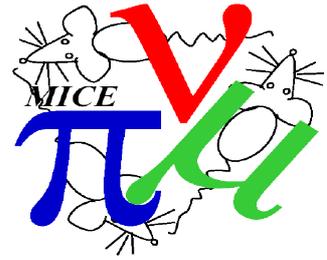# G4MICE C++ Style Guide

G4MICE software management team
Version 0.2 – Draft
2010-12-10

Adapted from Google's C++ Style Guide

Google Revision 3.178

Benjy Weinberger
Craig Silverstein
Gregory Eitzmann
Mark Mentovai
Tashana Landray

Table of Contents

Introduction

The purpose of this document is the pursuance of greater quality and consistency of coding in G4MICE.

The recommended G4MICE style for coding C++ is based on Google's C++ Style Guide (version 3.178). This document sets out that style guide as a first draft.

Programmers should understand that adopting this style is mandatory. However, in the current draft of this document it is recognised that edge cases may exist that are impractical. Users of the guide are therefore encouraged to identify, using the G4MICE issue-tracker, any areas where the recommendations are felt to be inappropriate or could be improved.

At some future date, code will be examined for its conformance with the current G4MICE coding style. Only conformant software will be eligible for release.

What follows is a copy of the Google C++ Style Guide with extremely minor amendments.

At the time of writing this document the Google Style Guide (version 3.178) could be found here:

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Background

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

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

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

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

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

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

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

Header Files

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

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

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

The #define Guard

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

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.hh in project foo should have the following guard:
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif // FOO_BAR_BAZ_H_

Header File Dependencies

Don't use an #include when a forward declaration would suffice. When you include a header file you introduce a dependency that will cause your code to be recompiled whenever the header file changes. If your header file includes other header files, any change to those files will cause any code that includes your header to be recompiled. Therefore, we prefer to minimize includes, particularly includes of header files in other header files.

You can significantly minimize the number of header files you need to include in your own header files by using forward declarations. For example, if your header file uses the Fileclass in ways that do not require access to the declaration of the File class, your header file can just forward declare class File; instead of having to #include "file/base/file.hh".

How can we use a class Foo in a header file without access to its definition?

- We can declare data members of type Foo* or Foo&.
- We can declare (but not define) functions with arguments, and/or return values, of type Foo. (One exception is if an argument Foo or const Foo& has a non-explicit, one-argument constructor, in which case we need the full definition to support automatic type conversion.)
- We can declare static data members of type Foo. This is because static data members are defined outside the class definition.

On the other hand, you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

Sometimes it makes sense to have pointer (or better, scoped_ptr) members instead of object members. However, this complicates code readability and imposes a performance penalty, so avoid doing this transformation if the only purpose is to minimize includes in header files.

Of course, .cc files typically do require the definitions of the classes they use, and usually have to include several header files.

Note: If you use a symbol Foo in your source file, you should bring in a definition for Foo yourself, either via an #include or via a forward declaration. Do not depend on the symbol being brought in transitively via headers not directly included. One exception is if Foo is used in myfile.cc, it's ok to #include (or forward-declare) Foo in myfile.hh, instead ofmyfile.cc.

Inline Functions

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

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

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

Cons:

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

Decision:

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

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

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

The -inl.hh Files

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

The definition of an inline function needs to be in a header file, so that the compiler has the definition available for inlining at the call sites. However, implementation code properly belongs in .cc files, and we do not like to have much actual code in .hh files unless there is a readability or performance advantage.

If an inline function definition is short, with very little, if any, logic in it, you should put the code in your .hh file. For example, accessors and mutators should certainly be inside a class definition. More complex inline functions may also be put in a .h file for the convenience of the implementer and callers, though if this makes the .hh file too unwieldy you can instead put that code in a separate -inl.hh file. This separates the implementation from the class definition, while still allowing the implementation to be included where necessary.

Another use of -inl.hh files is for definitions of function templates. This can be used to keep your template definitions easy to read.

Do not forget that a -inl.hh file requires a #define guard just like any other header file.

Function Parameter Ordering

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

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be non-const pointers. When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

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

Names and Order of Includes

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

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.hh` should be included as

#include "base/logging.hh"

In dir/foo.cc, whose main purpose is to implement or test the stuff in dir2/foo2.hh, order your includes as follows:

1. dir2/foo2.hh (preferred location — see details below).

2. C system files.
3. C++ system files.
4. Other libraries' .hh files.
5. Your project's .hh files.

The preferred ordering reduces hidden dependencies. We want every header file to be compilable on its own. The easiest way to achieve this is to make sure that every one of them is the first .hh file #included in some .cc.

dir/foo.cc and dir2/foo2.hh are often in the same directory (e.g. base/basictypes_test.cc and base/basictypes.hh), but can be in different directories too.

