

Modern C++ Programming

11. CODE CONVENTIONS

Federico Busato

University of Verona, Dept. of Computer Science
2020, v3.03



1 C++ Project Organization

- Project Directories
- Project Files
- src/include directories

2 Coding Styles and Conventions

- Coding Styles

3 #include and namespace

4 Variables and Preprocessing

5 Functions and Classes

6 Modern C++ Features

7 Control Flow

8 Naming and Formatting

C++ Project Organization

Project Organization

Project
Root



bin



build



doc



submodules



third_party



data



tests



examples



utils



include



src



LICENSE



README.md



CMakeLists.txt



Doxyfile



.gitignore



.clang-tidy



.clang-format

Fundamental directories

`include` Project *public* header files

`src/source` Project source files and *private* headers

`test` Source files for testing the project

Empty directories

`bin` Output executables

`build` All intermediate files

`doc` Project documentation

Optional directories

`submodules` Project submodules

`third_party` (less often `deps/external/extern`)
dependencies or external libraries

`data` Files used by the executables or for testing

`examples` Source files for showing project features

`utils` (or `script`) Scripts and utilities related to the
project

`cmake` CMake submodules (`.cmake`)

Project Files

`LICENSE` Describes how this project can be used and distributed

`README.md` General information about the project in Markdown format *

`CMakeLists.txt` Describes how to compile the project

`Doxyfile` Configuration file used by doxygen to generate the documentation (see next lecture)

others `.gitignore`, `.clang-format`, `.clang-tidy`, etc.

* Markdown is a language with a syntax corresponding to a subset of HTML tags github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

README

- README template:
`embedded-artistry-readme-template`

LICENSE

- Choose an open source license: `choosealicense.com`
- License guidelines: Why your academic code needs a software license

File extensions

Common C++ file extensions:

- **header** .h .hh .hpp .hxx
- **header implementation** .i.h, .i.hpp, -inl.h, .inl.hpp
 - (1) separate implementation from interface for inline functions and templates
 - (2) keep implementation “inline” in the header file
- **source/implementation** .c .cc .cpp .cxx

Common conventions:

- .h .c .cc **GOOGLE**
- .hh .cc
- .hpp .cpp
- .hxx .cxx

Organization:

- Public **headers** in `include`
- **source files, private headers, header implementations** in `src/source` directory
- The **main** file (if present) can be placed in `src/source` and called `main.*` or placed in the project root directory with an arbitrary name

The file should have the same name of the class/namespace that they implement

- `class MyClass`

`my_class.hpp` (`MyClass.hpp`)

`my_class.i.hpp` (`MyClass.i.hpp`)

`my_class.cpp` (`MyClass.cpp`)

- `namespace my_np`

`my_np.hpp` (`MyNP.hpp`)

`my_np.i.hpp` (`MyNP.i.hpp`)

`my_np.cpp` (`MyNP.cpp`)

Code Organization Example

- **include**

- `my_interface.hpp`

- **src**

- `my_class1.cpp`
- `my_tmpl_class.hpp`
- `my_tmpl_class.i.hpp`
(template/inline functions)
- `my_tmpl_class.cpp`
(specialization)

- **subdir1**

- `my_lib.hpp`
- `my_lib.i.hpp`
- `my_lib.cpp`

- `main.cpp` (if necessary)

- `README.md`

- `CMakeLists.txt`

- `Doxyfile`

- `LICENSE`

- **build** (empty)

- **bin** (empty)

- **doc** (empty)

- **test**

- `test1.cpp`

- `test2.cpp`

Coding Styles and Conventions

“one thing people should remember is there is what you can do in a language and what you should do”

Bjarne Stroustrup

Most important rule:

BE CONSISTENT!!

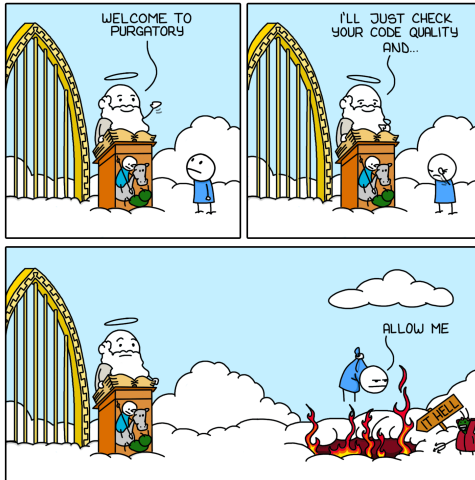
“The best code explains itself”

