceptable to name them like macros. 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,
};
```

Until January 2009, the style was to name enum values like macros. This caused problems with name collisions between enum values and macros. Hence, the change to prefer constant-style naming was put in place. New code should prefer constant-style naming if possible. However, there is no reason to change old code to use constant-style names, unless the old names are actually causing a compile-time problem.

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`.

Please see the description of macros; in general macros should *not* be used. However, if they are absolutely needed, then they should be named with all capitals and underscores.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

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

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.

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 Comments

Start each file with license boilerplate, followed by a description of its contents.

Legal Notice and Author Line

Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL).

If you make significant changes to a file with an author line, consider deleting the author line.

File Contents

Every file should have a comment at the top describing its contents.

Generally a `.h` file will describe the classes that are declared in the file with an overview of what they are for and how they are used. A `.cc` file should contain more information about implementation details or discussions of tricky algorithms. If you feel the implementation details or a discussion of the algorithms would be useful for someone reading the `.h`, feel free to put it there instead, but mention in the `.cc` that the documentation is in the `.h` file.

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

Class Comments

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

```
// Iterates over the contents of a GargantuanTable.  Sample u
//    GargantuanTableIterator* iter = table->NewIterator();
//    for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//      process(iter->key(), iter->value());
//    }
//    delete iter;
class GargantuanTableIterator {
  ...
};
```

If you have already described a class in detail in the comments at the top of your file feel free to simply state "See comment at top of file for a complete description", but be sure to have some sort of comment.

Document the synchronization assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

Function Comments

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

Function Declarations

Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments should be descriptive ("Opens the file") rather than 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 witl
// and it must not use the iterator once the GargantuanTable
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of t
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in t
// 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. Notice below that it is not necessary to say "returns false otherwise" because this is implied.

```
// Returns true if the table cannot hold any more entries.
bool IsTableFull();
```

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.

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.

Class Data Members

Each class data member (also called an instance variable or member variable) should have a comment describing what it is used for. If the variable can take sentinel values with special meanings, such as a null pointer or -1, document this. For example:

```
private:
 // Keeps track of the total number of entries in the table.
 // Used to ensure we do not go over the limit. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Global Variables

As with data members, all global variables should have a comment describing what they are and what they are used for. For example:

```
// The total number of tests cases that we run through in thi
const int kNumTestCases = 6;
```

Implementation Comments

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

Explanatory Comments

Tricky or complicated code blocks should have comments before them. Example:

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
  x = (x << 8) + (*result)[i];
  (*result)[i] = x >> 1;
  x &= 1;
}
```

Line Comments

Also, lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by 2 spaces. Example:

```
// If we have enough memory, mmap the data portion too.
```

```
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes,
  return;  // Error already logged.
```

Note that there are both comments that describe what the code is doing, and comments that mention that an error has already been logged when the function returns.

If you have several comments on subsequent lines, it can often be more readable to line them up:

```
DoSomething();                  // Comment here so the commen
DoSomethingElseThatIsLonger();  // Two spaces between the coc
{ // One space before comment when opening a new scope is all
  // thus the comment lines up with the following comments ar
  DoSomethingElse();  // Two spaces before line comments norn
}
vector<string> list{// Comments in braced lists describe the
                    "First item",
                    // .. and should be aligned appropriately
                    "Second item"};
DoSomething(); /* For trailing block comments, one space is f
```

nullptr/NULL, true/false, 1, 2, 3...

When you pass in a null pointer, boolean, or literal integer values to functions, you should consider adding a comment about what they are, or make your code self-documenting by using constants. For example, compare:

```
bool success = CalculateSomething(interesting_value,
                                  10,
                                  false,
                                  NULL);  // What are these a
```

versus:

```
bool success = CalculateSomething(interesting_value,
                                  10,     // Default base val
                                  false,  // Not the first ti
                                  NULL);  // No callback.
```

Or alternatively, constants or self-describing variables:

```
const int kDefaultBaseValue = 10;
const bool kFirstTimeCalling = false;
Callback *null_callback = NULL;
bool success = CalculateSomething(interesting_value,
                                  kDefaultBaseValue,
                                  kFirstTimeCalling,
                                  null_callback);
```

Don'ts

Note that you should *never* describe the code itself. Assume that the person reading the code knows C++ better than you do, even though he or she does not know what you are trying to do:

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

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.

TODO Comments

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

`TODO`s should include the string `TODO` in all caps, followed by the name, e-mail address, or other identifier of the person with the best context about the problem referenced by the `TODO`. The main purpose is to have a consistent `TODO` that can be searched to find out how to get more details upon request. A `TODO` is not a commitment that the person referenced will fix the problem. Thus when you create a `TODO`, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation opera
// TODO(Zeke) change this to use relations.
```

If your `TODO` is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code when all clients can handle XML responses.").

Deprecation Comments

Mark deprecated interface points with `DEPRECATED` comments.

