C++ Coding Style Guide

Rules and Recommendations

Version 0.51

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:

EI-58-coding-style-guide@list.bmw.com

A large set of rules and examples comes from the publicly available Google C++ Style Guide but has been modified to better suit the automotive real-time embedded system.

Updated versions of this guide can be found in the following SVN repository, as well as checker tools, some editor configurations files and template snippets for Visual Assist X:

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

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)

The #define Guard

Rule HDR-1: All header files should have `#define` guards to prevent multiple inclusions. The format of the symbol name should be *<UNIQUEPROJECTIDENTIFIER>_<FILENAME>_H_*.

To guarantee uniqueness, they should be based on some unique context identifier, such as the name of the (IDE) project or library to which the file belongs. For example, for the file foo.h, belonging to the project mycoolproject:

```
#ifndef MYCOOLPROJECT_FOO_H_
#define MYCOOLPROJECT_FOO_H_


...


#endif  // MYCOOLPROJECT_FOO_H_
```

Forward declarations

Rule HDR-2: You may forward declare ordinary classes in order to avoid unnecessary `#includes`.

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

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

Always `#include` the file that actually provides the declarations/definitions you need; do not rely on the symbol being brought in transitively via headers not directly included. One exception is that `myfile.cpp` may rely on `#include`s and forward declarations from its corresponding header file `myfile.h`.

Inline Functions

Rule HDR-3: Define functions inline only when they are small, say, 10 lines or less.

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.

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

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

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

Function Parameters Ordering

Rule HDR-4: When defining a function, parameter order is: inputs, inputs/outputs, outputs.

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually const 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 and, as always, consistency with related functions may require you to bend the rule.

5. Scoping (SCO)

Namespaces

Rule SCO-1: Unnamed namespaces in `.cpp` files are encouraged. With named namespaces, choose the name based on the project, and possibly its path. Do not use a *using-directive*. *Using-declaration* in source files are allowed [MISRA 7-3-4]

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

Unnamed Namespaces

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

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

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

Named Namespaces

Named namespaces should be used as follows:

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

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

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

}  // namespace mynamespace
// In the .cpp file
namespace mynamespace
{

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

}  // namespace mynamespace
```

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

```
#include "a.h"

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 behavior, i.e., not portable. To declare entities from the standard library, include the appropriate header file.
- Do 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 `.cpp` file, and in functions, methods or classes in `.h` files.

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

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

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

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

Nested Classes

Rule SCO-2: 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.

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
    {
      ...
    };
};
```

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

Nonmember, Static Member, and Global Functions

Rule SCO-3: Prefer nonmember functions within a namespace or static member functions to global functions; use completely global functions rarely.

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

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

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

Local Variables

Rule SCO-4: Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was

initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.

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

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

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

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

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

Static and Global Variables

Rule SCO-5: 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 declaring a static or global pointer, This pointer is then initialized at runtime with the address of an instance of your class (provided through a singleton pattern or similar construct).

6. Classes (CLA)

Doing Work in Constructors

Rule CLA-1: Avoid doing complex initialization in constructors. In particular, do not do initialization that can fail or that requires virtual method calls.

It is possible to perform initialization in the body of the constructor, but constructors should never call virtual functions or attempt to raise non-fatal failures. If your object requires non-trivial initialization, consider using a factory function or `Init()` method.

Default Constructors

Rule CLA-2: If your class defines member variables and has no other constructors, you must define a default constructor.

The default constructor is called when we create an object without passing arguments. It is always called when calling `new[]` (for arrays).

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

Rule CLA-3: 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.

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

Rule CLA-4: Provide a copy constructor and assignment operator only when necessary. Otherwise, disable them.

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.

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

If your class needs to be copyable, 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&);               \
  TypeName& operator=(const TypeName&);
```

Then, in `class Foo`:

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

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

Structs vs. Classes

Rule CLA-5: Use a `struct` only for passive objects that carry data; everything else is a `class`.

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

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

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

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

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

Inheritance

Rule CLA-6: Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

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.

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 (i.e. using composition).

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

Rule CLA-7: Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be pure interface classes tagged with the `Interface` suffix.

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*.

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

Interfaces

Rule CLA-8: Only classes that satisfy certain conditions are allowed, but not required, to end with an `Interface` suffix.

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

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

Operator Overloading

Rule CLA-9: Do not overload operators except in rare, special circumstances.

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

In general, do not overload operators. The assignment operator (`operator=`), in particular, is insidious and should be avoided. You can define functions like `Equals()` and `CopyFrom()` 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

Rule CLA-10: Make data members `private`, and provide access to them through accessor and mutator functions as needed. Exception: `static const` data members (typically called `kFoo`) need not be `private`.

