Coding Standard - C++

Author	Version	Date
Coen Custers	0.1 (proposal)	2014-11-17
Coen Custers	0.2 (review rewrite)	2015-01-04
Coen Custers	0.3 (2nd version)	2015-03-10

Table of Contents

1.Introduction

The Code Standard C++ requires or recommends certain practices while developing C++ programs and packages. The intent of this document is not to be extensive, but more an ongoing document. Only those elements of the language will be included and focused on which are of interest for the embedded software development team inside Metrological BV. In time this document can be extended or updated by the insight and needs at that time.

The objective of the document is to have a positive effect on:

- Avoidance of errors/bugs, especially hard to find ones.
- Maintainability, by promoting proven design principles.
- Maintainability, by requiring or recommending some unit of style
- Performance, by dissuading wasteful practices.

Finally, this document is sub-divided in three section, namely: Development Architecture, Code Style, Code Guidelines and Code Rules.

2.Development Architecture

2.1. Task Development Cycle.

All changes or improvements to the code base are sub-divided into tasks. Depending on the effort involved, either small or large, to complete the task, one has to go through either the partial or full task development cycle (see below).

a) Full Task Development Cycle.

➔ *Design – where we adhere to the principles of "design-by-contract".*

➔ *Branch - create a separate branch of the master for the task at hand.*

➔ *Implementation – code the functionality and an additional unit-test if needed.*

➔ *Local testing – test your code locally, before passing it on for a peer-review.*

➔ *Peer-review – let the task be code-reviewed by a peer.*

➔ *Global Testing – build and test you branch on the build-bot.*

➔ *Merge – if all went well merge the code with master (otherwise go back to implementation).*

b) Partial Task Development Cycle.

➔ *Design – where we adhere the principles of "design-by-contract".*

➔ *Implementation – code the functionality and an additional unit-test if needed.*

➔ *Local testing – test your code locally, before passing it on for a peer-review.*

➔ *Peer-review – let the task be code-reviewed by a peer.*

➔ *Merge – if the evaluation went well (with/without peer-review) merge with the master..*

➔ *Evaluate – build the merged master on the build-bot. If the build was successful, the task is done, otherwise go back to implementation.*

2.2. External Services and Software Packages

To enable the Task Development Cycle the following external services and software packages are used (this excluding all software packages used in each project individually). In this list are project specific sites excluded.

To enable this Task Development Cycle the following software packages and sites are used:

- http://www.github.com/ : A Git-repositories hosting provider for backup and version control.
- http://sourceforge.net/projects/cppunit/ : An unit-test framework for C++ language.
- http://jenkins-ci.org/ : An open-source continuous integration server
- https://www.joyent.com/: Enterprise hosting services that host our build bots.
- http://buildroot.uclibc.org/ : Tool for building embedded Linux system through cross-compilation.

3. Code Style

3.1. Introduction

First if all, code style is all about uniformity, not about “Right or wrong”. In our case it is more about keeping a kind of uniformity while keeping in mind the preferences of a individual developer of which text-editor/IDE to use. In order to achieve this kind of uniformity some rules-of-the-road a developer has to adhere to.

3.2. Naming conventions

Only the following naming rules exist, the rest of the naming is up to the developer.

filename	Pascal Case	MyFileName.h MyFileName-inl.h MyFileName.cpp
namespace	Pascal Case	*MyNameSpace* *{* …. *} // End of MyNameSpace.*

class	Pascal Case	*MyClass*
		{
		….
		};

class member method	Pascal Case	*GetMyMethod();*
class member variable	'_' + Camel Case **or** Came Case + '_'	*_myVariable;*

3.3. Indentation

In order to achieve uniformity of indentation for all possible text-editors the following rule exists, set the in the editor of choice to the following two settings:

- Set TAB-size to 2 or 4 SPACES, but replace TAB-character by SPACES.

This will has as result that each indentation unit is always set to 2 or 4 SPACES. And reversely, this also implies that when one wants to indent via the SPACE-button, one also has use 2 or 4 SPACES to insert an indentation in the code-text. *However, this rule only applies to source and header files and not to Makefiles (which makes use of the TAB-character).*

And for reasons of uniformity and readability, leave an empty line between two class-declarations.

