Code Style
for
Snake Project

17.11.2017
Version 1.1

Prepared by:
Alexander Orel

Table of Contents

Header Files

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

Correct use of header files can make a huge difference to the readability, size and performance of your code. The following rules will guide you through the various pitfalls of using header files.

Self-contained Headers

Header files should be self-contained (compile on their own) and end in *.h*. Non-header files that are meant for inclusion should end in *.inc* and be used sparingly.

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

Prefer placing the definitions for template and inline functions in the same file as their declarations. The definitions of these constructs must be included into every *.cpp* file that uses them, or the program may fail to link in some build configurations. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files (*-inl.h*). This practice was common in the past, but is no longer allowed.

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

The #define Guard

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

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
...
#endif // _FILE_NAME_H_
```

Forward Declarations

Avoid using forward declarations where possible. Just *#include* the headers you need. Try to avoid forward declarations of entities defined in another project. When using a function declared in a header file, always *#include* that header. When using a class template, prefer to *#include* its header file.

Inline Functions

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

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

Names and Order of Includes

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

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

```
// Related header
#include "foo.h"

// C library
#include <stdio.h>

// C++ library
#include <vector>

// Other library
#include "boost.h"

// Project's header
#include "bar.h"
```

Scoping

Namespaces

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name. Do not use using-directives (e.g. *`using namespace foo`*). Do not use inline namespaces.

Namespaces wrap the entire source file after includes, definitions/declarations and forward declarations of classes from other namespaces. Terminate namespaces with comments as shown in the given examples.

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

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

}  // namespace MyNamespace
```

```
// In the .cpp file
namespace MyNamespace
{

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

}  // namespace MyNamespace
```

Do not declare anything in namespace *`std`*, including forward declarations of standard library classes. Declaring entities in namespace *`std`* is undefined behavior. 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;
```

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

Unnamed Namespaces and Static Variables

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

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

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

Nonmember, Static Member, and Global Functions

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

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

```
namespace MyProject
{

namespace FooBar
{
void Function1();
void Function2();
}  // namespace FooBar

}  // namespace MyProject
```

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

Local Variables

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

C++ allows you to declare variables anywhere in a function. 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:

```
int i;
i = foo();      // Bad - Initialization separate from declaration.
```

```
int i = foo();  // Good - Declaration has initialization.
```

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

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

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

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

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

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

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

```
Foo f;  // Constructor and destructor get called once each.

for (int i = 0; i < 1000000; ++i)
{
    f.DoSomething(i);
}
```

Static and Global Variables

Variables of class type with [static storage duration](#) are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction. However, such variables are allowed if they are *constexpr*: they have no dynamic initialization or destruction.

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

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find. Do not allow non-local static variables to be initialized with the result of a function, unless that function does not itself depend on any other globals. However, a static POD variable within function scope may be initialized with the result of a function, since its initialization order is well-defined and does not occur until control passes through its declaration.

Classes

Doing Work in Constructors

Avoid virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error. Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or `Init()` method. Avoid `Init()` methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

Implicit Conversions

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

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

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

Constructors that cannot be called with a single argument should usually omit `explicit`. Constructors that take a single `std::initializer_list` parameter should also omit `explicit`, in order to support copy-initialization.

Copyable and Movable Types

Support copying and/or moving if these operations are clear and meaningful for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.

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

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

private:
    Field mField;
};
```

If you do not want to support copy/move operations on your type, explicitly disable them using `= delete` in the `public:` section:

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

Structs vs Classes

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

Keyword `struct` should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members.

Inheritance

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

Multiple Inheritance

Only very rarely is multiple implementation inheritance actually useful. Allow multiple inheritance only when at most one of the base classes has an implementation. All other base classes must be pure interface classes tagged with the `I` prefix.

Interfaces

Interface class should satisfies certain conditions and starts with `I` prefix:

- It has only public pure virtual ("= 0") methods and static methods (but see below for destructor).
- It may not have nonstatic 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 `I` prefix.

Operator Overloading

Overload operators judiciously. Do not create user-defined literals. Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use | as a bitwise- or logical-or, not as a shell-style pipe.

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

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

Do not overload `&&`, `||`, `,` (comma), or unary `&`. Do not overload `operator""`.

Declaration Order

Group similar declarations together, placing public parts earlier.

A class definition should starts with constructors and a destructor definition then should follow `public:` section, `protected:` and then `private:`. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order: types (including typedef, using, and nested structs and classes), constructors, assignment operators, destructor, factory functions, all other methods, constants, data members.