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

Declaration Order

Rule CLA-11: 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 `.cpp` 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

Rule CLA-12: Prefer small and focused functions. Try to keep them shorter than 40 lines.

Sometimes long functions may be appropriate, but if a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

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 to refactor 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.

7. 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`, `num_errors`.

Function names should typically be imperative (that is they should be commands): e.g., `OpenFile()`, `SetErrorsCount()`. There is an exception for accessors, which, 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 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 <*unique-prefix*>_<*class*>.<*ext*>

<*unique-prefix*> is the software component, library name or otherwise unique prefix related to the project

<*class*> 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 *EmSmartComponent* SWC, you could create the following files:

```
emsmartcomponent_kalmanfilter.cpp         // OK, exact class name
emsmartcomponent_kalman_filter.cpp        // Better, class name with
                                          // underscores to improve
                                          // readability
emsmartcomponent_measurements_fusion.cpp  // Better, logical entity
                                          // implemented in the class
kalmanfilter.cpp                    // Bad, missing prefix
emsmartcomponent_KalmanFilter.cpp   // Bad, contains uppercase characters
emsmartcomponent_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. For example:

```
// classes and structs
class UrlTable;
class UrlTableTester;
struct UrlTableProperties;

// 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

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_` to easily distinguish it from local variables.

Constant Names

NAM-5: 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;
```

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-6: Regular functions begin with a capital letter and have mixed case: `MyExcitingFunction()`, `MyExcitingMethod`.

Namespace Names

NAM-7: Namespace names are all lower-case, and based on project names and possibly their directory structure: `my_awesome_project`.

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

Enumerator Names

NAM-8: Enumerators should be named like constants: `kEnumName`.

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-9: 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
```

Exceptions to Naming Rules

NAM-10: If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme.

8. 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 either the `//` or `/* */` syntax, as long as you are consistent. Use doxygen (`//!` or `/*! */`) when you want the comment to be documented.

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
 *
 * \copyright   Copyright (C) 2014, BMW Group.
 *
 * \par         Project:
 *                MY_PROJECT
 *
 * DETAILED_DESCRIPTION_IF_NEEDED
 */
```

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:

```
/**
 * \class mynamespace::MYCLASS
 *
 * \brief my class brief description
 *
 * My class detailed description
 */
```

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: Doxygen comments at the function **declaration** describe **use** of the function; comments at the function **definition** describe **operation**.

Function declaration

```
/**
  * Description of my function (at the function declaration)
  *
  * \param[in]     parameter_1  description of parameter_1
  * \param[out]    parameter_2  description of parameter_1
  * \param[in,out] parameter_3  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 comment describing what it is used for. If the variable can take sentinel values with special meanings, such as a null pointer or -1, document this. For example:

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

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 test cases that we run through in this regression
test.
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.

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 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 invocation.
                                  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);
```

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.

`TODO`s should include the string \todo, followed by the email address of the person who can best provide context about the problem referenced by the `TODO`.

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 create a **Jira** issue and assign a release where it should be implemented. So you will not forget it when looking at the release plan. Also add the id (e.g. ABC-127) of the Jira issue to the \todo so you can easily find the connection between code and issue.

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 your name, e-mail address, or other identifier in parentheses.

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

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

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

9. 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: an `#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.

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.

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 (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();
```

Always use braces, even for single-line statements:

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

Loops and Switch Statements

Rule FOR-7: 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` (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.

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

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

Note that:

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

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

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

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


// These are fine, space following.
char* c;
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, ...;
```

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

Boolean Expressions

Rule FOR-9: 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, all of the logical AND operators (`&&`) are at the end of the line.

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: 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

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:
    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 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. 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.
- 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
{

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
{
```

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 < 5 ; ++i) ...  // For loops always have a space after 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++03 requires a space in > >.
set< list<string> > x;      // You may optionally use
                            // symmetric spacing in < <.
```

10. Other C++ Features (OTH)

Reference Arguments

Rule OTH-1: All parameters passed by reference must be labeled `const`.

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)`.

We decided that within function parameter lists all references must be `const`:

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

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.

However, there are some instances where using `const T*` is preferable to `const T&` for input parameters. For example:

- You want to pass in a null pointer.
- The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as `const T&`. Using `const T*` instead communicates to the reader that the input is somehow treated differently. So if you choose `const T*` rather than `const T&`, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

Function Overloading

Rule OTH-2: 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.

In C++, it's possible to 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);
};
```

Nevertheless, 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()`.