GOOGLE

“The worst thing that can happen to a code base is size”

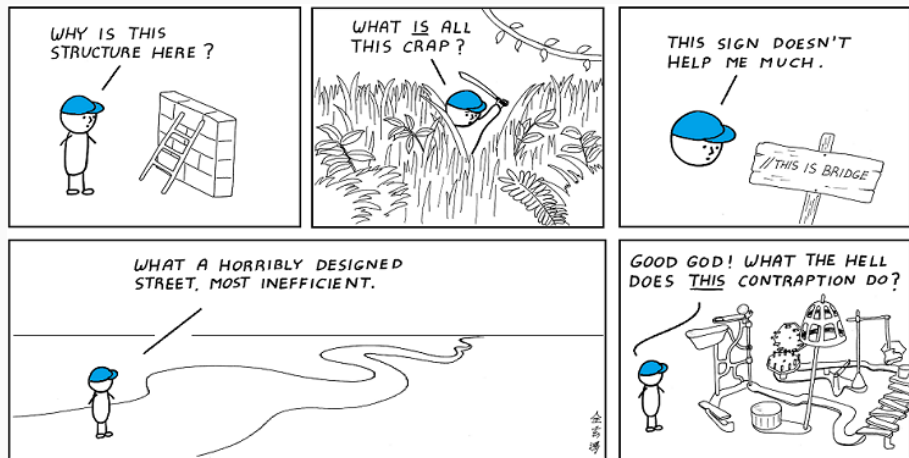
— Steve Yegge

LAST PUSH



Bad Code

How *my* code looks like for other people?



Coding styles are common guidelines to improve the *readability*, *maintainability*, prevent *common errors*, and make the code more *uniform*

Most popular coding styles:

- ***LLVM Coding Standards***

`llvm.org/docs/CodingStandards.html`

- ***Google C++ Style Guide***

`google.github.io/styleguide/cppguide.html`

- ***Webkit Coding Style***

`webkit.org/code-style-guidelines`

- ***Mozilla Coding Style***

`developer.mozilla.org`

- ***Chromium Coding Style***

`chromium.googlesource.com`

`c++-dos-and-donts.md`

- ***Unreal Engine***

`docs.unrealengine.com/en-us/Programming`

- ***μOS++***

`micro-os-plus.github.io/develop/coding-style`

`micro-os-plus.github.io/develop/naming-conventions`

※ → **Important!**

Highlight potential code issues such as bugs, inefficiency, and can compromise readability. Should not be ignored

* → **Useful**

It is not fundamental but it emphasizes good practices. Should be followed if possible

■ → **Minor / Obvious**

Style choice or not very common issue

#include and namespace

※ Every includes must be self-contained

- the project must compile with any include order
- do not rely on recursive `#include`

* Include as less as possible, especially in header files

- do not include unneeded headers
- it is not in contrast with the previous rule

LLVM, GOOGLE, CHROMIUM, UNREAL

▪ include guard vs. `#pragma once`

- Use include guard if portability is a strong requirement

GOOGLE, CHROMIUM

- `#pragma once` otherwise for performance WEBKIT, UNREAL

- `#include` preprocessor should be placed immediately **after** the *header comment* and *include guard*

LLVM

Order of #include

LLVM, GOOGLE

- (1) Main Module Header (it is only one)
- (2) Local project includes (in alphabetical order)
- (3) System includes (in alphabetical order)

System includes are self-contained, local includes might not

Project includes

LLVM, GOOGLE

- * Use `" "` syntax
- * Should be absolute paths from the project include root
e.g. `#include "directory1/header.hpp"`

System includes

LLVM, GOOGLE

- * Use `<>` syntax
e.g. `#include <iostream>`

* Use C++ headers instead of C headers:

`<cassert>` instead of `<assert.h>`

`<cmath>` instead of `<math.h>`, etc.