Within each section it is nice to order the includes alphabetically.

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

```
#include "foo/public/fooserver.hh"  // Preferred location.

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

#include <hash_map>
#include <vector>

#include "base/basictypes.hh"
#include "base/commandlineflags.hh"
#include "foo/public/bar.hh"
```

Scoping

Namespaces

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

Definition:

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

Pros:

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

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

Cons:

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

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

Use namespaces according to the policy described below.

Unnamed Namespaces

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

```
namespace {                        // This is in a .cc file.

// The content of a namespace is not indented
enum { kUnused, kEOF, kError };      // Commonly used
tokens.
bool AtEof() { return pos_ == kEOF; }  // Uses our
namespace's EOF.

}  // namespace
```

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

- Do not use unnamed namespaces in .hh files.

Named Namespaces

Named namespaces should be used as follows:

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

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

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

}  // namespace mynamespace

// In the .cc file
namespace mynamespace {

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

}  // namespace mynamespace
```

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

10

```
#include "a.hh"

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

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

namespace b {

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

}  // namespace b
```

- Do not declare anything in namespace std, not even forward declarations of standard library classes. Declaring entities in namespace std is undefined behaviour, i.e., not portable. To declare entities from the standard library, include the appropriate header file.
- You may not use a *using-directive* to make all names from a namespace available.

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

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

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

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

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

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

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

Note that an alias in a .hh file is visible to everyone #including that file, so public headers (those available outside a project) and headers transitively #included by

them, should avoid defining aliases, as part of the general goal of keeping public APIs as small as possible.

Nested Classes

Although you may use public nested classes when they are part of an interface, consider a namespace to keep declarations out of the global scope.

Definition:

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

```
class Foo {

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

};
```

Pros:

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

Cons:

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

Decision:

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

non-member, Static Member, and Global Functions

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

Pros:

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

Cons:

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

Decision:

Sometimes it is useful, or even necessary, to define a function not bound to a class instance. Such a function can be either a static member or a non-member function. non-member functions should not depend on external variables, and should nearly always exist in a namespace. Rather than creating classes only to group static member functions which do not share static data, use namespaces instead.

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

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

Local Variables

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

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

```cpp
int i;
i = f();      // Bad -- initialization separate from declaration.
int j = g();  // Good -- declaration has initialization.
```

Note that gcc implements for (int i = 0; i < 10; ++i) correctly (the scope of i is only the scope of the for loop), so you can then reuse i in another for loop in the same scope. It also correctly scopes declarations in if and while statements, e.g.

```cpp
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.

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

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

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

Static and Global Variables

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

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

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find. Therefore in addition to banning globals of class type, we do not allow static POD variables to be initialized with the result of a function, unless that function (such as getenv(), or getpid()) does not itself depend on any other globals.

Likewise, the order in which destructors are called is defined to be the reverse of the order in which the constructors were called. Since constructor order is indeterminate, so is destructor order. For example, at program-end time a static variable might have been destroyed, but code still running -- perhaps in another thread -- tries to access it and fails. Or the destructor for a static 'string' variable might be run prior to the destructor for another variable that contains a reference to that string.

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

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

Classes

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

Doing Work in Constructors

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

Definition:

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

Pros:

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

Cons:

The problems with doing work in constructors are:

- There is no easy way for constructors to signal errors, short of using exceptions (which are [forbidden](#)).
- If the work fails, we now have an object whose initialization code failed, so it may be an indeterminate state.
- If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
- If someone creates a global variable of this type (which is against the rules, but still), the constructor code will be called before main(), possibly breaking some implicit assumptions in the constructor code. For instance, [gflags](#) will not yet have been initialized.

Decision:

If your object requires non-trivial initialization, consider having an explicit Init() method. In particular, constructors should not call virtual functions, attempt to raise errors, access potentially uninitialized global variables, etc.

Default Constructors

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

Definition:

The default constructor is called when we new a class object with no arguments. It is always called when calling new[ ] (for arrays).

Pros:

Initializing structures by default, to hold "impossible" values, makes debugging much easier.

Cons:

Extra work for you, the code writer.

Decision:

If your class defines member variables and has no other constructors you must define a default constructor (one that takes no arguments). It should preferably initialize the object in such a way that its internal state is consistent and valid.

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

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

Explicit Constructors

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

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

Pros:

Avoids undesirable conversions.

Cons:

None.

Decision:

We require all single argument constructors to be explicit. Always put explicit in front of one-argument constructors in the class definition: explicit Foo(string name);

The exception is copy constructors, which, in the rare cases when we allow them, should probably not be explicit. Classes that are intended to be transparent wrappers around other classes are also exceptions. Such exceptions should be clearly marked with comments.

