C++ Style Guidelines

Rules and Recommendations

Version 1.04

1. Table of Contents

2. Introduction

The purpose of this document is to define one style of programming for C++ projects. Suggestions for improvements are encouraged and should be sent via email to:

ufm-standardization-group@list.bmw.com

Part of these style rules and related examples are based on the publicly available Google C++ Style Guide.

Updated versions of this guide can be found in the following SVN repository:

https://asc-repo.bmwgroup.net/svn/asc034/Shared/CodingStandards/trunk/Documents

Supporting files like configurations for clang-format, and template snippets for Visual Assist X, etc, can be found here:

https://asc-repo.bmwgroup.net/svn/asc034/Shared/CodingStandards/trunk/Config

3. Terminology

- An **identifier** is a name used to refer to a variable, constant, function or type in C++. When necessary, an identifier may have an internal structure consisting of a prefix, a name, and a suffix (in that order).
- A declaration introduces an identifier and describes its type, be it a type, object, or function. A declaration is what the compiler needs to accept references to that identifier. A definition actually instantiates/implements this identifier. It's *what the linker needs* in order to link references to those entities.
- A **class** is a user-defined data type consisting of data elements and functions operating on that data. In C++, this may be declared as a class; it may also be declared as a struct or a union. Variables defined in a class are called member variables and functions defined in a class are called member functions.
- A class/struct/union is said to be an **abstract data type** if it does not have any public or protected member variables.
- A **structure** is a user-defined type consisting of public member variables only.
- **Public members** of a class are member variables and member functions that are accessible from anywhere by specifying an instance of the class and the name.
- **Protected members** of a class are member variables and member functions that are accessible by specifying the name within member functions of derived classes.
- A **class template** defines a family of classes. A new class may be created from a class template by providing values for a number of arguments. These values may be names of types or constant expressions.
- A **function template** defines a family of functions. A new function may be created from a function template by providing values for a number of arguments. These values may be names of types or constant expressions.
- An **enumeration** type is an explicitly declared set of symbolic integral constants. In C++ it is declared as an `enum`.
- A **typedef** is another name for a data type, specified in C++ using a `typedef` declaration.
- A **reference** is another name for a given variable. In C++, the "address of" (`&`) operator is used immediately after the data type to indicate that the declared variable, constant, or function argument is a reference.
- A **macro** is a name for a text string defined in a `#define` statement. When this name appears in source code, the compiler's preprocessor replaces it with the defined text string.
- A **constructor** is a function that initializes an object.
- A **copy constructor** is a constructor in which the only argument is a reference to an object that has the same type as the object to be initialized.
- A **default constructor** is a constructor with no arguments.
- **Composition** is the process of building complex objects from simpler ones. It is used for objects that have a has-a relationship to each other (a Car has-an Engine) and it ensures that the containing object is responsible for the lifetime of the object it holds.

4. Header Files (HDR)

Function Parameters Ordering

Rule HDR-1: When defining a function, parameter order is: inputs, inputs/outputs, outputs.

Include Guards Ordering

Rule HDR-2: The header file should start with an introductory file comment (see COM-2) followed by the #include guard.

5. Classes (CLA)

Interfaces

Rule CLA-1: You may prefix a class with I (`IMyInterfaceClass`) only if the class is a *pure interface*.

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 must 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 also satisfy these conditions and are tagged with the "`I`" prefix.

A pure interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, the interface must also declare a virtual destructor (in an exception to the first rule, this should not be pure, and must have a definition, i.e. must at least be `~IMyInterfaceClass() {}`). See Stroustrup, *The C++ Programming Language*, 3rd edition, section 12.4 for details.

Declaration Order

Rule CLA-2: Use the specified order of declarations within a class: `public:` before `private:`, methods before data members.

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
- Constructors
- Destructor
- Methods
- Data Members

Friend declarations should always be in the private section.

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

6. Naming (NAM)

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

NAM-1: Function names, variable names, and filenames should be descriptive; avoid abbreviations. 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 errors_count;                  // Good.
int completed_connections_count;   // 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`, `errors_count`.

Function names should typically be imperative (that is they should be commands): e.g., `OpenFile()`, `SetErrorsCount()`.

