C++ Style Guide

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low level-memory manipulation.

It was designed wiht a bias toward system programming and embedded, resource-constrained and larg systems, with preformance, efficiency and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, server (e.g. e-commerce, web search or SQL servers), and preformance-critical applications(e.g. telephone switches or space probes). C++ is a compiled language, wiht implementations of it available on many platforms. Many vendors provide C++ compilers, including the Free Software Foundation, Microsoft, Intel, and IBM.

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as ISO/IEC 14882:20017 (informally known as C++17). The C++ programming language was initially standardized in 1998 as sISO/IEC 14882:1998, which was then amended by the C++03, C++11, and C++14 standards. The current C++17 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1988, C++ was developed by Bjarne Stroustrup at Bell Labs since 1979, as an extension of the C language as he wanted an efficient and flexible language similar to C, which also provided high-level features for programm organization. C++20 is the next planned standard thereafter.

Many other programming languages have been influenced by C++, including C#, D, Java, and newer versions of C.

Background

C++ is one of the main development languages used by many open-source projects. As every C++ programmer knows, the languages 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 out C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

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

Goals of the Style Guide

There are a few core goals that we believe this guide should serve. These are the fundamental **why**s that underlie all of the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to out broader community why the rules are in place and why particular decisions have been made. If you understand what goalds each rule is serving, it should be clearer to everyone when a rule may be waived (some can be), and what sort of argument or alternative whould be necessary to change a rule in the guide.

The goals of the style guide as we currently see them are as follows:

Style rules should pull their weight The benefit of a style rule must be large enough to justify asking all of our engineers to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This principle mostly explains the rules we don't have, rather than the rules we do: for example, goto contraveness many of the following principles, but is already vanishingly rare, so the Style Guide doesn't discuss it.

Optimize for the reader, not the writer Out codebase (ane most individual components submitted to it) is expected to continue for quite some time. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining , and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" isa particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, transfer of pointer ownership), leaving textual hints for the reader at the point of use is valuable (std::unique_ptr demonstrates the ownership transfer unambiguously at the call site).

Be consistent with existing code Using one style consistently though our codebase lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your #includes only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

Be consistent with the broader C++ community when appropriate Consistency with the way other organizations use C++ has value fo rthe

same reasons as consistency within our code base. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it. However, sometimes standard features and idioms are flawed, or were just designed without our codebase's needs in mind. In those cases (as described below) it's appropriate to constrain or ban standard features. In some cases we prefer a homegrown or third-party library over a library defined in the C++ Standard, either out of perceived superiority or insufficient value to transition the codebase to the standard interface.

Avoid surprising or dangerous constructs C++ has features that are more surprising or dangerous than one might think at a glance. Some style guide restrictions are in place to prevent falling into these pitfalls. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.

Avoid constructs that our average C++ programmer would find tricky or hard to maintain C++ has features sthat may not be generally appropriate because of the complexity they introduce to the code. In widely used code, it may be more acceptable to use trickier language constructs, because any benefits of more complex implementations are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid again when working with new portions of the codebase. When in doubt, waivers to rules of this type can be sought by asking your project leads. This is specifically important fo our codebase because code ownership and team membership changes over time: even if everyone that works with some piece of code currently understnads it, such understanding is not guaranteed to hold a few years from now.

Be mindful of our scale With a code base of 100+ million lines and thousands of engineers, some mistakes and simplifications for one engineer can become costly for many. For instance it's particularly important to avoid polluting the global namespace: name collisions across a codebase of hundreds of millions of lines are difficult to work with and hard to avoid if everyone puts things into the global namespace.

Concede to optimization when necessary Preformance optimizations can sometimes be necessary and appropriate, even when they conflict with the other principles of this document.

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

preformance 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 (compile on ther own) and end in .h. Non-header files that are meant for inclusion should end in .inc and be used sparingly.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have header_guards and include all other headers it needs.

Prefer placing the dfinitions for template and inline functions in the same file as their declarations. 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. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files (-inl.h); this practive was common in the past, but is no longer allowed.

As an exception, a template that is explicitly instantiated for all relative sets of template arguments, or that is a private implementation detail of a class, is allowed to be defined in the one and only .cc file that instantiates the template.