Copy Constructors

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

Definition:

The copy constructor and assignment operator are used to create copies of objects. The copy constructor is implicitly invoked by the compiler in some situations, e.g. passing objects by value.

Pros:

Copy constructors make it easy to copy objects. STL containers require that all contents be copyable and assignable. Copy constructors can be more efficient than CopyFrom()-style workarounds because they combine construction with copying, the compiler can elide them in some contexts, and they make it easier to avoid heap allocation.

Cons:

Implicit copying of objects in C++ is a rich source of bugs and of performance problems. It also reduces readability, as it becomes hard to track which objects are being passed around by value as opposed to by reference, and therefore where changes to an object are reflected.

Decision:

Few classes need to be copyable. Most should have neither a copy constructor nor an assignment operator. In many situations, a pointer or reference will work just as well as a copied value, with better performance. For example, you can pass function

16

parameters by reference or pointer instead of by value, and you can store pointers rather than objects in an STL container.

If your class needs to be copyable, prefer providing a copy method, such as CopyFrom() or Clone(), rather than a copy constructor, because such methods cannot be invoked implicitly. If a copy method is insufficient in your situation (e.g. for performance reasons, or because your class needs to be stored by value in an STL container), provide both a copy constructor and assignment operator.

If your class does not need a copy constructor or assignment operator, you must explicitly disable them. To do so, add dummy declarations for the copy constructor and assignment operator in the private: section of your class, but do not provide any corresponding definition (so that any attempt to use them results in a link error).

For convenience, a DISALLOW_COPY_AND_ASSIGN macro can be used:

```
// A macro to disallow the copy constructor and operator=
functions
// This should be used in the private: declarations for a class
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
    TypeName(const TypeName&);              \
    void operator=(const TypeName&)
```