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 dns_connections_count;  // 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

NAM-2: Implement only one class per translation unit. Filenames must be all lowercase and conform to the format *em<unique-prefix>_<logical-entity>.<ext>*

<unique-prefix> is a short acronym identifying the software component, library name or otherwise unique prefix related to the project

<logical-entity> is the name of the C++ class OR it reflects the logical entity implemented in the class.

<ext> must be `.cpp` for C++ implementation files and `.h` for C++ header files

You can only have one class per translation unit, i.e. one class implementation per `.cpp` file, with a corresponding `.h` file of the same name.

- To ensure that the prefix is unique, please consult the UFM Wiki of unique filename prefixes (link) for already used prefixes, and add new ones there accordingly.

For example if you implement a class named `KalmanFilter` in *EmObjectFusion* SWC, you could for example create the following files:

```
emof_kalmanfilter.cpp          // OK, exact class name
emof_kalman_filter.cpp         // Better, class name with
                               // underscores to improve
                               // readability
emof_measurements_fusion.cpp   // Better, logical entity
                               // implemented in the class
kalmanfilter.cpp               // Bad, missing prefix
emof_KalmanFilter.cpp          // Bad, contains uppercase characters
emof_count_lanes.cpp           // Bad, does not reflect logical entity
                               // implemented in the file
```

Inline functions must be in a `.h` file.

Note: the filename uniqueness requirement is pushed up from the SAS integration environment, where if two files have the same name, even if they are in different folders, will be subject to aliasing.

Type Names

NAM-3: 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 start with a capital letter and have a capital letter for each new word. No underscores. When well known acronyms are used, they still follow the same naming rules (i.e. no all uppercase names).

For example:

```
// classes and structs
class UrlTable;
class UrlTableTester;
struct UrlTableProperties;
class XmlStreamReader;
// typedefs
typedef hash_map<UrlTableProperties *, string> PropertiesMap;

// enums
enum UrlTableErrors;
```

Variable Names

NAM-4: 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

NAM-5: Use a `g_` prefix for global variables: `g_my_ugly_global`.

You should avoid global variables, but if you NEED to use one, prefix it with `g_` to easily distinguish it from local variables.

Global Constant Names

NAM-6: Use a `k` followed by mixed case: `kDaysInAWeek`.

All **global constants**, follow a slightly different naming convention from other variables. Use a `k` followed by words with uppercase first letters:

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

Rationale: although it is an accepted convention to use uppercase letters for constants and enumerators, it is possible for third party libraries to replace constant/enumerator names as part of the macro substitution process (macros are also typically represented with uppercase letters).

Function Names

NAM-7: Regular functions begin with a capital letter and have mixed case: `MyExcitingFunction()`, `MyExcitingMethod`.

Namespace Names

NAM-8: Namespace names are all lower-case, and based on project names and follow the bmw hierarchical namespace: `bmw::adas::em::my_awesome_project`.

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

Enumerator Names

NAM-9: Enumerators are named like global constants: `kEnumName`. Enumerators in the *global namespace* should additionally have the type integrated in the name, following this pattern: `kMyRteType_MyVariableName`

The enumeration name, `UrlTableErrors` (and `AlternateUrlTableErrors`), is a type, and therefore begins with a capital letter and has mixed case.

```
enum UrlTableErrors
{
    kOk = 0,
    kErrorOutOfMemory,
    kErrorMalformedInput,
};
```

Macro Names

NAM-10: Macros are named with all capitals and underscores, like in `MY_MACRO_THAT_SCARES_SMALL_CHILDREN`.

Note: macros should generally ***not*** be used for the reasons described in the rule OTH-9.

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

Function and Template Parameters

NAM-11: Function and value template parameters are called like local variables (i.e. lowercase with underscores).

```
template<typename T, size_t stack_capacity>
class StackAllocator;
```

7. Comments (COM)

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

Rule COM-1: Use `//` for comments. Use `///` for doxygen comments (prefix doxygen keywords with `@`, e.g. `@file`)

Write all comments in English.

File Comments

Rule COM-2: Every source file must be documented with an introductory doxygen comment containing *at least* file name and copyright information.