3.4. Comments

Try to keep additions of comments as minimal as possible, in other words, try to make the code as self-explanatory as possible. If you do choose to add a comment, the comment has to add additional necessary context to the code. Of course, any comment also has to be written in correct English.

Any comment should be included relative to their position in the code, just before or after the piece of code you are commenting on, but not on the same line. This will increase the readability and predictability of where to expect comments.

4. Code Rules and Guidelines: General

4.1. Introduction

The following distinction is made between a Code Rule and a Code Guideline. On a Code Rule there are no exception to the rule, i.e. a Code Rule applies always in any circumstance. A Code Guideline is a good practice, so if there are valid arguments to deviate from the Code Guideline this is still a good practice.

4.2. General Design

The Code Guidelines and Code Rules are intended to develop code that exhibits the following important qualities :

- **Reliability**: Executable code should consistently fulfill all the requirements in a predictable manner**.**
- **Portability**: Source code should be portable (i.e. not compiler or linker dependent).
- **Maintainability**: Source code should be written in a manner that is consistent, readable, simple in design, and easy to debug.
- **Testability**: Source code should be written to facilitate testability. Minimizing the following characteristics for each software module will facilitate a more testable and maintainable module: code size, complexity, static path count (number of paths through a piece of code).
- **Re-usability**: The design of reusable components is encouraged. Component reuse can eliminate redundant development and test activities (i.e. reduce costs).
- **Extensibility**: Requirements are expected to evolve over the life of a product. Thus a system should be developed in an extensible manner (i.e. changes in requirements shall preferably be managed through local extensions rather than through architectural changes).
- **Readability**: Source code should be written in a manner that is easy to read, understand and comprehend.

4.3. High Cohesion and loose coupling

If a system is decomposed into modules, we are able to talk about coupling and cohesion as properties of that system. Cohesion is a measure of how well the internal parts of the modules fit together and Coupling is a measure how many interactions/relationships there exist between the different modules.

The design principles behind OO techniques lead to data cohesion within modules and clean interfaces between modules enables the modules to be loosely coupled. Besides the fact that data encapsulation and data protection provides mechanisms for enforcing the coupling and cohesion goals. Which leads to the following guidelines for high cohesion and loose coupling:

G1	*Limit hardware and external software interfaces to a small number of functions.*
G2	*Minimize the use of global data.*
G3	*Minimize the exposure of implementation detail.*

4.4. Headers and Sources

G4	*In general, every <filename>.cpp file should have and associated <filename>.h file.*

<u>Reasoning</u>:

There are some common exceptions, such as unit-tests and small .cpp files containing just a main() function. Correct use of header files can make a huge difference to the readability, size and performance of your code.

R1	*A header should be self-contained, i.e. a header should not rely on other headers that have to be included before them.*

<u>Reasoning</u>:

Suppose that the header was not self contained. In that situation if you wish to refer to the symbols *declared in that header in a different client, then the new client would not even compile unless you found and pulled the necessary headers.*

R2	*All header files should have* `#define` *guards to prevent multiple inclusion.*

Reasoning:

To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file `/project/plugin/go.h` *in project* `foo` *should have the following guard:*

```
#ifndef PROJECT_PLUGIN_GO_H_

#define PROJECT_PLUGIN_GO_H_

  ....

#endif // PROJECT_PLUGIN_GO_H_
```

4.5. Exceptions and Asserts

R3	*Only use exceptions in exceptional cases, otherwise just use assert-statements.*

Reasoning:

Because an exception-handling requires a full rollback of the call-stack, which is an costly mechanism and not necessarily thread-safe. Therefore in situations where the requirements cannot be met, use the assert-statement instead, until the requirements can be met in the future, in which case the assert-statement will become obsolete anyway.