You can mark an interface as deprecated by writing a comment containing the word `DEPRECATED` in all caps. The comment goes either before the declaration of the interface or on the same line as the declaration.

After the word `DEPRECATED`, write your name, e-mail address, or other identifier in parentheses.

A deprecation comment must include simple, clear directions for people to fix their callsites. In C++, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point `DEPRECATED` will not magically cause any callsites to change. If you want people to actually stop using the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

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.

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

Pros:

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?

Cons:

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.

Decision:

80 characters is the maximum.

Exception:

If a comment line contains an example command or a literal URL longer than 80 characters, that line may be longer than 80 characters for ease of cut and paste.

Exception:

A raw-string literal may have content that exceeds 80 characters. Except for test code, such literals should appear near top of a file.

Exception:

An `#include` statement with a long path may exceed 80 columns.

Exception:

You needn't be concerned about header guards that exceed the maximum length.

Non-ASCII Characters

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

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII.

Hex encoding is also OK, and encouraged where it enhances readability — for example, `"\xEF\xBB\xBF"`, or, even more simply, `u8"\uFEFF"`, is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Use the `u8` prefix to guarantee that a string literal containing `\uXXXX` escape sequences is encoded as UTF-8. Do not use it for strings containing non-ASCII characters encoded as UTF-8, because that will produce incorrect output if the compiler does not interpret the source file as UTF-8.

You shouldn't use the C++11 `char16_t` and `char32_t` character types, since they're for non-UTF-8 text. For similar reasons you also shouldn't use `wchar_t` (unless you're writing code that interacts with the Windows API, which uses `wchar_t` extensively).

Spaces vs. Tabs

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

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

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.

Functions look like this:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_r
  DoSomething();
  ...
}
```

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

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
                                             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:

- 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 at the end of the same line as the last parameter.
- The close curly brace is either on the last line by itself or (if other style rules permit) 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 named, with identical names in the declaration and implementation.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

If some parameters are unused, comment out the variable name in the function definition:

```
// Always have named parameters in interfaces.
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

// Always have named parameters in the declaration.
class Circle : public Shape {
 public:
  virtual void Rotate(double radians);
};

// Comment out unused named parameters in definitions.
void Circle::Rotate(double /*radians*/) {}
```

```
// Bad - if someone wants to implement later, it's not clear
// variable means.
void Circle::Rotate(double) {}
```

Lambda Expressions

Format parameters and bodies as for any other function, and capture lists like other comma-separated lists.

For by-reference captures, do not leave a space between the ampersand (&) and the variable name.

```
int x = 0;
auto add_to_x = [&x](int n) { x += n; };
```

Short lambdas may be written inline as function arguments.

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&b
               return blacklist.find(i) != blacklist.end();
             }),
             digits.end());
```

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 retval = 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 retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

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

```
if (...) {
  ...
  ...
  if (...) {
    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 retval = DoSomething(my_heuristic, x, y, z);
```

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

```
bool retval = DoSomething(scores[x] * y + bases[x],  // Score
                          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);
```

Braced Initializer List Format

Format a braced initializer list exactly like you would format a function call in its place.

If the braced list follows a name (e.g. a type or variable name), format as if the `{}` were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
// Examples of braced init list on a single line.
return {foo, bar};
functioncall({foo, bar});
pair<int, int> p{foo, bar};

// When you have to wrap.
SomeFunction(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other values},
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

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 c
  ...
} else {
  ...
}
```

If you prefer you may add spaces inside the parentheses:

```
if ( condition ) {  // spaces inside parentheses - rare
  ...  // 2 space indent.
} else {  // The else goes on the same line as the closing br
  ...
}
```

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 EL
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 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

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 `{}` 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, simply `assert`:

```
switch (var) {
  case 0: {  // 2 space indent
    ...      // 4 space indent
    break;
  }
  case 1: {
    ...
    break;
  }
  default: {
    assert(false);
  }
}
```

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 `{}` or `continue`, but not a single semicolon.

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - empty boc
while (condition) continue;  // Good - continue indicates no
```

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

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 string &str;

// These are fine, space following.
char* c;    // but remember to do "char* c, *d, *e, ...;"!
const string& str;
```

```
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

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

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.

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

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 simp
// Parentheses OK to make a complex expression more readable.
return (some_long_condition &&
        another_condition);
```

```
return (value);                // You wouldn't write var = (v
return(result);                // return is not a function!
```

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};
string name = "Some Name";
string name("Some Name");
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.