Then, in class Foo:

```
class Foo {
  public:
    Foo(int f);
    ~Foo();

  private:
    DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

Structs vs. Classes

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

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

structs should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behaviour but should only be used to set up the data members, e.g., constructor, destructor, Initialize(), Reset(), Validate().

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

For consistency with STL, you can use struct instead of class for functors and traits.

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

Inheritance

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

Definition:

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

Pros:

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

Cons:

For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation. The base class may also define some data members, so that specifies physical layout of the base class.

Decision:

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

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

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

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

When redefining an inherited virtual function, explicitly declare it virtual in the declaration of the derived class. Rationale: If virtual is omitted, the reader has to check all ancestors of the class in question to determine if the function 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 <u>pure interface</u> classes tagged with the Interface suffix.

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

Pros:

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

Cons:

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

Decision:

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

Note: There is an <u>exception</u> to this rule on Windows.

Interfaces

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

Definition:

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

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

An interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, they must also declare a virtual destructor (in an exception to the first rule, this should not be pure). See Stroustrup, The C++ Programming Language, 3rd edition, section 12.4 for details.

Pros:

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

Cons:

The Interface suffix lengthens the class name, which can make it harder to read and understand. Also, the interface property may

be considered an implementation detail that shouldn't be exposed to clients.

Decision:

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

Operator Overloading

Do not overload operators except in rare, special circumstances.

Definition:

A class can define that operators such as + and / operate on the class as if it were a built-in type.

Pros:

Can make code appear more intuitive because a class will behave in the same way as built-in types (such as int). Overloaded operators are more playful names for functions that are colourfully named, such as Equals() or Add(). For some template functions to work correctly, you may need to define operators.

Cons:

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

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

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

Decision:

In general, do not overload operators. The assignment operator (operator=), in particular, is insidious and should be avoided. You can define functions like Equals() andCopyFrom() if you need them. Likewise, avoid the dangerous unary operator& at all costs, if there's any possibility the class might be forward-declared.

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

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

See also Copy Constructors and Function Overloading.

Access Control

Make data members private, and provide access to them through accessor functions as needed (for technical reasons, we allow data members of a test fixture class to beprotected when using Google Test). Typically a variable would be called foo_ and the accessor function get_foo(). You may also want a mutator function set_foo(). Also: static const data members (typically called kFoo) should be private.

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

See also Inheritance and Function Names.

Declaration Order

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

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

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

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

Friend declarations should always be in the private section, and the DISALLOW_COPY_AND_ASSIGN macro invocation should be at the end of the private: section. It should be the last thing in the class. See Copy Constructors.

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

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

Write Short Functions

Prefer small and focused functions.

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

Even if your long function works perfectly now, someone modifying it in a few months may add new behaviour. This could

result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

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

Google-Specific Magic

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

Smart Pointers

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

"Smart" pointers are objects that act like pointers but have added semantics. When a scoped_ptr is destroyed, for instance, it deletes the object it's pointing to. shared_ptr is the same way, but implements reference-counting so only the last pointer to an object deletes it.

Generally speaking, we prefer that we design code with clear object ownership. The clearest object ownership is obtained by using an object directly as a field or local variable, without using pointers at all. On the other extreme, by their very definition, reference counted pointers are owned by nobody. The problem with this design is that it is easy to create circular references or other strange conditions that cause an object to never be deleted. It is also slow to perform atomic operations every time a value is copied or assigned.

Although they are not recommended, reference counted pointers are sometimes the simplest and most elegant way to solve a problem.

cpplint

Use cpplint.py to detect style errors.

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

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

Other C++ Features

All parameters passed by reference must be labeled const.

Definition:

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

Pros:

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

Cons:

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

Decision:

Within function parameter lists all references must be const:

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

In fact it is a very strong convention in Google code that input arguments are values or const references while output arguments are pointers. Input parameters may be const pointers, but we never allow non-const reference parameters.

One case when you might want an input parameter to be a const pointer is if you want to emphasize that the argument is not copied, so it must exist for the lifetime of the object; it is usually best to document this in comments as well. STL adapters such as bind2nd and mem_fun do not permit reference parameters, so you must declare functions with pointer parameters in these cases, too.

Function Overloading

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

Definition:

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

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

Pros:

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

23

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

Decision:

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

Default Arguments

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

Pros:

Often you have a function that uses lots of 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.

Cons:

People often figure out how to use an API by looking at existing code that uses it. Default parameters are more difficult to maintain because copy-and-paste from previous code may not reveal all the parameters. Copy-and-pasting of code segments can cause major problems when the default arguments are not appropriate for the new code.

Decision:

Except as described below, we require all arguments to be explicitly specified, to force programmers to consider the API and the values they are passing for each argument rather than silently accepting defaults they may not be aware of.

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

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

Variable-Length Arrays and alloca()

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

Pros:

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

Cons:

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

Use a safe allocator instead, such as scoped_ptr/scoped_array.

Friends

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

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

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

Exceptions

G4MICE code can use C++ exceptions.

Pros:

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

Cons:

- When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if f() calls g() calls h(), and h throws an exception that f catches,g has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.

- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

Decision:

The benefits of using exceptions outweigh the costs, especially in new projects.

However, exception handling in C++ is not as sophisticated as in Java. Exceptions should be used to allow things to fail gracefully (if possible) and to provide stack-trace information.

The hierarchy of exception handling must be carefully constructed and all possible use-cases must be considered.

See also section on Boost, below.

Run-Time Type Information (RTTI)

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

Definition:

RTTI allows a programmer to query the C++ class of an object at run time.

Pros:

It is useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type.

In rare circumstances, it is useful even outside of tests.

Cons:

A query of type during run-time typically means a design problem. If you need to know the type of an object at runtime, that is often an indication that you should reconsider the design of your class.

Decision:

Do not use RTTI, except in unit tests. If you find yourself in need of writing code that behaves differently based on the class of an object, consider one of the alternatives to querying the type.

Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.

If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

If you think you truly cannot use those ideas, you may use RTTI. But think twice about it. :-) Then think twice again. Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags.

Casting

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

Definition:

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

Pros:

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

Cons:

The syntax is nasty.

Decision:

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

- Use static_cast as the equivalent of a C-style cast that does value conversion, or when you need to explicitly up-cast a pointer from a class to its superclass.
- Use const_cast to remove the const qualifier (see const).
- Use reinterpret_cast to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.
- Do not use dynamic_cast except in test code. If you need to know type information at runtime in this way outside of a unittest, you probably have a design flaw.

Streams

G4MICE code can use streams.

Definition:

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

Pros:

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

27

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

Decision:

G4MICE uses streams, consistency trumps the debate.

Extended Discussion

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

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

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

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

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

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

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

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

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

Preincrement and Predecrement

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

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

Pros:

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

Cons:

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

Decision:

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

Use of const

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

Definition:

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

Pros:

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

Cons:

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

Decision:

const variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors

29

as soon as possible. Therefore we strongly recommend that you use const whenever it makes sense to do so:

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

However, do not go crazy with const. Something like const int * const * const x; is likely overkill, even if it accurately describes how const x is. Focus on what's really useful to know: in this case, const int** x is probably sufficient.

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

Where to put the const

Some people favor the form int const *foo to const int* foo. They argue that this is more readable because it's more consistent: it keeps the rule that const always follows the object it's describing. However, this consistency argument doesn't apply in this case, because the "don't go crazy" dictum eliminates most of the uses you'd have to be consistent with. Putting the const first is arguably more readable, since it follows English in putting the "adjective" (const) before the "noun" (int).

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

Integer Types

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

Definition:

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

Pros:

Uniformity of declaration.

Cons:

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

Decision:

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

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

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

You should not use the unsigned integer types such as uint32_t, unless the quantity you are representing is really a bit pattern rather than a number, or unless you need defined twos-complement overflow. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.

On Unsigned Integers

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

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

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

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

64-bit Portability

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

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

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

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

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