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.

Functions

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

Function signature should be the same as in definition.

Write Short Functions

Prefer small and focused functions.

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.

Reference Arguments

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

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

However, there are some instances where using *const T** is preferable to *const T&* for input parameters. For example, you want to pass in a null pointer or the function saves a pointer or reference to the input.

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.

Default Arguments

Default arguments are banned on virtual functions, where they don't work properly, and in cases where the specified default might not evaluate to the same value depending on when it was evaluated.

In some other cases, default arguments can improve the readability of their function declarations enough to overcome the downsides above, so they are allowed. When in doubt, use overloads.

Other C++ Features

Rvalue References

Use rvalue references only to define move constructors and move assignment operators, or for perfect forwarding. You may use `std::move` to express moving a value from one object to another rather than copying it.

Friends

Allowing to 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.

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

Do not use C++ exceptions. On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code.

Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI). RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unit tests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code.

Casting

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

Preincrement and Predecrement

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

Use of const

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

Using *const* variables, data members, methods and arguments add a level of compile-time type checking. It is better to detect errors as soon as possible.

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

Use of constexpr

In C++11, use *constexpr* to define true constants or to ensure constant initialization. The *constexpr* definitions enable a more robust specification of the constant parts of an interface. Use *constexpr* to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with *constexpr*. Do not use *constexpr* to force inlining.

Preprocessor Macros

Avoid defining macros, especially in headers. Prefer inline functions, enums, and const variables. Do not use macros to define pieces of a C++ API.

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

Exporting macros from headers (i.e. defining them in a header without *#undefing* them before the end of the header) is extremely strongly discouraged.

Use of 0 and nullptr/NULL

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

Use of sizeof

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

Use of auto

Use *auto* to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader. Continue to use manifest type declarations when it helps readability. Never initialize an auto-typed variable with a braced initializer list.

Braced Initializer List

You may use braced initializer lists.

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

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

In C++11, this syntax was generalized, and any object type can now be created with a braced initializer list, known as a braced-init-list in the C++ grammar.

```
std::vector<string> v = {"foo", "bar"};
std::map<int, string> m = {{1, "one"}, {2, "two"}};
```

Never assign a braced-init-list to an auto local variable. Always use assign operator with a braced-init-list for more readability.

Lambda expressions

Use lambda expressions where appropriate. Prefer explicit captures when the lambda will escape the current scope.

```
{
    Foo foo;
    ...
    executor->Schedule([&] { Frobnicate(foo); })
    ...
}
// BAD! The fact that the lambda makes use of a reference to ‘foo’ and
// possibly ‘this’ (if ‘Frobnicate’ is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both ‘foo’
// and the enclosing object could have been destroyed.
```

```
{
    Foo foo;
    ...
    executor->Schedule([&foo] { Frobnicate(foo); })
```

```
    ...
}
// BETTER - The compile will fail if ‘Frobnicate’ is a member
// function, and it's clearer that ‘foo’ is dangerously captured by
// reference.
```

Template metaprogramming

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

Use of std::hash

Do not define specializations of `std::hash`. You can use `std::hash` with the types that it supports "out of the box", but do not specialize it to support additional types. If you need a hash table with a key type that `std::hash` does not support, consider using legacy hash containers (e.g. `hash_map`) for now. They use a different default hasher, which is unaffected by this prohibition.

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.

General Naming Rules

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

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

File Names

Filenames should use camel notation and start with a project name. C++ files should end in *.cpp* and header files should end in *.h*. Files that rely on being textually included at specific points should end in *.inc*.

Type Names

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

Variable Names

The names of variables (including function parameters) and data members of structures are all start with lowercase and use camel style.

Class Data Members

Data members of classes, both static and non-static, are named like ordinary nonmember variables, but start with `m` letter.

```
class TableInfo
{
    ...
private:
    string mTableName;
```

```
    static Pool<TableInfo>* mPool;
};
```

Constant Names

Variables declared *constexpr* or *const*, and whose value is fixed for the duration of the program, are with uppercase and with underscores between words.

```
const int DAYS_IN_WEEK = 7;
```

All such variables with static storage duration should be named this way.

Function Names

Ordinarily, functions should start with a capital letter and have a capital letter for each new word. Such names should not have underscores. Prefer to capitalize acronyms as single words.

