Coding Standard C++ Embedded Projects

|  |  |  |
| --- | --- | --- |
| Author | Version | Date |
| Richard Peters | 5 | 22-5-2015 |

# Introduction and Scope

This coding standard defines various aspects of the contents of C++ source files. The goal is to provide a usable standard from which programmers should only deviate in very specific circumstances. Care shall be taken to not restrict programmers in any way but form, so that new techniques and new insights are not obstructed by this standard.

This standard assumes that the programmer writes Clean Code. This influences decisions, for example whether or not multiple return statements are allowed.

This standard assumes the use of Standard C++, of which the current version is published in 2014. If a compiler is used that does not support specific constructs in this coding standard, those rules are exempted from compliance.

The scope of this coding standard is limited to C++ source and header files. In e.g. C and C# other constructions and other de facto conventions exist, and the aim is not to supersede those.

Currently, the structure of files in directories is out of scope, since this is often dependent on preferences in specific tools. This may later be standardized. Furthermore, the partitioning of source files into layers like Infra, HAL, etc. is also out of scope.

Tools and libraries used are out of scope. This standard does not force a choice between tools, e.g. between Google Test and Hippomocks.

# Principles

Much of writing a standard on coding rules is just making an arbitrary decision. For each decision arguments exist why another decision would also work. However, as a complete set of decision, this standard strives to adhere to these principles:

* Consistency: At the very least, following this standard should result in consistent-looking code. This is important because our brains are wired to very quickly identify known patterns and discern structure.
* Show importance: Each line of code is constructed by more important and less important parts. Some parts should immediately capture our attention; other parts are details which should minimize interference with more important parts. An example of this is giving business-logic-level classes long names, while giving arguments short names.

# Definitions

## Use of Shall and Should

While there are exceptions to every rule, deviation of this standard should only happen in very specific circumstances where following this standard would obviously result in sub-optimal and non-Clean Code. However, not every rule is intended as a hard definition. In the cases where mere guidelines are presented, should is used, for instance “namespace names should be short”. In contrast, shall is used in cases where definitions are given, for instance “Identifiers shall not contain any prefix”.

I used shall instead of must, because it is easier to describe what code should look like than it is to describe programmer’s actions. Writing “programmers must create descriptive identifiers” is more tedious to read and write than “identifiers shall be descriptive”.

## Letter Case

Table 1 shows the different letter case forms used in identifiers:

|  |  |
| --- | --- |
| Name and Example | Notes |
| UPPER\_CASE | Words are separated by underscores. |
| lower\_case | Words are separated by underscores. |
| MixedCase | The identifier is started with an upper case character; each new word starts with an upper case character. No underscores are used. |
| camelCase | The identifier is started with a lower case character; each new word starts with an upper case character. No underscores are used. |

Table 1: Letter case definitions

# Naming Identifiers

1. Table 2 defines the letter case of identifiers:

|  |  |  |
| --- | --- | --- |
| Identifier | Letter Case | Notes |
| Variable name | camelCase | In C++, there is no separate notion of constants, since they are just variables with the const and static qualifiers. Therefore, they follow the same definition |
| Function name | MixedCase |  |
| Class name | MixedCase |  |
| Enum name | MixedCase |  |
| Enum members | camelCase | Enum members function like variables in the sense that they contain values. Therefore, the follow the Variable name definition. |
| Namespace name | lower\_case | Namespace names should be short and only consist of one word. |
| Template parameter | MixedCase | Both type and value parameters follow this definition. |
| Macro names | UPPER\_CASE |  |
| Header include guards | NAMESPACE\_UPPPER\_CASE\_HPP | Header include guards are macro names, and are therefore written in UPPER\_CASE. In addition to that, to ensure uniqueness the namespace is prepended, e.g. INFRA\_FUNCTION\_HPP. |
| File names | MixedCase.cpp/.hpp | The extension .hpp is chosen in favour of .h. Atmel Studio does not apply C++ syntax colouring to .h files. |
| Directory names | lower\_case |  |