The reason is that 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.

Friends

Rule OTH-3: 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

Rule OTH-4: We do not use C++ exceptions.

Generally speaking, exception handling is a powerful tool to effectively deal with errors. There are, however, some disadvantages in enabling them in real-time embedded software, mainly because of a negative impact in code size and difficultly in writing correct exception-safe code.

Run-Time Type Information (RTTI)

Rule OTH-5: Avoid using Run Time Type Information (RTTI).

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of `typeid` or `dynamic_cast`. RTTI has some legitimate uses (e.g. unit tests) but is prone to abuse.

Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed. Also, RTTI is normally disabled in the target compiler.

Casting

Rule OTH6 Use C++ casts like `static_cast<>()`. Do not use other cast formats like `int y = (int)x;` or `int y = int(x);`.

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

Do not use C-style casts. Instead, use these C++-style casts, as they remove the ambiguity between type conversion and casting:

- 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` (see rule OTH-5).

Preincrement and Predecrement

Rule OTH-7: Use prefix form (`++i`) of the increment and decrement operators with iterators and other template objects.

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, use the pre-increment form in this case.

Use of const

Rule OTH-8: Use `const` whenever it makes sense.

In C++, 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; };`).

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

If a function does not modify an argument passed by reference or by pointer, that argument should be `const`.

Declare methods to be `const` whenever possible. Accessors should almost always be `const`. Other methods should be const if they do not modify any data members, do not call any non-`const` methods, and do not return a non-`const` pointer or non-`const` reference to a data member.

Consider making data members `const` whenever they do not need to be modified after construction.

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

Some people favor the form `int const *foo` to `const int* foo`, but putting the `const` first is arguably more readable, since it follows English in putting the "adjective" (`const`) before the "noun" (`int`), therefore this form is encouraged.

Preprocessor macros

Rule OTH-9: Be very cautious with macros. Prefer inline functions, enums, and `const` variables to macros.

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

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

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

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

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

0 and NULL

Rule OTH-10: Use `0` for integers, `0.0F` for reals, `NULL` for pointers, and `'\0'` for chars.

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

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

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

Sizeof

Rule OTH-11: 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.

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

11. C++ and AUTOSAR (AUT)

Object constructions by runtime environment

Rule AUT-1: All objects outside of functions/members have to be constructed by the runtime system

This is especially important if the object's class has virtual functions or virtual base classes. In this case the according pointers have to be initialized before the class is being used. This is also done in auto-generated code before the actual constructor is called. Some ECUs, however, may not call the constructors of static objects on startup or after wakeup (where the memory may be corrupted due to power loss). In this case, the only possible way to properly initialize the class is by using the **placement-new** technique. Unlike a normal new operator call, this does not allocate heap memory but constructs an object in a pre-defined memory location. The syntax of the call looks like this: new (&object) ClassOfObject(arg1, arg2, ...). This will construct an instance of 'ClassOfObject' at the location of 'object'. It is **very** important that 'ClassOfObject' is at most as large as the original type of 'object'. Otherwise, you will have a buffer overflow.

RTE calls in member functions

Rule AUT-2: Member functions shouldn't call RTE functions

This may seem strange at first. However, this is one consequence from an important AUTOSAR property that also applies to C: Whenever an AUTOSAR component reads/writes data or executes Client/Server calls, its model has to declare a data send/data receive point or a server call point for its runnables. As a consequence, the developer has to ensure that any function called by a runnable only accesses the ports declared by the model. Therefore, if a port access is added to a function, a port access point has to be added to all runnables that currently use this function, directly or indirectly. While this is feasible in C if you limit yourself to adding RTE calls only to static functions near your runnable and you create one module for each runnable, this will turn into a dependency nightmare with objects. One special case are constructors called by the runtime system: Since there is no runnable you could assign port access points to, any RTE call is illegal in that context.

C/C++ standard libraries

Rule AUT-3: Large portions of the STL are unusable

In general, C and C++ standard libraries are not part of the AUTOSAR world. However, there are incarnations of the STL which only need headers, so one might be tempted to use them anyways. Although this may indeed work for things like algorithms and binders, most container classes rely on the availability of the new operator or any other means of dynamic memory management. While the allocation of the actual objects can be customized using allocators, some containers need additional administrative memory areas that they allocate using malloc/new. To enjoy the convenience and safety of template containers, you should instead use inplace lists or lists with a preallocated maximum number of objects. A sample implementation can be found in the SBL in the shared department of Ascent.