Enumerator Names

Enumerators (for both scoped and unscoped enums) should be named like constants.

```
enum UrlTableErrors
{
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

Macro Names

In general macros should not be used. However, if they are absolutely needed, then they should be named with all capitals and underscores.

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

Comments

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

Comment Style

Use the `//` syntax, as long as you are consistent.

File Comments

Start each file with license boilerplate.

Class Comments

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

Function Comments

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

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.

Implementation Comments

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

Punctuation, Spelling and Grammar

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

```
//bad style) dO not Wr!tE c0mments in_such_way
```

```
// Good style. Please, follow to punctuation, spelling, and grammar.
```

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the

end of a line of code, can sometimes be less formal, but you should be consistent with your style.

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

Usage of gotcha keywords

Use gotcha keywords for comments. Embedded keywords are used to point out issues and potential problems:

- *:TODO: (mySingle ID)* Means there's more to do here, don't forget.
- *:BUG: [bugID]* Means there's a Known bug here, explain it and optionally give a bug ID.
- *:KLUDGE:* When you've done something ugly say so and explain how you would do it differently next time if you had more time.
- *:TRICKY:* Tells somebody that the following code is very tricky so don't go changing it without thinking.
- *:WARNING:* Beware of something.
- *:COMPILER:* Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.
- *:ATTRIBUTE: value:* The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

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.

Line Length

Each line of text in your code should be at most 120 characters long.

Non-ASCII Characters

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

Spaces vs Tabs

Use only spaces, and indent 4 spaces at a time. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

Functions look like this:

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

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

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

Lambda Expressions

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

Function Calls

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

Function calls have the following format:

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

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

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

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

```
// Transform the widget by a 3x3 matrix.
widget.Transform(x1, x2, x3,
                 y1, y2, y3,
                 z1, z2, z3);
```

Conditionals

Prefer no spaces inside parentheses. The `if` and `else` keywords belong on separate lines. Even single-line statements should be wrapped with braces.

```
if (condition) // No spaces inside parentheses
{ // Braces on the new line
    ...  // 4 space indent.
}
else if (...) {  // The else goes on the new line
    ...
}
else
{
    ...
}
```

Note that in all cases you must have a space between the if and the open parenthesis.

Loops and Switch Statements

Switch statements should use braces for blocks. Braces are mandatory for single-statement loops. Empty loop bodies should use empty braces or continue. Switch statements should always have a `default` case.

```
switch (var)
{
    case 0:
    {  // 4 space indent
        ...      // 4 space indent
        break;
    }

    case 1:
    {
        ...
        break;
    }

    default:
    {
        assert(false);
    }
}

for (int i = 0; i < size; ++i)
{
    printf("I take it back\n");
}
```

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;
```

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

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

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

It is disallowed to declare multiple variables in the same declaration.

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. The logical operator should be at the start of the new lines:

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

Note, that the operands in logical conditions should be always wrapped with brackets.

Return Values

Do not needlessly surround the `return` expression with parentheses. Do that only when it to make a complex expression more readable.

Variable and Array Initialization

It is prefer always to use = during initialization.

```
int x = 3;
int x = {3};
string name = "Some Name";
string name = {"Some Name"};
std::vector<int> v = {100, 1};
```

Preprocessor Directives

The hash mark that starts a preprocessor directive should always be at the beginning of the line. Even when preprocessor directives are within the

body of indented code, the directives should start at the beginning of the line.

```
if (lopsided_score)
{

#ifdef DISASTER_PENDING
    DropEverything();
#endif
    BackToNormal();
}
```

Class Format

Sections in *public*, *protected* and *private* order, each no indented. Constructors, assignment operator and destructor should be at the top of class definition in any case.

```
class MyClass : public OtherClass
{
public:
    MyClass();
    explicit MyClass(int var);
    ~MyClass() {}

    void SomeFunction();
    void SomeFunctionThatDoesNothing() {}

    int GetSomeVar() const
    {
        return mSomeVar;
    }

private:
    bool SomeInternalFunction();

    int mSomeVar;
};
```

Constructor Initializer Lists

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

The acceptable formats for initializer lists are:

```
// When everything fits on one line:
MyClass::MyClass(int var) : mSomeVar(var)
```

```
{
    DoSomething();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
    : mSomeVar(var)
    , mSomeOtherVar(var + 1)
{}
```

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.

Parting Words

Use common sense and BE CONSISTENT.

If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

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

OK, enough writing about writing code. The code itself is much more interesting. Have fun!