There are rare cases where a file designed to be included is not self-contained. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use header_guards, and might not include their prerequisits. Name such files in the .inc extension. Use sparingly, ans prefer self-contained headers when possible.

The #define Guard

All headers 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

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

Definition: A "forward declaration" is a declaration of a class, function, or template without an associated definition.

Pros:

- Forward declarations can save compile time as #includes force the compiler to open more files and process more input.
- Forward declarations can save on unnecessary recompilation. #includes can force your code to be recompiled more often due to unrelated changes in the header.

Cons:

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change.
- A forward declaration may be broken by subsequent changes to the library. Forward declarations of functions and templates can prevent the header owners from making otherwise compatible changes to their APIs, such as widening a parameter type, adding a template parameter with a default value, or migrating to a new namespace.
- Forward declaring symbols from namespace std:: yields undefined behavior.
- It can be difficult to determine whether aforward declaration or a full #include is needed. Replacing an #include with a forward declaration can silently change the meaning of code:

```
// b.h:
struct B();
struct D : B {};

// good_user.cc:
#include "b.h"
voidf(B*);
void f(void*);
void test(D* x) { f(x); }  // calls f(B*)
```

  If the #include was replaced with forward decls for B and D, test() would call f(void*).

- Forward declaring multiple symbols from a header can be more verbose than simply #includeing the header.
- Structuring code to enable forward declarations (e.g. using pointer members instead of object members) can make the code slower and more complex.

Descision:
- Try to avoid forward declarations of entities defined in another project.
- When using a function declared in a header file, always #include that header.
- When using a class template, prefer to #include its header file.

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 fewer.

Definition: You can declare functions in a way that allows the compiler to expand them inline rather than calling them trhough the usual funciton 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, preformance-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 of 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 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.

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 prpose 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 dirr2/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 the usual case 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 sits intent to provide you the smbols of bar.h. However, any includes present in the related header do not need to be included against 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 incudes. Such code can put conditional incudes after other includes. Of course, keep your system-specific code samll and localized. Example:

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

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

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LNAG_CXX11
```

Scoping

Namespaces

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g. using namespace foo). Do no tuse inline namespaces. For unnamed namespaces, see Unnamed Namespaces and Static Variables.

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

Pros: Namespaces provide a method for preventing name conflicts in large programs while allowing most code to use reasonably short names.

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, and code within each project's namespace can continue to refer to Foo without the prefix.

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

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

The expressions X::Y::foo() and X::foo() are interchangable. Inline namespaces are primarily intended for ABI compatibility across version.

Cons: Namespaces cna be confusing, because they complicate the mechanics of figuring out what definitoin a name refers to.

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.

In some contexts, it's necessary to repeatedly refer to symbols by their fully-qualified names. For deeply-nested namespaces, this can add a lot of clutter.

Decision: Namespaces should be used as follos:

- Follow the rules on Namespace Names.
- Terminate the namespaces with comments as shown in the given examples.
- 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 function is within scope of the namespace.
void MyClass::Foo() {
  ...
}

}  // namespace mynamespace
```

More complex .cc files might have additional details, like flags or using-declarations.

```
#include "a.h"

DEFINE_FLAG(bool, someflag, false, "dummy flag");

namespace a {
```

```
using ::foo::bar;

...code for a...  // Code goes against the left margin.

} // namespace a
```

- Do not declar anything in namespace std, including forward declarations of standard library classes. Declaring entities in namespace std in undefined behavior, i.e., not portable. To declar 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;
```

- Do not use *Namespace aliases* at namespace scope in header files except in explicity marked internal-only namespaces, because anything imported into a namespace in a header file becomes part of the public API exported by that file.

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

// Sorten acces to some commonly used names (in a .h file).
namespace librarian {
namespace impl { // Internal, not part of the API.
namespace sidetable = ::pipeline_diagnostics::sidetable;
}  // namespace impl

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

- Do not use inline namespaces.

Unnamed Namespaces and Static Variables

When defintions in a .cc file do not need to be referenced outside that file, place them in an unnamed namespace or declare them static. Do not use either of these constructs in .h files.

Definition: All declarations can be given internal linkage by placing them in unnamed namespaces, and functions and variables can be given internal

linkage by declaring them static. This means that naything you're declaring can't be accessed from another file. If a different file declars something with the same name, then the two entities are completely independent.

Descision: use of internal linkage in .cc files is encouraged for all code that doens not need to be referenced elsewhere. Do not use internal linkage in .h files.

Format unnamed namespaces like named namespaces. In the terminating comment, leave the namespace name empty:

```
namespace {
...
}  // namespace
```

Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Prefer grouping functions with a namespce instead of using a class as if it were a namespace. Static methods of a class should generally be closely related to instances of the class or the class's static data.

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 signifivant dependencies.

Decision: Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Rather than creating classes only to group static member functions which do not share static data, use namespaces instead. For a header myproject/foo_bar.h, for example, write

```
namespace myproject {
namespace foo_bar {
void Function1();
void Function2();
}  // namespace foo_bar
}  //namespace myproject
```

instead of

```
namespace myproject {
class FooBar {
```

```
 public:
  static void Function1();
  static void Function2();
};
}  // namespace myproject
```

If you define a nonmember function and it is only needed in its .cc file, use internal linkage to limit its scope.

Local Variables

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

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

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

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

std::vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization
v.push_back(2);

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

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

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

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

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f; // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}
```

It may be more efficient to declare such a variable used in a loop outsize 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

Variables of class type with static storage duration are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such varaibles 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 varaibles, 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 varaibles 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 non-local static variables to be initialized with the result of a function, unless that funciton (such as getenv() or getpid()) does not itself depend on any other globals. However, a static POD variable within function scope may be initialized with the result of a function, 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 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 wuick_exit() does not invoke destructors and does not invoke any handlers that were registered by calling ateit(). 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 dat. This rule completely disallows std::vector (use C arrays instead), or string (use const char []).

If you need a static or global varaible of a class type, consider initializing a pointer (which will never be freed), from either you main() funciton 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 virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error.

Definition: It is possible to preform arbitrary initialization in the body of the constructor.

Pros:

- No need to worry about whether the class has been initialized or not.
- Objects that are fully initialized by constructor call can be const and may also be easier to use with standard containers or algorithms.

Cons:

- 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 you class is not currently subclassed, causing much confusion.
- There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions (which are forbidden).
- If the work fails, we now have an object whose initialization code fails, so it may be an unusual state requiring a bool IsValid() state checking mechanism (or similar) which is easy to forget to call.
- You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

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

Implicit Conversions

Do not define implicit conversion. Use the explicit keyword for conversion operators and single-argument constructors.

Definition: Implicit conversions allow an object of one type (called the source type) to be used where a different type (called the destination type) is expected, such as when passing an int argument to a function that takes a double parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g. operator bool()). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The explicit keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with cast. This applies not only to implicit conversions, but to C++11's list initialization syntax:

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

Void Func(Foo f);

Func({42, 3.14});  // Error
```

This kind of code isn't technically an implicit conversion, but the language treats it as one as far as explicit is concerned.

Pros:

- Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
- Implicit conversions can be a simpler alternative to overloading.
- List initialization syntax is a concise and expressive way of initializing objects.

Cons:

- Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectations, or the user is unaware that any convension will take place.
- Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actyally getting called.
- Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.

- When a single-argument constructor is not marked explicit, ther's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
- It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.
- List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single elemnt.

Decision: Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition. As an exception, copy and move constructors should not be explicit, sinch they do not preform type conversion. Implicit conversion can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, contact your project leads to request a waiver to this rule.

Constructors that cannot be called with a single argument should usually omit explicit. Constructors that take a single std::initializer_list parameter should also omit explicit, in order to support copy-initialization (e.g. MyType m ={1, 2};).

Copyable and Movable Types

Support cypying and/or moving if these operations are clear and meaningful for your type. Otherwise, disable the implicitly genreated special functions that preform 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 type, the move behavior is defined by the move constructor and the move-assignment operator.

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

Pros: Objects of cypyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, 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,

maintain, and optimize by the compiler. Fourther, such objects can be used with generic APIs that require pass-flexibility in e.g., type composition.

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 elisison.

Move operators allow the implicit and efficient transfer of resouces out of rvalue objects. The allows a plainer coding style in some cases.

Cons: Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (Registerer), objects tied to specific scope (Cleanup), or closely coupled to object identity (Mutex) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically 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 invooked implicitly, which makes the invocation easy to miss. This may cause confusion to programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cuase preformance problems.

Decision: Provide the copy and move operations if their meaning is clear to a casual user and the copying/moving does not incur unexpected costs. I fyou define a copy or move constructor, define the corresponding assignment operator, and vice-versa. If your type is copyable, do not define move operations unless they are significantly mroe efficient than the corresponding copy operations. If your type is not copyable, but the correctness of a move is obvious to users of the type, you may make the type mvoe-only by defining both of the move operations.

If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code, and to document that your class is copyable and/or cheaply movable if that's an API guarantee.