The following template can be used:

```
///
/// @file        my_file.cpp
/// @copyright   Copyright (C) 2016, BMW Group.
///
/// @brief       DESCRIPTION
///
```

Class Comments

Rule COM-3: Every class definition should have an accompanying doxygen comment that describes what it is for and, if relevant, how it should be used.

The following template can be used:

```
///
/// @brief my class brief description
///
/// FULL DESCRIPTION IF NEEDED
///
```

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.

Functions Comments

Rule COM-4: Add doxygen comments at the function **declaration** to describe **use** of the function; if needed, add normal comments at the function **definition** to describe **operation**.

Function declaration

```
///
/// @brief    Description of my function (at the function declaration)
///
/// @param    parameter_1  description of parameter_1
/// @param    parameter_2  description of parameter_1
/// @return   information about return value
///
```

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.

Other types of things to mention in comments at the function declaration:

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

When commenting constructors and destructors, document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

Function definition

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 `.h` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

Rule COM-5: 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 doxygen comment describing what it is used for. If the variable can take sentinel values with special meanings, such as a null pointer or -1, document this. For example:

```
  private:
    ///
    /// @brief 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 total_entries_count_;
```

Global Variables

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

```
/// @brief Total number of tests that we run through in this regression.
const int kTestCasesCount = 6;
```

Implementation Comments

Rule COM-6: In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.

```
// increment counter
++i;
```

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 a null pointer, boolean, or literal integer values to functions, you should make your code self-documenting by using constants. For example, this function call:

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

Is much clearer and refactor-friendly when using self-describing variables:

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

Punctuation, Spelling and Grammar

Rule COM-7: Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

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

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

TODO Comments

Rule COM-8: Use doxygen `@todo` comments for code that is temporary, a short-term solution, or good-enough but not perfect. If the todo is not trivial, it must also contain the Jira Ticket ID that addressed the specific issue or document follow up tasks

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.").

For bigger tasks the related **Jira** issue should have a “fix version” field to indicate a deadline for the change.

Deprecation Comments

Rule COM-9: Mark deprecated interface points with doxygen `@deprecated` comments.

You can mark an interface as deprecated by writing a comment containing the word `@deprecated` followed by your email address in parentheses. The comment goes either before the declaration of the interface or on the same line as the declaration.

After the word `DEPRECATED`, write the related Jira Ticket ID in parentheses.

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

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

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

8. Formatting (FOR)

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 time 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.

Visual assist snippets and other configuration files to help you with it can be found in this repository:

`https://asc-repo.bmwgroup.net/svn/asc034/Shared/CodingStandards/trunk/Config/VisualAssist`

Line Length

Rule FOR-1: Each line of text in your code should be at most 120 characters long.

Exception: if a comment line contains an example command or a literal URL longer than 120 characters, that line may be longer than 120 characters for ease of cut and paste.

Exception: a `#include` statement with a long path may exceed 120 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

Rule FOR-2: Do not use non-ASCII characters.

Spaces vs. Tabs

Rule FOR-3: Do not use tabs, use only spaces, and indent 4 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

Rule FOR-4: 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_nam1, Type par_nam2,
                                             Type par_nam3)
{
    DoSomething();
    ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
        Type par_name1,  // 8 space indent
        Type par_name2,
        Type par_name3)
{
    DoSomething();  // 4 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.
- All parameters should be named, with identical names in the declaration and implementation.
- All parameters should be aligned if possible.
- Default indentation is 4 spaces.
- Wrapped parameters have a 8 space indent (*continuation indent*).

Function Calls

Rule FOR-5: Place 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,  // 8 space indent
                argument2,
                argument3,
                argument4);
    }
```

Conditionals

Rule FOR-6: Do not put spaces inside parentheses. The `else` keyword belongs on a new line. Always use braces, also for single line statements.

You must have a space between the `if` and the open parenthesis.