```
#define PRIoS __PRIS_PREFIX "o"
```

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

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

- Remember that sizeof(void *) != sizeof(int). Use intptr_t if you want a pointer-sized integer.

- You may need to be careful with structure alignments, particularly for structures being stored on disk. Any class/structure with a int64_t/uint64_t member will by default end up being 8-byte aligned on a 64-bit system. If you have such structures being shared on disk between 32-bit and 64-bit code, you will need to ensure that they are packed the same on both architectures. Most compilers offer a way to alter structure alignment. For gcc, you can use __attribute__((packed)). MSVC offers #pragma pack()and __declspec(align()).

- Use the LL or ULL suffixes as needed to create 64-bit constants. For example:

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

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

Preprocessor Macros

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

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

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

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

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

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

0 and NULL

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

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

For pointers (address values), there is a choice between 0 and NULL. Bjarne Stroustrup prefers an unadorned 0. We prefer NULL because it looks like a pointer. In fact, some C++ compilers, such as gcc 4.1.0, provide special definitions of NULL which enable them to give useful warnings, particularly in situations where sizeof(NULL) is not equal to sizeof(0).

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

sizeof

Use sizeof(varname) instead of sizeof(type) whenever possible.

Use sizeof(*varname*) because it will update appropriately if the type of the variable changes. sizeof(*type*) may make sense in some cases, but should generally be avoided because it can fall out of sync if the variable's type changes.

Struct data;

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

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

Boost

Boost libraries are not yet approved for G4MICE. Use only approved libraries from the Boost library collection.

Definition:

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

Pros:

Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits, better binders, and better smart pointers. It also provides an implementation of the TR1 extension to the standard library.

The Boost exception library offers a Java-like approach to exception handling.

Cons:

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

Decision:

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

- Call Traits from boost/call_traits.hpp
- Compressed Pair from boost/compressed_pair.hpp
- Pointer Container from boost/ptr_container except serialization and wrappers for containers not in the C++03 standard (ptr_circular_buffer.hpp andptr_unordered*)
- Array from boost/array.hpp
- The Boost Graph Library (BGL) from boost/graph, except serialization (adj_list_serialize.hpp) and parallel/distributed algorithms and data structures (boost/graph/parallel/* and boost/graph/distributed/*).
- Property Map from boost/property_map, except parallel/distributed property maps (boost/property_map/parallel/*).
- The part of Iterator that deals with defining iterators: boost/iterator/iterator_adaptor.hpp, boost/iterator/iterator_facade.hpp, andboost/function_output_iterator.hpp

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

C++0x

Do not use yet. Use only approved libraries and language extensions from C++0x. Currently, none are approved.

34

Definition:

C++0x is the next ISO C++ standard, currently in [final committee draft](#) form. It contains [significant changes](#) both to the language and libraries.

Pros:

We expect that C++0x will become the next standard, and eventually will be supported by most C++ compilers. It standardizes some common C++ extensions that we use already, allows shorthands for some operations, and has some safety improvements.

Cons:

The C++0x standard is substantially more complex than its predecessor (1,300 pages versus 800 pages), and is unfamiliar to many developers. The long-term effects of some features on code readability and maintenance are unknown. We cannot predict when its various features will be implemented uniformly by tools that may be of interest (gcc, icc, clang, Eclipse, etc.).

As with [Boost](#), some C++0x extensions encourage coding practices that hamper readability—for example by removing checked redundancy (such as type names) that may be helpful to readers, or by encouraging template meta-programming. Other extensions duplicate functionality available through existing mechanisms, which may lead to confusion and conversion costs.

Decision:

Use only C++0x libraries and language features that have been approved for use. Currently, no such features are approved. Features will be approved individually as appropriate.

Naming

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

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

General Naming Rules

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

How to Name

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Examples of well-chosen names:

```
int num_errors;                // Good.
int num_completed_connections;   // Good.
```

Poorly-chosen names use ambiguous abbreviations or arbitrary characters that do not convey meaning:

```
int n;                  // Bad - meaningless.
int nerr;               // Bad - ambiguous abbreviation.
int n_comp_conns;       // Bad - ambiguous abbreviation.
```

Type and variable names should typically be nouns:
e.g., FileOpener, num_errors.

Function names should typically be imperative (that is they should be commands): e.g., OpenFile(), set_num_errors(). There is an exception for accessors, which, described more completely in Function Names, should be named the same as the variable they access.

Abbreviations

Do not use abbreviations unless they are extremely well known outside your project. For example:

```
// Good
// These show proper names with no abbreviations.
int num_dns_connections;  // Most people know what "DNS"
stands for.
int price_count_reader;   // OK, price count. Makes sense.
```

```
// Bad!
// Abbreviations can be confusing or ambiguous outside a small
group.
int wgc_connections;  // Only your group knows what this stands
for.
int pc_reader;        // Lots of things can be abbreviated "pc".
```

Never abbreviate by leaving out letters:

```
int error_count;  // Good.
```

```
int error_cnt;    // Bad.
```

File Names

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

Examples of acceptable file names:

```
my_Useful_Class.cc
my-useful-class.cc
myUsefulClass.cc
myUsefulClass_test.cc // _unittest and _regtest are deprecated.
```

C++ files should end in .cc and header files should end in .hh.

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

In general, make your filenames very specific. For example, use http_server_logs.hh rather than logs.hh. A very common case is to have a pair of files called, e.g., foo_bar.hh and foo_bar.cc, defining a class called FooBar.

36

Inline functions must be in a .hh file. If your inline functions are very short, they should go directly into your .hh file. However, if your inline functions include a lot of code, they may go into a third file that ends in -inl.hh. In a class with a lot of inline code, your class could have three files:

```
url_table.hh      // The class declaration.
url_table.cc      // The class definition.
url_table-inl.hh  // Inline functions that include lots of code.
```

See also the section [-inl.hh Files](#)

Type Names

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

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

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

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

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

Variable Names

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

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 (also called instance variables or member variables) are lowercase with optional underscores like regular variable names, but always end with a trailing underscore.

```
string table_name_;  // OK - underscore at end.
string tablename_;   // OK.
```

Struct Variables

Data members in structs should be named like regular variables without the trailing underscores that data members in classes have.

```
struct UrlTableProperties {
    string name;
    int num_entries;
}
```

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

Global Variables

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

Constant Names

Use a k followed by mixed case: kDaysInAWeek.

All compile-time constants, whether they are declared locally, globally, or as part of a class, follow a slightly different naming convention from other variables. Use a k followed by words with uppercase first letters:

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

Function Names

Regular functions have mixed case; accessors and mutators match the name of the variable with the prefix "get" or "set": MyExcitingFunction(), MyExcitingMethod(), getMyExcitingMemberVariable(), setMyExcitingMemberVariable(),.

Regular Functions

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

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

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

Accessors and Mutators

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

```
class MyClass {
  public:
    ...
    int get_num_entries() const { return num_entries_; }
```

```
    void set_num_entries(int num_entries) { num_entries_ =
num_entries; }

  private:
    int num_entries_;
};
```

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

Namespace Names

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

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

Enumerator Names

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

Preferably, the individual enumerators should be named like constants. However, it is also acceptable to name them like macros. The enumeration name, UrlTableErrors (and AlternateUrlTableErrors), is a type, and therefore mixed case.

```
enum UrlTableErrors {
    kOK = 0,
    kErrorOutOfMemory,
    kErrorMalformedInput,
};
enum AlternateUrlTableErrors {
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

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

Macro Names

You're not really going to define a macro, are you? If you do, they're like this: MY_MACRO_THAT_SCARES_SMALL_CHILDREN.

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

#define ROUND(x) ...

```
#define PI_ROUNDED 3.0
```

Exceptions to Naming Rules

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

bigopen()
> function name, follows form of open()

uint
> typedef

bigpos
> struct or class, follows form of pos

sparse_hash_map
> STL-like entity; follows STL naming conventions

LONGLONG_MAX
> a constant, as in INT_MAX

Comments

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

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

Comment Style

Comments should follow the doxygen style. See: http://www.stack.nl/~dimitri/doxygen/docblocks.html

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

Use:

//! <overview of Function()>

or

//! <detailed description of Function()>

void Function()

File Comments

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

Legal Notice and Author Line

Every file should contain the following items, in order:

- a copyright statement
- a license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example GPL)
- an author line to identify the original author of the file

If you make significant changes to a file that someone else originally wrote, add yourself to the author line. This can be very helpful when another contributor has questions about the file and needs to know whom to contact about it.

File Contents

Every file should have a comment at the top, below the copyright notice and author line, that describes the contents of the file.

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

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

Class Comments

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

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

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

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

Function Comments

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

Function Declarations

Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments should be descriptive ("Opens the file") rather than imperative ("Open the file"); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

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

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

Here is an example:

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

However, do not be unnecessarily verbose or state the completely obvious. Notice below that it is not necessary to say "returns false otherwise" because this is implied.