```
vector<int> v(100, 1);  // A vector of 100 1s.
vector<int> v{100, 1};  // A vector of 100, 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.
```

Preprocessor Directives

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning o
    DropEverything();
# if NOTIFY               // OK but not required -- Spaces af
    NotifyClient();
# endif
#endif
    BackToNormal();
  }
```

```
// Bad - indented directives
  if (lopsided_score) {
    #if DISASTER_PENDING  // Wrong!  The "#if" should be at b
    DropEverything();
    #endif                // Wrong!  Do not indent "#endif"
    BackToNormal();
  }
```

Class Format

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

The basic format for a class declaration (lacking the comments, see Class Comments for a discussion of what comments are needed) 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 for rules on ordering declarations within each of these sections.

Constructor Initializer Lists

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

There are two acceptable formats for initializer lists:

```
// When it all fits on one line:
MyClass::MyClass(int var) : some_var_(var), some_other_var_(v
```

or

```
// When it requires multiple lines, indent 4 spaces, putting
// the first initializer line:
MyClass::MyClass(int var)
    : some_var_(var),             // 4 space indent
      some_other_var_(var + 1) {  // lined up
  ...
  DoSomething();
  ...
}
```

Namespace Formatting

The contents of namespaces are not indented.

Namespaces do not add an extra level of indentation. For example, use:

```
namespace {

void foo() {  // Correct.  No extra indentation within namesp
  ...
}

}  // namespace
```

Do not indent within a namespace:

```
namespace {

  // Wrong.  Indented when it should not be.
  void foo() {
    ...
  }

}  // namespace
```

When declaring nested namespaces, put each namespace on its own line.

```
namespace foo {
namespace bar {
```

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 b
  ...
int i = 0;  // Semicolons usually have no space before them.
// Spaces inside braces for braced-init-list are optional.  I
// put them on both sides!
int x[] = { 0 };
int x[] = {0};

// Spaces around the colon in inheritance and initializer lis
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between 
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty b
  void Reset() { baz_ = 0; }  // Spaces separating braces fro
  ...
```

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 ar
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parenth
```

```
switch (i) {
for (int i = 0; i < 5; ++i) {
// Loops and conditions may have spaces inside parentheses, b
// 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 m
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
  ...

// Range-based for loops always have a space before and after
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 coc
```

Operators

```
// Assignment operators always have spaces around them.
x = 0;

// Other binary operators usually have spaces around them, bu
// OK to remove spaces around factors.  Parentheses should ha
// 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
vector<string> x;
y = static_cast<char*>(x);

// Spaces between type and pointer are OK, but be consistent.
vector<char *> x;
set<list<string>> x;        // Permitted in C++11 code.
set<list<string> > x;       // C++03 required a space in > >.

// You may optionally use symmetric spacing in < <.
set< list<string> > x;
```

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 discriminating with your use of blank lines inside functions.

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. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgement. But in general, minimize use of vertical whitespace.

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

- Blank lines at the beginning or end of a function very rarely help readability.
- Blank lines inside a chain of if-else blocks may well help readability.

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.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code. Remember that *consistency* includes local consistency, too.

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.

It is worth reiterating a few of the guidelines that you might forget if you are used to the prevalent Windows style:

- Do not use Hungarian notation (for example, naming an integer `iNum`). Use the Google naming conventions, including the `.cc` extension for source files.
- Windows defines many of its own synonyms for primitive types, such as `DWORD`, `HANDLE`, etc. It is perfectly acceptable, and encouraged, that you use

these types when calling Windows API functions. Even so, keep as close as you can to the underlying C++ types. For example, use `const TCHAR *` instead of `LPCTSTR`.

- When compiling with Microsoft Visual C++, set the compiler to warning level 3 or higher, and treat all warnings as errors.
- Do not use `#pragma once`; instead use the standard Google include guards. The path in the include guards should be relative to the top of your project tree.
- In fact, do not use any nonstandard extensions, like `#pragma` and `__declspec`, unless you absolutely must. Using `__declspec(dllimport)` and `__declspec(dllexport)` is allowed; however, you must use them through macros such as `DLLIMPORT` and `DLLEXPORT`, so that someone can easily disable the extensions if they share the code.

However, there are just a few rules that we occasionally need to break on Windows:

- Normally we forbid the use of multiple implementation inheritance; however, it is required when using COM and some ATL/WTL classes. You may use multiple implementation inheritance to implement COM or ATL/WTL classes and interfaces.
- Although you should not use exceptions in your own code, they are used extensively in the ATL and some STLs, including the one that comes with Visual C++. When using the ATL, you should define `_ATL_NO_EXCEPTIONS` to disable exceptions. You should investigate whether you can also disable exceptions in your STL, but if not, it is OK to turn on exceptions in the compiler. (Note that this is only to get the STL to compile. You should still not write exception handling code yourself.)
- The usual way of working with precompiled headers is to include a header file at the top of each source file, typically with a name like `StdAfx.h` or `precompile.h`. To make your code easier to share with other projects, avoid including this file explicitly (except in `precompile.cc`), and use the `/FI` compiler option to include the file automatically.
- Resource headers, which are usually named `resource.h` and contain only macros, do not need to conform to these style guidelines.

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 4.45