▪ Report at least one function used for each include

`<iostream>` `// std::cout, std::cin`

Example:

```
#include "MyClass.hpp"           // MyClass
                                [ blank line ]
#include "my_dir/my_headerA.hpp" // npA::ClassA, npB::f2()
#include "my_dir/my_headerB.hpp" // np::g()
                                [ blank line ]
#include <iostream>                // std::cout
#include <cmath>                   // std::fabs()
#include <vector>                  // std::vector
```

- ※ Avoid `using namespace`-directives at global scope
LLVM, GOOGLE, WEBKIT, UNREAL, HIC
- ※ Limit `using namespace`-directives at local scope and prefer explicit namespace specification
GOOGLE, WEBKIT
- ※ Always place code in a namespace to avoid *global namespace pollution*
GOOGLE, WEBKIT
- ※ Avoid *anonymous* namespaces in headers
GOOGLE
- Prefer *anonymous* namespaces instead of static variables
GOOGLE

Style guidelines:

- The content of namespaces is not indented

GOOGLE, WEBKIT

- Close namespace declarations

```
} // namespace <namespace_identifier>
```

LLVM

```
} // namespace (for anonymous namespaces)
```

GOOGLE

Anonymous namespaces and source files:

- Items local to a source file (e.g. .cpp) file should be wrapped in an anonymous namespace. While some such items are already file-scope by default in C++, not all are; also, shared objects on Linux builds export all symbols, so anonymous namespaces (which restrict these symbols to the compilation unit) improve function call cost and reduce the size of entry point tables

CHROMIUM

Variables and Preprocessing

- ※ Place a variables in the narrowest scope possible, and *always* initialize variables in the declaration

GOOGLE, ISOCPP, MOZILLA, HIC

- Use assignment syntax `=` when performing "simple" initialization or for constructors

CHROMIUM

- Avoid static global variables

LLVM, GOOGLE

- Declaration of pointer/reference variables or arguments may be placed with the asterisk/ampersand *adjacent* to either the *type* or to the variable *name* for all in the same way

GOOGLE

- `char* c;`

WEBKIT, MOZILLA, CHROMIUM, UNREAL

- `char *c;`

- `char * c;`

- ✱ Use fixed-width integer type (e.g. `int64_t`, `int8_t`, etc.).
Exception: `int` and `unsigned` [GOOGLE](#), [UNREAL](#)
- ✱ Use `size_t` for object and allocation sizes, object counts, array and pointer offsets, vector indices, and so on. (integer overflow behavior for signed types is undefined) [CHROMIUM](#)
- ✱ Use `int64_t` instead of `size_t` for object counts and loop indices [GOOGLE](#)
- Use brace initialization to convert arithmetic types (narrowing) e.g. `int64_t{x}` [GOOGLE](#)
- ✱ Use `true`, `false` for boolean variables instead numeric values `0`, `1` [WEBKIT](#)

- ⌘ Do not shift `<<` signed operands HIC
- ⌘ Do not directly compare floating point `==`, `<`, etc. HIC
- Do not use `auto` to deduce a raw pointer/reference. Use `auto*` / `auto&` instead

Style:

- Use floating-point literals to highlight floating-point data types, e.g. `30.0f` WEBKIT (opposite)
- Avoid redundant type, e.g. `unsigned int`, `signed int` WEBKIT

Code guidelines:

- ※ Avoid defining macros, especially in headers GOOGLE
- ※ `#undef` macros wherever possible
- ※ Prefer `const` values and `inline` functions to `#define` WEBKIT
- ※ Do not use macro for enumerator, constant, and functions
- ※ Always use curly brackets for multiline macro

```
#define MACRO    \  
{              \  
    line1;      \  
    line2;      \  
}
```


Style:

- Close `#endif` with the respective condition of the first `#if`

```
#if defined(MACRO)  
    ...  
#endif // defined(MACRO)
```

- The hash mark that starts a preprocessor directive should always be at the beginning of the line

GOOGLE

```
#if defined(MACRO)  
#    define MACRO2  
#endif
```

- Place the `\` rightmost for multilines macro

```
#define MACRO2                                \  
    macro_def...
```

- Prefer `#if defined(MACRO)` instead of `#ifdef MACRO`

Functions and Classes

- ⌘ Default arguments are allowed only on *non*-virtual functions

GOOGLE

- Prefer return values rather than output parameters

GOOGLE

- Limit overloaded functions

GOOGLE

- Do not declare functions with an excessive number of parameters. Use a wrapper structure instead

HIC

- * Passing function arguments by `const` *pointer* or *reference* if those arguments are not intended to be modified by the function

UNREAL

- Do not pass `by-const value` for built-in types, especially in the declaration (same signature of `by-value`)

- ✳ Prefer pass `by-reference` instead `by-value` except for raw arrays and built-in types

WEBKIT

- ※ Never return pointers for new objects. Use

`std::unique_ptr` instead

CHROMIUM

```
int* f() { return new int[10]; } // wrong!!  
std::unique_ptr<int> f() { return new int[10]; } // correct
```