Table 2: Identifier letter case

1. Both class names and namespace names define scope. We chose lower\_case for namespace in order to visualize the higher importance of class names: In util::ByteRange, util is less important than ByteRange.
2. File names and directory names follow the same reasoning as Rationale 1: The file name is more important than the directory name, therefore the directory name is written in lower\_case and the file name in MixedCase.
3. Macro names, and only macro names, are written in UPPER\_CASE. The C++ Preprocessor creates a special layer on top of the rest of the C++ syntax, which makes macros special in the sense that they do not follow the scoping of namespaces. Therefore, they should be easily recognizable, so UPPER\_CASE is chosen, and UPPER\_CASE is not used for any other identifier.
4. An exemption exists for member functions of classes that behave like Standard C++ container classes. Member functions of those classes shall be written in lower\_case, in order to be able to conform to requirements placed on containers by the C++ Standard.
5. Identifiers shall not contain any prefix or postfix in the form of s…, m…, \_ptr, etc. Since we write Clean Code, scope and type of identifiers should be clear from the context.
6. Parameters for which their logical name collides with the name of a class data member shall be used in the following way:   
   void C::Enable(bool enabled)  
   {  
    this->enabled = enabled;  
   }
7. Macros shall not be used to define constants.
8. enum class shall be used to define enums, in order to provide scope to the enum members. Enum members shall not contain a prefix indicating their type.
9. Abbreviations should only be used when the abbreviation is more well-known or more often used than their full name. In C++, larger identifiers are the norm, so abbreviations should be used judiciously.
10. The word used to indicate destination is spelled “for”, not “4”; the word used to indicate direction is spelled “to”, not “2”.
11. Acronyms should be considered as one word, so a UART implementation class should be named Uart, and a variable holding an object of that class uart.
12. If the UART class would be named UART, the name would look like a macro. An example of an actual collision is the case of DMA on the STM32F4xx: The name DMA actually clashes with the macro DMA defined in one of the platform’s headers.
13. Use descriptive names in favour of acronyms, when possible. Example: Queue instead of FIFO.
14. Mimic names from the C++ language when defining similar concepts. Example: Iterator instead of Cursor.
15. Identifiers shall not contain references to project names.
16. Project names are never descriptive of functionality. For example, there is nothing descriptive about “WoodstoveFanAlgorithm”. Consider “LogarithmicFanAlgorithm” or “StepwiseFanAlgorithm” instead. Moreover, source code is reused in spin-off projects; project names are either replaced by the new project’s name which takes effort, or, more likely, the old project names are kept in the identifiers which leads to confusion and legacy.

# Horizontal and Vertical Spacing

1. Indentation is a multiple of 4 spaces; tabs are not allowed.
2. Lines of code following the use of { up to the corresponding } are indented.
3. This rule applies even to the first-level namespace. Other coding standards often allow omission of indentation in the first-level namespace, but since we write Clean Code where nesting is never very deep, we can apply Rule 13 consistently.
4. The initialiser list in constructor bodies is indented. Example:  
   Timer::Timer()  
    : registered(false)  
    , nextTriggerTime()  
   {}
5. The purpose of the indentation in Rule 14 is to make clear that the values initialised are less important than the constructor’s signature.
6. In the declaration of a member template, the signature of the function is indented. If the function is defined outside of the class, it is not indented. Example:  
   class X  
   {  
    template<class T>  
    void F(); ← This line is indented  
   };  
     
   template<class T>  
   void X<T>::F() ← This line is not indented  
   {}
7. Expressions spread over multiple lines are indented one step in all lines following the first. Example:  
   a = firstObject.VeryLongFunction(withSomeParameters)  
    + secondObject.SomeOtherFunction() / localValue  
    + otherValue;
8. #ifdef, #if, #else, #elif need not be followed by indentation, although there are certainly cases where clarity would be improved by applying indentation.
9. A logical block consists of lines of code that logically belong together. Logical blocks shall not contain blank lines. Logical blocks are separated from other blocks by one blank line.
10. A blank line shall not be followed by another blank line.
11. A { belongs to the logical block beneath it, and a } belongs to the logical block above it, and they shall therefore not be succeeded resp. preceded by a blank line.
12. All #include statements form one single logical block.
13. Class members that have similar functionality, e.g. all constructors, all modifiers like push and pop, or all accessors like front and back belong to the same logical block. When a block becomes too big, it should be split up. Data members and member functions shall not be mixed in the same block. Example:  
    class A  
    {  
    public:  
     A();  
     A(const A& other);  
     A& operator=(const A& other);  
     ~A();  
      
     void DoThis(); A new block starts here, so a blank line preceeds here.  
     void DoThat(); DoThis and DoThat belong together  
      
    private:  
     int32\_t x;  
     int32\_t y;  
      