```
if (condition)
{
    ...  // 4 space indent
}
else if (...)
{
    ...
}
else
{
    ...
}
if(condition)     // Bad - space missing after IF.
if (condition)  // Good - proper space after IF.
```

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` clause:

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

```
if (condition)
{
    DoSomething();  // 4 space indent.
}
```

Loops and Switch Statements

Rule FOR-7: `Switch`, `while do..while` or `for` statements must use braces for blocks (MISRA 6.3.1). 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` (for debugging during the software development phase, be aware that in the target compiler asserts are normally disabled):

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

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

Rule FOR-8: No spaces around period or arrow. Pointer operators do not have trailing spaces. When declaring a pointer variable or pointer/reference argument, you must place the * or & operator adjacent to the type name.

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

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

Note that:

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

When declaring a pointer variable or argument, place the asterisk adjacent to the type. When a const qualifier must be used, follow the pattern in the examples:

```
// These are fine, space following.
char* c;
const string& str;
// const
const uint8* my_const_variable;
const uint8* const my_other_const_variable;
```

```
// These are not fine, space preceding.
Char *c;
const string &str;
char * c;  // Bad - spaces on both sides of *
const string & str;  // Bad - spaces on both sides of &
```

Never declare multiple variables on the same line:

```
char* c, *d, *e, ...;
```

Boolean Expressions

Rule FOR-9: When you have a boolean expression that is longer than the standard line length, break up the lines and consistently put the logical operators at the beginning of the new line.

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

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

Return Values

Rule FOR-10: 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

Rule FOR-11: Prefer the `()` notation for variables initialization.

Prefer the following notation:

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

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

Preprocessor Directives

Rule FOR-12: The hash mark that starts a preprocessor directive should always be at the beginning of the line.

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

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

Rule FOR-13: Sections in `public`, `protected` and `private` order, each indented two spaces.

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 2 space indent!
    MyClass();  // Regular 4 space indent.
    explicit MyClass(int var);
    ~MyClass() {}

    void SomeFunction();
    void SomeFunctionThatDoesNothing()
    {
    }

    void SetSomeVar(int var) { some_var_ = var; }
    int  SomeVar() const { return some_var_; }

  private:
    MyClass(const MyClass &);
    MyClass &operator=(const MyClass &);

    bool SomeInternalFunction();

    int some_var_;
    int some_other_var_;
};
```

Things to note:

- Any base class name should be on the same line as the subclass name, subject to the 120-column limit.
- The `public:`, `protected:`, and `private:` keywords should be indented two spaces.
- Except for the first instance, these keywords should be preceded by a blank line.
- Do not leave a blank line after these keywords.
- The `public` section should be first, followed by the `protected` and finally the `private` section.
- There should be at most one public, one protected and one private section.
- See Declaration Order for rules on ordering declarations within each of these sections.

Constructor Initializer Lists

Rule FOR-14: Constructor initializer lists should be on the subsequent lines and indented eight spaces.

```
// Indent 8 spaces, putting the colon on the first initializer line:
MyClass::MyClass(int var)
        : some_var_(var),             // 8 space indent
          some_other_var_(var + 1)    // lined up
{    ...
    DoSomething();
    ...
}
```

Namespace Formatting

Rule FOR-15: The contents of namespaces are not indented.

Namespaces do not add an extra level of indentation. The end of a namespace should be documented with an end-of-line comment. For example, use:

```
namespace my_project
{

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

}  // namespace myproject
```

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
{
```

End of file

Rule FOR-16: The last line of an implementation or header file must be terminated by a newline (that is, the file ends with a single blank line).

This is required by the C++ standard, but not enforced by most compilers.

Whitespaces

Rule FOR-17: Use of whitespaces depends on location. Never put trailing whitespace at the end of a line.

General

```
int i = 0;  // Semicolons usually have no space before them.
int x[] = { 0 };  // Spaces inside braces for array initialization are
int y[] = {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: do not introduce trailing whitespace and remove them if you find them in a line you are already changing.

Loops and Conditionals

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

```
while (test) {}   // No space inside parentheses.
if (test) ...
switch (i) ...
for (int i = 0; i < 5; ++i) ...
for (; i < 5a; ++i) ...  // For loops always have a space after the
  ...                     // semicolon.

switch (i)
{
    case 1:          // No space before colon in a switch case.
    {
        ...
    }
```

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++03 requires a space in > >.
set< list<string> > x;      // You may optionally use
                            // symmetric spacing in < <.
```