```
class Foo {
 public:
  Foo(Foo&& other) : file_(other.field) {}
  // Bad, defines only move constructor, but not operator=.

 private:
```

```
  Field field_;
};
```

Due to risk of slicing, avoid providing an assignment operator of public copy/move constructor for a clas sthat's intended to be derived form (and avoid deriving from a class with such members). If your base class needs to be copyable, provide a public virtual Clone() method, and 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 in the public: section:

```
// MyClass is neither copyable nor movable.
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

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 bot 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 difitions of all the data and operations that the parent base class defines. In practive, inheritance is used in two major wasy 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. Becuse inheritance is a compile-time declaration, you and th ecompiler cna 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 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 understnad an implementation. The sub-class cannot override functions that are not virtual, sot he sub-class cannot change implementation. The base class may also define some dat amembers, 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 incliding 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 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 fo 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 haave 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 posible exception of the first one, are pure interfaces. In order to ensure that they remain pure interfaces, they must end with the Interface suffix.

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:

- I has only public pure virtal("= 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 provied, 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 neer be directly instantiated because of the pure virtual method(s) it declars. 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, seciton 12.4 for details.

Pros: Taggins a class with the Interface suffix lets other 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 lengthends 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 requirement. We do not require the converse, however: classes that meet the above requirements are not required to end with Interface.

Operator Overloading

Overload operators judiciously. Do not create user-defined literals.

Definition: C++ permits user code to declare overloaded versions of the built-in operators using the operator keyword, so long as one fo the parameters is a user-defined type. The operator keyword also permits user code to define new kinds of literals using operator"", and define type-conversion functions such as operator bool().

Pros: Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g. ==, <, =, and <<), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with librries that expoct those names.

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

Cons:

- Providing a correct, consistent, and unsurprising set of operators overloads requires some care, and failure to do so cand lead to confusion and bugs.
- Overuse of operators can lead to obfuscated code, particularly if the overladed operator's semantics don't follow convention.
- The hazards of function overloading apply just as much to operator overloading, if not more so.
- Operator overlads cna fool our intuition into thinking that expensive operations are chep, built-in operators.
- Finding the call sites for overlaeded operators may require a search tool that's aware of C++ syntax, rather than e.g. grep.
- If you get the argument type of an overloaded operator wrong, you may get a different overload rather than a compiler error. For example, foo < bar may do one thing, while &foo < &bar does something totally different.
- Certain operator overloads are inherently hazardous. Overloading unary & can cause the same code to have different meanings depending on whether the overload declaration is visible. Overloads of &&, ||, and , (comma) cannot mathc the evaluation-order semantics of the built-in operators.
- Operators are often defined outside the class, so there's a risk of different files introdicing differeent definitions of the same operator. If both definitions are linked into the smae binary, this results in undefined behavior, which cna manifest as subtle run-time bugs.
- User-defined literals allow the create of new syntactic forms that are unfamiliar even to experienced C++ programmers.

Decision: Define overladed operators only if their meanding is obvious, un-

suprising, and consistant with the corresongding built-in operators. For example, use | as a bitwise- or logical-or, not as a shell-style pipe.

Define operators only on your own types. More precisely, define them in the same headers, .cc files, and namespaces as the types they operate on. That way, the operators are available wherever the type is, minimizing risk of multiple definitions. If possible, aoid defining operatros as templates, because they must satisfy this rule fo rany possible template arguments. If you define an operator, also define any related operators that make senese, and make sure they are defined consistently. FOr example, if you overload <, overload all the comparison operators, and make sure < and > never return true for the same arguments.