Style guidelines:

- All parameters should be aligned if they do not fit in a single line (especially in the declaration)

GOOGLE

```
void f(int a,  
      const int* b);
```

- Parameter names should be the same for declaration and definition

CLANG-TIDY

- Do not use `inline` when declaring a function (only in the definition)

LLVM31/61

Forward declarations vs. `#includes`

- *Prefer forward declaration:* reduce compile time, less dependency

CHROMIUM

- *Prefer `#include`:* safer

GOOGLE

Code guidelines:

- ※ Objects are fully initialized by constructor call

GOOGLE, WEBKIT

- Use a `struct` only for passive objects that carry data; everything else is a `class`

GOOGLE

Minors:

- Use braced initializer lists for aggregate types `A{1, 2};`

LLVM, GOOGLE

- Do not use braced initializer lists `{}` for constructors. It can be confused with `std::initializer_list` object

LLVM

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

GOOGLE

Style guidelines:

※ Declare class data members in special way*. Examples:

- Trailing underscore (e.g. `member_var_`) GOOGLE, μ OS
- Leading underscore (e.g. `_member_var`) EDALAB, .NET
- Public members (e.g. `m_member_var`) WEBKIT

▪ Class inheritance declarations order:

`public`, `protected`, `private` GOOGLE

▪ First data members, then function members

▪ If possible, **avoid** `this->` keyword

*

- It helps to keep track of class variables and local function variables
- The first character is helpful in filtering through the list of available variables 34/61


```
struct A {           // passive data structure
    int    x;
    float  y;
};

class B {
public:
    B();
    void public_function();

protected:
    int    _a;                // in general, it is not public in
                              // derived classes
    void _protected_function(); // "protected_function()" is not wrong
                              // it may be public in derived classes

private:
    int    _x;
    float  _y;

    void _private_function();
};
```

- In the constructor, each member should be indented on a separate line, e.g.

WEBKIT, MOZILLA

```
A::A(int x1, int y1, int z1) :  
    x(x1),  
    y(y1),  
    z(z1) {
```

- *Multiple inheritance* and *virtual inheritance* are discouraged

GOOGLE, CHROMIUM

- Prefer **composition** over *inheritance*

Modern C++ Features

Use modern C++ features wherever possible

- ※ `static_cast` `reinterpreter_cast` instead of *old style cast* (type) GOOGLE, μ OS, HIC
- ※ Use `explicit` constructors / conversion operators

Use C++11/C++14/C++17 features wherever possible

- ※ Use `constexpr` instead of *macro* GOOGLE
- ※ Use `using` instead `typedef`
- ※ Prefer `enum class` instead of plain `enum` UNREAL, μ OS
- ※ `static_assert` compile-time assertion UNREAL, HIC
- ※ `lambda` expression UNREAL
- ※ `move` semantic UNREAL^{37/61}

※ `nullptr` instead of `0` or `NULL` [LLVM](#), [GOOGLE](#), [UNREAL](#)
[WEBKIT](#), [MOZILLA](#), [HIC](#)

※ Use *range-for* loops wherever possible
[LLVM](#), [WEBKIT](#), [UNREAL](#)

※ Use `auto` to avoid type names that are noisy, obvious, or unimportant
`auto array = new int[10];`
`auto var = static_cast<int>(var);` [LLVM](#), [GOOGLE](#)
lambda, iterators, template expression [UNREAL](#) (only)

▪ Use `[[deprecated]]` / `[[noreturn]]` to indicate deprecated functions / that do not return

▪ Avoid `throw()` expression. Use `noexcept` instead [HIC](#)38/61

Use **C++11/C++14/C++17** features for classes

※ Use always `override/final` function member keyword
WEBKIT, MOZILLA, UNREAL, CHROMIUM

* Use braced *direct-list-initialization* or *copy-initialization* for setting default data member value. Avoid initialization in constructors if possible
UNREAL

```
struct A {  
    int x = 3;    // copy-initialization  
    int x { 3 }; // direct-list-initialization (best option)  
};
```

▪ Prefer *defaulted* default constructor `= default`
MOZILLA, CHROMIUM

▪ Use `= delete` to mark deleted functions

Control Flow

- ✖ The `if` and `else` keywords belong on separate lines
- ✖ Each statement should get its own line

```
if (c1) <statement1>; else <statement2> // wrong!!
```