4.6. Disable copying objects

R4	*Prevent (accidental copying) of objects.*

Reasoning:

By declaring the copying-method *and the* assign-operator *private and not defining them (i.e. not implementing them), will disallow copying of objects. This leads to compilation or linking errors whenever an instance of this object is copied or assigned. By keeping the number of objects at a bare minimum both benefits less memory usage as higher performance in execution.*

4.7. Use const pro-actively

R5	*If something can be made* ***const****, make it* ***const***

Reasoning:

*"***const*** is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables whenever it is sensible and make* **const** *your default when you define a value: It's safe, it's checked at compile time, and it's integrated with C++'s type system. Don't cast away* **const** *except to call a* **const***-incorrect function",* quote from **[1]** .

4.8. Initialization

R6	*Always initialize variables upon definition.*

Reasoning:

Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it.

R7	*Every class member variable must always be initialized.*

Reasoning:

For each class member variable an in-class initializer must be provided in the constructor or alternatively each member variable must be initialized. If you do not declare any constructors *yourself then the compiler will generate a default constructor for you, which may leave some fields uninitialized or initialized to inappropriate values.*

4.9. Operators

R8	*Preserve natural semantics for overloading operators.*

Reasoning:

Only overload operators for a good reason, preserving the natural semantics. If that is difficult, you probably are misusing the operator overloading.

4.10. Type Casting

R9	Always use C++-style casting and never C-style casting.

Reasoning:

C++-style casting provides extra compiler checks for unsafe-casting, besides it also improves search ability during debugging.

4.11. Data types

R10	Always use integer-types with explicit bit-size.

Reasoning:

Default atomic integer-types like: char, short, integer, long or size_t are compiler dependent. An integer, for example, could be 16-, 32-, of 64-bits long, instead of the assumed 32-bits. And a char could be 16-bits long, instead of 8-bits. By using integer-types with explicit bit-size this variance is avoided. Like, for example, Sint8, Sint16, Sint32, Sint64, Uint8, Uint16, Uint32 and Uint64.

4.12. Use of Parentheses

G5	It is better to use parentheses liberally.

Reasoning:

Even in cases where operator precedence unambiguously dictates the order of evaluation of an expression, often its is beneficial for reasons of readability and maintainability.

4.13. Referencing versus Pointers

R11	Use reference wherever you can, pointers wherever you must.

Reasoning:

References are usually preferred over pointers whenever you don't need "re-seating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value can be NULL.

4.14. Composition versus inheritance

G6	Prefer composition versus inheritance.

Reasoning:

Avoid inheritance taxes: Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.

In this context "composition" means simply means embedding a member of a type within another type. This way, you can hold and use the object in ways that allow you the control of the strength of the coupling., quote from ***[1] .***5. Code Rules and Guidelines: multi-threading

5. Code Rules and Guidelines: Multi-threading

5.1. Introduction

Besides the expected Code Rules and Guidelines, some sections mainly exists to provide context to the introduced Code Rules and Guidelines. This chapter will focus on concurrency and multi-threading. In this respect several (probably) known concepts and names will be defined (again) here in this document. This is done because sometimes these concepts are mixed or mis-used, which only troubles the waters.

5.2. Concurrency versus Parallelism

Concurrency is related to but distinct from parallel computing, though these concepts are frequently confused, and both can be described as *"multiple processes executing during the same period of time"*. In parallel computing the execution literally occurs as the same instant, for example on separate processors on a multi-core machine. By contrast, concurrent computing consists of process lifetimes overlapping, but the execution need not to happen on the same instant. The goal is to model processes in the outside world that happen concurrently, such as multiple clients accessing a server at the same time.

For example, concurrent processes can be executed on a single core by interleaving the execution steps of each process via time slices: only one process runs at a time, and if it does not complete during its time slice, it is paused, another process begins and resumes, and then later the original process resumes where it was paused. Concurrent computations may, however, be executed in parallel by assigning, for example, each process to a separate processor.

5.3. Hard real-time versus soft real-time

Hard real-time means you must absolutely hit every deadline (clock-cycle). Very few systems have this requirement. Some examples are nuclear systems, some medical applications such as pacemakers, a large number of defense applications, avionics, etc.

Soft real-time systems can miss some deadlines, but eventually performance will degrade if too many are missed. A good example are the video or the sound system on your computer. If a few bits of either video or audio are missed, it is no big deal, however if too many are missed, the video and audio eventually degrades. But more importantly, in this case nobody will die if the deadline is not absolutely hit every time.