Prefer to define non-modifing binary operators as non-member functions. If a binary operator is defined as a class member, implicit conversion will apply to the right-hand argument, but not the left-hand one. It will confuse your users if a < b compiles but b < a doesn't.

Don't go out of your way to avoid defining operator overloads. For example prefer to define ==, =, and <<, rather than Equals(), CopyFrom(), and PrintTo(). Conversely, don't define operator overloads just because other libraries expect them. For example, if your type doesn't have a natrual ordering, but you wnat to store it in a std:set, use a custom comparator rather than overlading <.

Do not overload &&, ||, , (comma) or unary &. Do not overload operator"", i.e. do not introduce user-defined literals.

Type conversion operators are coverd int he seciton on implicit conversions. THe = operator is covered in the secion on copy constructors. Overloading << for use with streams is covered in the seciton on stream. See also the rules for function overloading, which apply to operator overloading as well.

Access Control

Make data members private, unless they are static const (and follow the nameing convention for constants). For technical reasons, we allow data members of test fixture class to be protected when using Google Test.

Declaration Order

Group similar declarations togeth, placing public parts earlier.

A class definitions should usually start with a public: seciton, follwed by protected:, then private:. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declaration together, and generally prefer the following order: types (including typedef,

using, and nested structs and classes), constants, factory functions, constructors, assignment operators, destructors, all other methds, data members.

Do no tput large method definitions inline in the class definition. Usually, only trivial or preformance-critical, and very short, methods may be defined inline. See Inline Functions for more details.

Functions

Parameter Ordering

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

Parametes to C/C++ functiosn are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be pointers to non-const. When ordering function parameters, put all input-only parameters before any output parameters. In particulary, do not add new parameters to the end of the function just because they are new; place new input-only parameters before th eoutput 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 functoins may require you to bend the rule.

Write Short Functions

Prefer small and focused function.

We recognize that long funcitons are sometimes appropriate, so no hard limit is place don function length. If a function exceeds about 40 lines, think about whether it can be broken up without harming th estructure of the program.

Even if your long funciton 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 wnat to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

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, e.g. 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 stirng &in, strign *out);
```

In fact it is a very strong convention in Google code that input arguments are values of 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 wnat ot 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 treaded 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 exit.

Function Overloading

Use overloaded functions (includeing 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 stirng &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 convention for Visitors.

Cons: If a function is overloaded by the argument types alone, a reader may have to undarstand 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 variantes of a function.

Decision: If you want ot overload a function, consider qualifying the name with some information about the arguments, e.g., AppendString(), AppendInt() rather than just Append(). If you are overloading a function to support variable number of arguments of the same type, consider making it take a std::vector so that the user can use an initializer list to specify the arguments.

Default Arguments

Default argumentws are allowed on non-virtual functions when the default is guaranteed to always have the same value. Follow the same restrictions as for function overloading, and prefer overloaded functions if the readability gained with default arguments doesn't outweight the downsides below.

Pros: Oftem 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 poilerplate and a clearer distinction between 'required' and 'optional' arugments.

Cons: Defaulted arguments are another way to achive the semantics of overloaded functions, so all the reasons not to overload functions apply.

The defaults for arguments in a firtual function call are determined by the static type of the target object, and there's no guarantee that all overrieds of a given function declare the same defaults.

Default parameters are re-evaluated at each call site, which can bloat the generated code. Readers may also expect the default's value to be fixed at the declaration instead of varying at each call.

Funciton pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding function overloads avoids these problems.

Decision: Defautl arguemtns are banned on virtual functions, where they don't work properly, and in cases where the specified default might not evaluate to the same value depending on when it was evaluated. (For example, don't write void f(int n = counter++);)

In some other cases, default argumetns can improve the readability of their function declarations enought to overcom ethe downsides above, so they are allowed. When in doubt, use overloads.

Trailing Return Type Syntax

Use trailing return types only where using the ordinary syntax (leading return types) is impracticaly or much less readable.

Definition: C++ allows two different forms of function declarations. In the older form, the return type appears before the function name. For example:

```
int foo(int x);
```

The new form, introduced in C++11, uses the auto keyword before the funciton name and a trailing return type after the argument list. For example, the declaration above could equivalently be written:

```
auto foo(int x) -> int;
```

The trailing return type is in the function's scope. This doesn't make a difference for a simple case like int but it matters for more complicated casses, like types declared in class scope or types written in terms of the function parameters.

Pros: Trailing retunr types are the only way to explicitly specify the return type of a lambda expression. In some cases the compiler is able to deduce a lambda's return type, but not in all cases. Even when the compiler can deduce it automatically, sometimes specifying it explicitly would be clearer for readers.

Sometimes it's easier and more readable to specify a return type after the function's parameter list has already appeared. This is particularly true when the return type depends on teplate parameters. For example:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

versus

```
template <class T,class U> decltype(declval<T&>() = declval<U&>()) add(T t, U u);
```

Cons: Trailing return type syntax is relatively new and it has no analogue in C++-like langauges like C and Java, so some readers may find it unfamiliar.

Existing code bases have an enormous number of function declarations that aren't going to get chenged to use the new syntax, so the realistic choices are using the old syntex only or using a mixture of the two. Using a single version is better for uniformity of style.

Decision: In most cases, continue to use the older style of function declaration where the return type goes before the funciton name. Use the new trailing-

return-type form only in cases where it's required (such as lambdas) or where, by putting the type after the function's parameter list, it allows you to writ ethe type in a much more readable way. The latter case should be rare; it's mostly an issue in fairly complicated template code, which is discouraged in most cases.

Other C++ Features

Rvalue References

Friends

Exceptions

Run-Time Type Information(RTTI)

Casting

Streams

Preincrement and Predecriment

Use of const

Use of constexpr

Integer Types

64-bit Portability

Preprocessor Macros

0 and nullptr/NULL

sizeof

auto

Braced Initializer List

Lambdaw expressions

Template metaprogramming

Boost

std::hash

C++11

Nonstandard Extensions

Aliases

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 varaibles, a function, a constant, a mactor, etc., without requiring us to search the declaration of that entity. The pattern-matching engine in our brain relies a great deal on thse naming rules.

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

General Naming Rules

Names should be descriptive; avoid 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 immediatly 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 convention.
int num_dns_connections;  // Most people know what "DNS" stands for.
```

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

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

Template parameters should follow the naming style for their category: type template parameters shoudl follow the rules for type names, and non-type template parameters shoudl follow the rules for variable_names.

File Names

File names should be all lowercase and cna include underscores(_) or dashes(-). Follow 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

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 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, type aliases, enums, and type template parameters – have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

```
// classes and structs
class UrlTable { ... }
class UrlTableTester { ... }
struct UrlTableProperties { ... }

// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// using aliases
using PropertiesMap = has_map<UrlTableProperties *, string>;

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

Variable Names

The names of variables (including function parameters) 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_;  // OIK - underscore at end.
  string tablename_;   // OK.
  static Pool<tableInfo>* pool_;  // OK.
}
```

Struct Data Members

Data members of structs, bot static and non-static, are named like ordinary nonmember varialbes. 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.

Constant Names

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case. For example:

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

All such variables with static storage duration (i.e statics and globals, see Storage Duration for detauls) should be named this way. This convention is optimal for variables of other storage classes, e.g. automatic variables, otherwise the usual variable naming rules apply.

Function Names

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

Oridnarily, funcitons should start with a capital letter and have a capital letter for each new word (a.k.a "Camel Case" or "Pascal case"). Such names should not have underscores. Prefer to capitalize acronyms as single words (e.e. StartRpc(), not StartRPC()).

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

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

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

Namespace Names

Namespace names are all lowere-case. Top-level namespace anems are based on the project name. Avoid collisions between nested namespaces and well-known top-level namespaces.

The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose basename matches the namespace name (or subdirectories thereof).

Keep in mind that the rule against abbreviated names applies to namespaces just as much as variable anmes. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespac enames can lead to suprising build breaks because of name lookup rules. In particular, do not created any nested std namespaces. Prefer unique project identifiers (websearch::index, websearch::index_util) over collision-prone names kile websearch::util.