     bool done;   
    };
14. Each class definition and each function definition shall be surrounded by blank lines.
15. The elements of the inheritance list and the elements of the initialiser list shall each be placed on a separate line; the : or the , are placed in front of the element. Example:  
     class TriStatePin  
     : public InputPin  
     , public OutputPin  
     {
16. The names of variables in successive declarations shall not be aligned vertically.
17. Vertical alignment is hard to maintain; the pattern is often broken after addition of new variables, or renaming of existing variables. Moreover, restoring alignment after such an operation modifies non-related lines, which causes unnecessary long diffs and increases conflicts in version control systems.
18. The statements if, for, while, do, and switch shall be followed by a space. Example:  
    if (a) ← correct  
    if(a) ← incorrect
19. The substatements of if, else, for, while, and do shall not be written on the same line as the if, else, for, while, and do statements.
20. The case and default clauses inside a switch statement shall be indented. Example:  
    switch (a)  
    {  
     case 1: ← One indent  
     break; ← Two indents  
     default:  
     break;  
    }
21. When for, while, or do is followed by an empty substatement, {} shall be used in favour of ;. Example:  
    while (buffer[++i] != 0)  
    {}
22. When if, else, for, while, or do is followed by a single statement, {} shall be omitted, except in matching if/else if/else blocks for symmetry. Example:  
    if (a >= b)  
     return a;  
    else  
     return b;  
      
    if (a >= b)  
    {  
     a += b;  
     return a;  
    }  
    else  
    {  
     return b;  
    }
23. When writing Clean Code, many of these substatements will consist of a single statement, which often is a single function call.
24. The return type of a function should be placed on the same line as the rest of the function signature, unless the return type is very long.
25. Template specifications shall not be placed on the same line as the function to which they belong. Example:  
    template<class T>  
    void C<T>::F() ← Placed on a new line
26. Binary operators shall be surrounded by spaces. Example:  
    a = b + c; ← correct  
    a = b+c; ← incorrect
27. Unary operators shall be written adjacent to the expression. Example:  
    ++b; ← correct  
    ++ b; ← incorrect
28. In the declaration of overloaded operator functions, spaces shall be omitted. Example:  
    BigInteger& operator+=(const BigInteger&); ← correct  
    BigInteger& operator += (const BigInteger&); ← incorrect
29. In operator overloading, the name of the += operator is operator+=, as one word.

# Parentheses, Braces, Brackets, and Angle Brackets

1. return and throw are statements, not a function, therefore their argument shall not be enclosed in ( ).
2. ( shall not be followed by a space, and ) shall not be preceded by a space. Example:  
   a = (b + c) \* d; ← correct  
   a = ( b + c ) \* d; ← incorrect
3. When { and } are used on a single line to define a short array or a single statement lambda expression (see Rule 42) { shall be followed by a space and } shall be preceded by a space. Example:  
   std::array<int32\_t, 4> a = { 0, 2, 8, 64 };
4. When used in template specifications, < and > are not surrounded by spaces. Example:  
   template<class T>  
   void C<T>::F()
5. With spacing, < > are easily confused with comparison operators. Example:  
   Function0< int32\_t > a; ← Here, > looks like a comparison operator
6. ( ) shall be used where they increase clarity (in addition to, of course, where they are necessary). They shall be omitted where they do not increase clarity.
7. Operator precedence knowledge is expected of the operators listed in Table 3, which lists operators with their precedence. ( ) should therefore be omitted when Table 3 already defines the precedence. Example:  
   if (((a + b \* c < d) && e != f)  
    || a == b)  
   In this example, relative operator precedence of \* above +, < above \*, and && above != is well-known and therefore omitted. Although operator precedence of && above < is known, adding ( ) improves readability in this quite long line. Since relative precedence of && and || is not expected to be well-known, ( ) is added around the && arguments. Table 3 indicates this not well-known information by listing && and || in separate cells next to each other.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Precedence |  | Operators | |  | |  | |
| High | ++ -- | | | + - (Unary) | | | |
|  | \* / % | & | ^ | | | | | << >> |
|  | + - |
|  | < <= > >= |
|  | == != |
|  | && | | | || | | | |
| Low | = += -= \*= /= %= <<= >>= &= ^= |= | | | | | | |