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

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

Function Definitions

Each function definition should have a comment describing what the function does if there's anything tricky about how it does its job. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

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

Variable Comments

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

Class Data Members

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

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

Global Variables

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

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

Implementation Comments

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

Class Data Members

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.
```

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

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

```
DoSomething();                 // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Comment here so there are 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.
}
```

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

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

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

versus:

```
bool success = CalculateSomething(interesting_value,
                                  10,     // Default base value.
                                  false,  // Not the first time we're calling this.
                                  NULL);  // No callback.
```

Or alternatively, constants or self-describing variables:

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

Don'ts

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

```
// Now go through the b array and make sure that if i occurs,
```

```
// the next element is i+1.
...         // Geez.  What a useless comment.
```

Punctuation, Spelling and Grammar

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

Comments should usually be written as complete sentences with proper capitalization and periods at the end. 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. Complete sentences are more readable, and they provide some assurance that the comment is complete and not an unfinished thought.

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

TODO Comments

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

TODOs should include the string TODO in all caps, followed by your name, e-mail address, or other identifier in parentheses. A colon is optional. The main purpose is to have a consistent TODO format searchable by the person adding the comment (who can provide more details upon request). A TODO is not a commitment to provide the fix yourself.

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

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

Deprecation Comments

Mark deprecated interface points with DEPRECATED comments.

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

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

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

Marking an interface point DEPRECATED will not magically cause any callsites to change. If you want people to actually stop using

45

the deprecated facility, you will have to fix the callsites yourself or recruit a crew to help you.

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

Formatting

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

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

Line Length

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

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

Pros:

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

Cons:

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

Decision:

80 characters is the maximum.

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

Exception: an `#include` statement with a long path may exceed 80 columns. Try to avoid situations where this becomes necessary.

Exception: you needn't be concerned about [header guards](#) that exceed the maximum length.

Non-ASCII Characters

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

You shouldn't hard-code user-facing text in source, even English, so use of non-ASCII characters should be rare. However, in certain cases it is appropriate to include such words in your code. For example, if your code parses data files from foreign sources, it may be appropriate to hard-code the non-ASCII string(s) used in those data files as delimiters. More commonly, unittest code (which does not need to be localized) might contain non-ASCII strings. In such cases, you should use UTF-8, since that is an encoding understood by most tools able to handle more than just ASCII. Hex encoding is also OK, and encouraged where it enhances readability — for example, "\xEF\xBB\xBF" is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

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.

Functions look like this:

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

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

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

or if you cannot fit even the first parameter:

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

Some points to note:

- The return type is always on the same line as the function name.
- The open parenthesis is always on the same line as the function name.

- There is never a space between the function name and the open parenthesis.
- There is never a space between the parentheses and the parameters.
- The open curly brace is always at the end of the same line as the last parameter.
- The close curly brace is either on the last line by itself or (if other style rules permit) on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be named, with identical names in the declaration and implementation.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

If your function is const, the const keyword should be on the same line as the last parameter:

```
// Everything in this function signature fits on a single line
ReturnType FunctionName(Type par) const {
  ...
}

// This function signature requires multiple lines, but
// the const keyword is on the line with the last parameter.
ReturnType ReallyLongFunctionName(Type par1,
                                  Type par2) const {
  ...
}
```

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

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

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

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

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

Function Calls

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

Function calls have the following format:

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

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

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

If the function has many arguments, consider having one per line if this makes the code more readable:

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

If the function signature is so long that it cannot fit within the maximum [line length](#), you may place all arguments on subsequent lines:

```
if (...) {
  ...
  ...
  if (...) {
    DoSomethingThatRequiresALongFunctionName(
        very_long_argument1,  // 4 space indent
        argument2,
        argument3,
        argument4);
  }
```

Conditionals

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

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

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

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

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 between the close parenthesis and the curly brace, if you're using one.

```
if(condition)     // Bad - space missing after IF.
if (condition){   // Bad - space missing before {.
if(condition){    // Doubly bad.
```

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

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

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

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

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

In general, curly braces are not required for single-line statements, but they are allowed if you like them; conditional or loop statements with complex conditions or statements may be more readable with curly braces. Some projects require that an if must always 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. Empty loop bodies should use {} or continue.

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

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

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

Empty loop bodies should use {} or continue, but not a single semicolon.

```
while (condition) {
   // Repeat test until it returns false.
}
for (int i = 0; i < kSomeNumber; ++i) {}  // Good - empty body.
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 operators do not have trailing spaces.

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

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.

51

- Pointer operators have no space after the * or &.