For internal namespaces, be wary of other code being added to the same internal namespace causing a collision (internal helpers within a team tend to be related and may lead to collisions). In such a situation, using the filename to make a

unique internal name is helpful (websearch::index::frobber_internal for use in frobber.h)

Enumerator Names

Enumerators (for both scoped and unscoped enums) should be named *either* like constants or like macros: either kEnumName ro 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 therefor 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 cange 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
```

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. Giing sensible names to types and variables is much better than using obscure names that you must then explain through comments.

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

Comment Style

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

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

File Comments

Start each file with license boilerplate.

File comments describe the contents of a file. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required. All other files must have file comments.

Legal Notice and Author Line

Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project.

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

File Contents

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

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

Class Comments

Every non-obvious class declaration should have an accompanying comment that descries what it is for and how it should be used.

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

The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. 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.

The class comment is often a good place for a small example code snippet demonstrating a simple and focused useage of the class.

When sufficiently seperated (e.g. .h and .cc files), comments describing the use of the class should go together with its interface definition; comments about the class operation and implementation shoudl accompany the implementation of the class's methods.

Function Comments

Declaration comments describe use of the function (when it is non-obvious); comments at the definition of a function describe operation.

Function Declarations

Almost every function declaration should have comments immediately preceding it that describes what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g. simple accessors for obvious properties of the class). These comments should 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 preforms its task. Instead, that should be left to comments in the funciton definition.

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

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

Here is an example:

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

However, od not be unnecessarily verbose or state the completely obviouse. 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 documenting funciton overrides, focuse on the specifics of the override itself, rather than repeating the comment form the overridden function. In many cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and esctructors, remember that the person reading you 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. Is 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 funciotn declaration, in the .h file or wherever. It's okay to recapitualte briefly what the funciton does, but the focuse of the comment 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

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

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

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Global Variables

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

```
// The total number of tests cases that we run through in this regression test.
const int kNumTestCases = 6;
```

Implementation Comments

In your implementation you should have comments in tricky, non-obvious, intresting, 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, mlock))
  return;  // Error already logged.
```

Not 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 comments line up.
DoSomethingElseThatIsLonger();  // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                     // Comments in braced lists describe the next element...
                     "First item",
                     // .. and should be aligned appropriately.
                     "Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

Function Argument Comments

WHen the meaning of a function argument is nonobvious, consider one of the following remedies:

- If the argument is a literal constant, and the same constant is used in multiple function calls in a way that tacitly assumes they're the same, you can use a named constant to make that constraint explicit, and to guarantee that it holds.
- Consider changing the funciton signature to replace a bool argument with an enum argument. THis will make the argument values self-describing.
- For functions that have several configuration options, consider defining a single class or struct to hold all the options, and pass an instance fo that. This approach has several advantages. Options are referenced by name at the call site, which clarifies their meaning. It also reduces function argument count, which makes function calls easier to read and write. As an added benefit, you don't have to change call sites when you add another option.
- Replace large or complex nested expressions with named variables.
- As a last resort, use comments to clarify argument meanings at the call site.

Consider the following example:

```
// What are these arguments?
const DecimalNumber product = CalculateProduct(values, 7, false, nullptr);
```

versus:

```
ProductOptions options;
options.set_precision_decimals(7);
options.set_use_cache(ProductOptions::kDontUseCache);
const DecimalNumber product =
    CalculateProduct(values, options, /*completion_callback=*/nullptr);
```

Don'ts

Do not state the obvious. In particular, don't literally describe what code does, unless the behavior is nonobvious to a reader who understands C++ well. Instead, provide higher level comments that describe *why* the code does what it does, or make the code self describing.

Compare this:

```
// Find the element in the vector.  <-- Bad: obvious!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

To this:

```
// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

Self-describing code doesn't need a comment. The comment from the example above would be obvious:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

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 radable as narative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentance 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 comman 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-enought but not perfect.

TODOs should include the string TODO in call caps, follwed by the name, e-mail address, bug ID, or other identifier of the person or issue 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 proble. THu when you create a TODO with a name, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

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 whn all clients can handle XML responses.").

Deprecation Comments

Mark seprecated interface points wiht DEPRECATED comments.

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

After the word DEPRECATED, write you name, e-mail address, or other identifer 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 undersntad the direcitions, 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 importnat that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Line Length

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

We recognize that this ruel is controversial, but so much existing code already adheres to it, and we fell 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 dont have room to widen their windows in any case. People set up their work environment

assuming a particular maximum window width, and 8- columns has been the traditional standard. Why change it?