Table 3: Operator Precedence

1. When long expressions are broken down over multiple lines, they shall be broken down just before an operator. Therefore, the first token on the new line shall be an operator. Example:  
   return a == b  
    && c == d;
2. When an ? : expression is broken down over multiple lines, both ? and : shall be the first token on a line. Example:  
   Colour c = todayIsMonday  
    ? SomeVeryLargeFactoryFunctionThatTranslatesColours(blue)  
    : SomeVeryLargeFactoryFunctionThatTranslatesColours(green);
3. { and } shall be the single token in one line (except when } is followed by ;), except when used to define short arrays, when used in a lambda expressions containing only a single statement, and when used in an empty definition. Example:  
   class C  
   {  
    std::array<int32\_t, 4> a = { 0, 2, 8, 64 }; ← short array  
     
    void F()  
    {  
    Schedule([this]() { F(); }); ← lambda function containing only a  
    } single statement  
     
    void Empty()  
    {} ← When nothing is placed between { and }, place them on the same line  
   };
4. Closing an empty {} on the same line communicates clearly that a class/function/loop is left empty by design. The special pattern of placing them together makes recognition instantaneous.
5. When defining a “Name tag”, where the type defined contains no functionality but only acts as a unique name, place the whole declaration on a single line. Example:  
   struct Uart: hal::Uart::Name<Uart> {};

# Miscellaneous

1. #include statements shall include standard headers in < >, and all other headers in " ". If a standard C header is needed (such as <stdlib.h>), the equivalent C++ header shall be used (<cstdlib>). Standard header shall be included after other headers, and (as second ordering criterion) includes shall be alphabetically ordered. Example:  
   #include "event/EventHandler.hpp"  
   #include "util/InterfaceConnector.hpp"  
   #include "util/Optional.hpp"  
   #include <array>  
   #include <cassert>
2. Placing standard headers after other headers minimizes the chance that a normal header gratuitously includes a standard header; any such omissions are therefore easier to detect.
3. Class data members shall be initialized by the constructor. Note that data members of class type are always initialized and therefore need no explicit initialization. Example:  
   struct C  
   {  
    C()  
    : member1(0) int32\_ts are not initialized by default and therefore require  
    {} explicit initialization  
     
    int32\_t member1;  
    int32\_t member2 = 3; If possible, in-class initialization is preferred since this  
   }; eliminates the need to initialize the member in each constructor
4. Stack variables shall be initialized at their point of declaration. Example:  
   void F()  
   {  
    int x = 0;  
    if (SomeCondition())  
    x = 5;  
    else  
    x = 10;  
   }
5. Multiple return statements are preferred in favour of more complex code. A precondition for this is that either the return is an early return on a precondition check, or the containing function is short, with logic that is easy to follow. Example:  
   int32\_t Max(int32\_t a, int32\_t b)  
   {  
    if (a >= b)  
    return a;  
    else  
    return b;  
   }
6. int32\_t shall be used in favour of int. In general, exact-width integer types should be used in favour of types like short, long, etc.
7. Exact-width integer types shall be used without their std:: namespace qualification.
8. The using namespace directive shall not be used, not even in source files.
9. Namespace qualifications improve recognition of symbols. Since source files are not owned by single persons, the same rule applies for source files for consistency.
10. NULL and 0 shall not be used as null pointers. Use nullptr instead.
11. When overriding virtual functions in a derived class, the keyword override shall be used to explicitly declare the function to override a base function, and virtual shall be omitted from the declaration.
12. The ++ and -- operators shall be written in front of the variable, unless post increment/post decrement is really the intended operation.
13. In pointer and reference declarations, the \* and & are placed adjacent to the type, not to the variable. Example:  
    int32\_t\* x; ← correct  
    int32\_t \*x; ← incorrect
14. C-style casts shall not be used. Use C++-style casts instead.
15. With a C-style cast, multiple (unexpected) casts can be done simultaneously, e.g. a static\_cast can be combined with a const\_cast:  
    (Derived\*)x; ←if the type of x was const Base\*, const is unexpectedly cast away.
16. The #pragma once declaration shall not be used, since this declaration is not part of the C++ standard.
17. Global data shall not be declared static. Instead, prefer a static private class member; place data in an unnamed namespace if a static private class member is not a proper solution.
18. In template definitions, class is used to denote template type parameters, in favour of typename. Example:  
    template<class T>  
    class X {};
19. While any type can be substituted in a template type parameter, and not just classes, the use of typename can be very misleading when it is used in a template variable parameter to denote that a nested identifier is a typename. For example:  
    template<typename StorageName, typename StorageName::Type Value>  
    class X {};  
    Here, StorageName is a type parameter, while Value is a value parameter, despite the typename in front of the type. It is used like this:  
    struct Y  
    {  
     typedef int32\_t Type;  
    };  
      