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

```
// These are fine, space preceding.
char *c;
const string &str;

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

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

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

Boolean Expressions

When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

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

```
if (this_one_thing > this_other_thing &&
        a_third_thing == a_fourth_thing &&
        yet_another && last_one) {
    ...
}
```

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

Return Values

Do not needlessly surround the return expression with parentheses.

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

```
return result;            // No parentheses in the simple case.
return (some_long_condition &&  // Parentheses ok to make a complex
            another_condition);    //    expression more readable.
```

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

Variable and Array Initialization

Your choice of = or ().

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

```
int x = 3;
int x(3);
string name("Some Name");
string name = "Some Name";
```

Preprocessor Directives

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

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

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
#endif
    BackToNormal();
  }
```

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

Class Format

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

The basic format for a class declaration (lacking the comments, see Class Comments for a discussion of what comments are needed) is:

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
```

```
    bool SomeInternalFunction();

    int some_var_;
    int some_other_var_;
    DISALLOW_COPY_AND_ASSIGN(MyClass);
};
```

Things to note:

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

Constructor Initializer Lists

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

There are two acceptable formats for initializer lists:

```
// When it all fits on one line:
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var
+ 1) {}
```

or

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

Namespace Formatting

The contents of namespaces are not indented.

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

```
namespace {

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

}  // namespace
```

Do not indent within a namespace:

namespace {

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

} // namespace

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

namespace foo {
namespace bar {

Horizontal Whitespace

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

General

void f(bool b) { // Open braces should always have a space before them.
  ...
int i = 0; // Semicolons usually have no space before them.
int x[] = { 0 }; // Spaces inside braces for array initialization are
int x[] = {0};   // optional. If you use them, put them on both sides!
// Spaces around the colon in inheritance and initializer lists.
class Foo : public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {} // No spaces inside empty braces.
  void Reset() { baz_ = 0; } // Spaces separating braces from implementation.
  ...

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

Loops and Conditionals

if (b) {   // Space after the keyword in conditions and loops.
} else {   // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
switch (i) {   // Loops and conditions may have spaces inside
if (test) {   // parentheses, but this is rare. Be consistent.
for (int i = 0; i < 5; ++i) {
for (; i < 5 ; ++i) { // For loops always have a space after the

```
    ...               // semicolon, and may have a space before the
                               // semicolon.
switch (i) {
   case 1:          // No space before colon in a switch case.
      ...
   case 2: break;  // Use a space after a colon if there's code after
it.
```

Operators

```
x = 0;            // Assignment operators always have spaces
around
                        // them.
x = -5;           // No spaces separating unary operators and their
++x;              // arguments.
if (x && !y)
   ...
v = w * x + y / z;  // Binary operators usually have spaces around
them,
v = w*x + y/z;      // but it's okay to remove spaces around
factors.
v = w * (x + z);    // Parentheses should have no spaces inside
them.
```

Templates and Casts

```
vector<string> x;         // No spaces inside the angle
y = static_cast<char*>(x);  // brackets (< and >), before
                               // <, or between >( in a cast.
vector<char *> x;         // Spaces between type and pointer are
                               // okay, but be consistent.
set<list<string> > x;     // C++ requires a space in > >.
set< list<string> > x;    // You may optionally use
                               // symmetric spacing in < <.
```

Vertical Whitespace

Minimize use of vertical whitespace.

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be discriminating with your use of blank lines inside functions.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgement. But in general, minimize use of vertical whitespace.

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

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

Exceptions to the Rules

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

Existing Non-conformant Code

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

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

Windows Code

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

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

- Do not use Hungarian notation (for example, naming an integer iNum). Use the Google naming conventions, including the .cc extension for source files.
- Windows defines many of its own synonyms for primitive types, such as DWORD, HANDLE, etc. It is perfectly acceptable, and encouraged, that you use these types when calling Windows API functions. Even so, keep as close as you can to the underlying C++ types. For example, use const TCHAR * instead of LPCTSTR.
- When compiling with Microsoft Visual C++, set the compiler to warning level 3 or higher, and treat all warnings as errors.
- Do not use #pragma once; instead use the standard Google include guards. The path in the include guards should be relative to the top of your project tree.
- In fact, do not use any nonstandard extensions, like #pragma and __declspec, unless you absolutely must. Using __declspec(dllimport) and __declspec(dllexport) is allowed; however, you must use them through macros such as DLLIMPORT and DLLEXPORT, so that someone can easily disable the extensions if they share the code.

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

- Normally we forbid the use of multiple implementation inheritance; however, it is required when using COM and some ATL/WTL classes. You may use multiple implementation inheritance to implement COM or ATL/WTL classes and interfaces.

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

Parting Words

Use common sense and BE CONSISTENT.

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