5.4. Mutex versus Semaphore

R12	Although mutexes and semaphores have similarities in their implementation, they should always be used differently.

Reasoning:

The correct use of a semaphore is for signaling from one task to another. A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects. By contrast, tasks that use semaphores either signal or wait—not both.

Below you find an example for use of mutexes:

```
/* Task 1 */
mutexWait(mutex_mens_room);
// Safely use shared resource
mutexRelease(mutex_mens_room);

/* Task 2 */
mutexWait(mutex_mens_room);
// Safely use shared resource
mutexRelease(mutex_mens_room);
```

By contrast, you should always use a semaphore like this:

```
/* Task 1 - Producer */
semPost(sem_power_btn);  // Send the signal

/* Task 2 - Consumer */
semPend(sem_power_btn);  // Wait for signal
```

More importantly, semaphores can also be used to signal from an interrupt service routine (ISR) to a task. Signaling a semaphore is a non-blocking RTOS behavior and thus ISR safe.

5.5. Priority Inversion

G7	When implementing threads and mutexes be aware of the possibility of priority inversion.

Reasoning:

The proper use of a mutex to protect a shared resource can have a dangerous unintended side effect. Any two threads that operate at different priorities and coordinate via a mutex, create the opportunity for priority inversion **[a1]**. *The risk is that a third task that does not need that mutex—but operates at a priority between the other tasks—may from time to time interfere with the proper execution of the high priority task.*

An unbounded priority inversion can spell disaster in a real-time system, as it violates one of the critical assumptions underlying the Rate Monotonic Algorithm (RMA) **[a2]**. *Since RMA is the optimal method of assigning relative priorities to real-time tasks and the only way to ensure multiple tasks with deadlines will always meet them, it is a very bad thing to risk breaking one of its assumptions. Additionally, a priority inversion in the field is a very difficult type of problem to debug, as it is not easily reproducible.*

Fortunately, the risk of priority inversion can be eliminated by changing the operating system's internal implementation of mutexes **[a3]**. *Of course, this adds to the overhead cost of acquiring and releasing mutexes.*

5.6. Deadlock

G8	When implementing threads, mutexes or semaphores be aware of deadlocks.

Reasoning:

In concurrent programming, a ***deadlock*** *is a situation in which two or more competing tasks are each waiting for the other to finish, and thus neither ever does. Suppose, as a simple example, that a computer has three CD drives and three processes. Each process holds one of these drives. If each process now requests another drive, all three processes will be in a deadlock.*

Each process will be waiting for the "CD drive released" event, which can be only caused by one *of the other waiting processes. Thus, it results in a circular chain* [**a4**], *as situation you want to avoid. A* ***deadlock*** *occurs when the following four conditions* Coffman conditions [**a5**] are met.

Appendix

[a1] http://en.wikipedia.org/wiki/Priority_inversion

[a2] http://www.barrgroup.com/Embedded-Systems/How-To/RMA-Rate-Monotonic-Algorithm

[a3] http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Priority-Inversion

[a4] http://en.wikipedia.org/wiki/Circular_reference

[**a5**] http://en.wikipedia.org/wiki/Deadlock

References

[1] *Effective Modern C++ by Scott Meyers*

[2] *Effective C++ in an Embedded Environment by Scott Meyers*

[3] *Effective STL by Scott Meyers*

[4] C++ gotchas: Avoding common problems in Coding and Design by Stephen C. Dewhurst

[5] *Coding Standards 101 rules, guidelines and best practice* by Herb Sutter and Andrei Alexandrescu.

[6] Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu

[7] C++ Primer by Stanley B. Lippman, Josee Lajoie and Barbara E. Moo

[8] *Bjarne Stroustrup's C++ Style and Technique FAQ* - http://www.stroustrup.com/bs_faq2.html

[9] *Google C++ Style Guide* **-** http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

[10] *JSF air vehicle C++ coding standard* **-** www.stroustrup.com/JSF-AV-rules.pdf

[11] *Code Standard* **-** https://isocpp.org/wiki/faq/coding-standards