    X<Y, 5> a;  
    So even though both parameters start with typename, the first parameter takes a type while the second parameter takes a value.
20. Treat warnings as errors.
21. Warnings only help when they draw the attention. If code containing warnings is checked in, warnings will accumulate and drown out new warnings, defeating the purpose of warnings.
22. Disable all senseless warnings without hesitation.
23. The rules of the C++ language are documented in the C++ standard. In addition to these standard rules, vendors’ hallucinations about what might be unsafe are stacked on top of this, in the form of warnings that change with platform and version, and for which the cure may be worse than the disease. While warnings exist that really help improving quality, many warnings are just white noise. Examples of the latter category are warnings about unused parameters (GCC), and warnings about the use of the standard function std::copy (MSVC), with the unhelpful suggestion to use the non-portable and thus unusable make\_checked\_array\_iterator extension.  
    When in doubt, disable each warning that pops up, because our unit tests are far better suited at communicating whether your code is correct or not.
24. Platform software such as CMSIS often produces warnings out-of-the-box. Since this code is most often written in C, and since we usually do not write code in C, consider disabling all warnings for the C language.
25. When using the GMock framework, the EXPECT\_\* macros shall be used to state expectations. If a test contains any precondition checks on functionality that is already tested in other tests, the ASSERT\_\* macros shall be used to test the preconditions. Example:  
    TEST(MyTest, FirstTest)  
    {  
     EXPECT\_TRUE(theWorldIsRound);  
    }  
      
    TEST(MyTest, SecondTest)  
    {  
     ASSERT\_TRUE(theWorldIsRound); ← If this does not hold,  
     the purpose of the following test is moot  
     EXPECT\_TRUE(triangularInequalityHoldsOnEarthsSurface);  
    }

# Appendix: Cheat Sheet

1. This section contains a Cheat Sheet; i.e. nonsensical code written for the purpose of demonstrating the rules. It is not normative: no rules originate from the Cheat Sheet example.

#ifndef INFRA\_CHEAT\_SHEET\_HPP

#define INFRA\_CHEAT\_SHEET\_HPP

#include "infra\_event/Timer.hpp"

#include "infra\_util/Compare.hpp"

#include <algorithm>

namespace infra

{

class Example

{};

class CheatSheetExample

: public Example

, public Equals<CheatSheetExample>

{

public:

static const uint8\_t constant = 8;

enum class State

{

initializing,

operational

};

CheatSheetExample(uint8\_t identifier);

void Operate();

void OperateOnSomethingElse();

uint16\_t NumberOfOperationsDone() const;

template<class T>

void OperateOnType();

void operator==(const CheatSheetExample& other);

private:

uint8\_t identifier;

State state;

std::array<uint8\_t, 10> data = {};

uint8\_t\* otherData = nullptr;

};

//// Implementation ////

template<class T>

void CheatSheetExample::OperateOnType<T>()

{

identifier = sizeof(T);

}

}

#endif

Figure 1: infra\_util/CheatSheet.hpp

Figure 2: infra\_util/CheatSheet.cpp

#include "infra\_cheat/CheatSheet.hpp"

#include <cstdlib>

namespace infra

{

namespace

{

const uint8\_t extraData = 5;

}

CheatSheetExample::CheatSheetExample(uint8\_t identifier)

: identifier(identifier)

, state(State::initializing)

{}

void CheatSheetExample::Operate()

{

if (state == State::operational)

std::cout << "Operating" << std::endl;

else

std::cout << "Not operating" << std::endl;

}

void CheatSheetExample::OperateOnSomethingElse()

{

switch (state)

{

case State::initializing:

{

std::cout << "Not operating" << std::endl;

break;

}

case State::operational:

{

std::cout << "Operating" << std::endl;

break;

}

default:

std::abort();

}

for (uint8\_t& i : data)

i += 5;

}

uint16\_t CheatSheetExample::NumberOfOperationsDone() const

{

uint32\_t result = 0;

for (uint8\_t& i : data)

if (i != result)

result += i;

return extraData + (static\_cast<uint16\_t>(result) + 8) \* 13;

}

void operator==(const CheatSheetExample& other)

{

return identifier == other.identifier

&& state == other.state;

}

}

# Appendix: Settings files for development environments

This section contains settings files that instruct development environments to apply some of the coding rules.

## Visual Studio 2013

Apply this file in Visual Studio 2013 by selecting Tools -> Options, select “Import selected environment settings”, Optionally backup your current settings, then select Browse to select this settings file.



## Eclipse

Apply this file in Eclipse (Tested with Eclipse Kepler and LPCXpresso 7.3) by selecting Window -> Preferences -> C/C++ -> Code Style -> Formatter, use Import to select this settings file.