Cons: Proponents of change argume 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.

Descision: 80 characters is the maximum.

Exception:
- Comment lines can be longer tham 80 characters if it is not feasible to split them without harming readability, ease of cut and paste or auto-linking – e.g. if a line contains an example command or a literal URL longer than 80 characters.
- A raw-string literal may have content that exceeds 80 characters. Except fo rtest code, such literals should appear near the top of a file.
- An #include statement with a long path may exceed 80 columns.
- You needn't be concerend about header guards that exceed the maxiumum length.

Non-ASCII Characters

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

You shouldn't hard-code user facing text in source, even English, so use non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in you 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 string. 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 OK, and encouraged where it enhances readability – for example, "\\xEF\\xBB\\xBF", or, even more simply, u8"\\uFEFF", is the Unicode xero-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 string 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_name2) {
  DoSomething();
  ...
}
```

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

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

Some points to note:

- Choose good parameter names.
- Parameter names may be omitte donly if the parameter is unused and its purpose is obvious.
- 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 definitoin do not indent.
- The open parenthesis is always on the same line as the function name.
- There is never a spcae betwen the funciton name and the open parenthesis.

- There is never a space between the parentheses and the parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- THe close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be alligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

Unused parameters that are obvious from context may be omitted:

```
class Foo {
 public:
  Foo(Foo&&);
  Foo(const Foo&);
  Foo& operator=(Foo&&);
  Foo& operator=(const Foo&);
};
```

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

```
class Shape {
 public:
  virtual void Rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void Rotate(double radians) override;
};

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

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

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

```
MUST_USE_RESULT bool IsOk();
```

Lambda Expressions

Format parameters and bodies as for any other funcitn, and capture list 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 x_plus_n = [&x](int n) -> int { return 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(), [&blacklist](int i) {
               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 areguments on a new line indented by four spaces and continue at that 4 space indent. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

Function calls have the following format:

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

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

```
bool result = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);
```

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

```
if (...) {
  ...
  ...
  if (...) {
    bool result = DoSomething(
        argument1, argument2,  // 4 space indent
        argument3, argument4);
    ...
```

```
  }
}
```

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

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

```
int my_heuristic = scores[x] * y + bases[x];
bool result = DoSomething(my_heuristic, x, y, z);
```

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

```
bool result = DoSomething(scores[x] * y + bases[x],  // Score heuristic.
                          x, y, z);
```

If there is still a case where one argument is significantl more readable on its own line, the 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 areguments form a structure that is important for readability. In those cases, feel free to format the argument 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 of 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});
std::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 consistant*. If you are modifying a file, use the format that is already present. If you are writing new code, use the format that the other files in that directory or project use. If in doubt and you have no personal preference, do not add the spaces.

```
if (condition) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else if (...) {  // The else goes on the same line as the closing brace.
  ...
} else {
  ...
}
```

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 brace.
  ...
}
```

Note that in all cases you must have a space between the if and the open parenthesis. You must also have a space betwen 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 thsi enhances readability. You may use this only when the line is brief and the statement does not use the else clause.

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

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

```
// Not allowed - IF statement on one line when there is an ELSE clause
if (x) DoThis();
else DoThat();
```

In general, curly braces are not required for single-line statements, but they are allowed if you like them; conditional or loop statements with complex conditions or statements may be more readably 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 empty braces or continue.

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

If not conditional on an enumerated value, switch statements should always have a desfault 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 an empty pari of braces or continue, but not a single semicolon.

```
while (condition) {
  // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - one newline is also OK.
while (condition) continue;  // Good - continue indicates no logic.

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

Pointer and Reference Expressions

No spaces around period or arrow. Pointer operatord ddo 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 sround the period or arrwo 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;
const string& str;
```

It is allowed (if unsual) to declare multiple varaibles in the same declaration, but it is disallowed if any of theose have poitner ro reference decorations. Such declarations are easily misread.

```
// Fine if helpful for readability.
int x, y;

int x, *y;  // Disallowed - no & or * in multiple declaration
char * c;  // Bad - spaces on both sides of *
const 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

Return Values

Variable and Array Initialization

Preprocessor Directives

Class Format

Constructor Initializer Lists

Namespace Formatting

Horizontal Whitespace

Vertical Whitespace