GOOGLE, WEBKIT

- Multi-lines statements and complex conditions require curly braces

GOOGLE

- Curly braces are not required for single-line statements (but allowed) (`for`, `while`, `if`)

GOOGLE

```
if (c1) { // not mandatory  
    <statement>  
}
```


- ✧ Tests for `null/non-null`, and `zero/non-zero` should all be done without equality comparisons [WEBKIT, MOZILLA](#)

```
if (!ptr)           // wrong!!  
    return;  
if (!count)         // wrong!!  
    return;
```

```
if (ptr == nullptr) // correct  
    return;  
if (count == 0)     // correct  
    return;
```

- ✧ Prefer `(ptr == nullptr)` and `x > 0` over `(nullptr == ptr)` and `0 < x`

[CHROMIUM](#)

- Boolean expression longer than the standard line length requires to be consistent in how you break up the lines [GOOGLE](#)
- Prefer `empty()` method over `size()` to check if a container has no items [MOZILLA](#)

※ Avoid redundant control flow (see next slide)

- Do not use `else` after a `return` / `break`

LLVM, MOZILLA, CHROMIUM

- Avoid `return true/return false` pattern
- Merge multiple conditional statements

※ Do not use `goto`

μOS

```
if (condition) {    // wrong!!
    < code1 >
    return;
}
else // <-- redundant
    < code2 >
//-----
if (condition) {    // Corret
    < code1 >
    return;
}
< code2 >
```

```
if (condition)    // wrong!!
    return true;
else
    return false;
//-----
return condition; // Corret
```

- Use *early exits* (`continue` , `break` , `return`) to simplify the code

LLVM

```
for (<condition1>) {    // wrong!!
    if (<condition3>)
        ...
}
//-----
for (<condition1>) {    // Correct
    if (!<condition3>)
        continue;
    ...
}
```

- Turn predicate loops into predicate functions

LLVM

```
for (<loop_condition1>) { // should be
    if (<condition2>) {    // an external
        var = ...        // function
        break;           //
    }                    //
}                        //
}                        //
```

Naming and Formatting

Spacing

- ※ Use always the same indentation style:

- tab → 2 spaces
- tab → 4 spaces
- tab = 4 spaces

GOOGLE, MOZILLA

LLVM, WEBKIT

UNREAL

- ※ Separate commands, operators, etc., by a space

LLVM, GOOGLE, WEBKIT

```
if(a*b<10&& c)           // wrong!!  
if (a * c < 10 && c)      // correct
```

- ※ Line length (width) should be at most **80 characters** long (or 120) → help code view on a terminal

LLVM, GOOGLE, MOZILLA

- Never put trailing white space or tabs at the end of a line

GOOGLE, MOZILLA

General rule:

- ✧ Use full words, except in the rare case where an abbreviation would be more canonical and easier to understand `WEBKIT`
- Avoid short and very long names

Style Conventions

Camel style Uppercase first word letter (sometimes called *Pascal style* or *Capital case*) (less readable, shorter names)

```
CamelStyle
```

Snake style Lower case words separated by single underscore (good readability, longer names)

```
snake_style
```

Macro style Upper case words separated by single underscore (sometimes called *Screaming style*) (good readability, longer names)

```
MACRO_STYLE
```


Variable Variable names should be nouns

- Camel style e.g. `MyVar` LLVM, UNREAL
- Snake style e.g. `my_var` GOOGLE, μ OS

- Constant**
- Camel style + k prefix,
e.g. `kConstantVar` GOOGLE, MOZILLA
 - Macro style e.g. `CONSTANT_VAR` WEBKIT, OPENSTACK

- Enum**
- Camel style + k
e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }` GOOGLE
 - Camel style
e.g. `enum MyEnum { EnumVar1, EnumVar2 }` LLVM, WEBKIT

- Namespace**
- Snake style, e.g. `my_namespace` GOOGLE, LLVM
 - Camel style, e.g. `MyNamespace` WEBKIT
- Typename**
- Camel style (including classes, structs, enums, typedefs, etc.)
e.g. `HelloWorldClass` LLVM, GOOGLE, WEBKIT
 - Snake style μ OS (class)

Function ✖ Should be descriptive verb (as they represent actions)

WebKit

- Use `set` prefix for modifier methods

WebKit

- Do not use `get` for observer (const) methods without parameters

WebKit

- Style:

- Lowercase Camel style, e.g. `myFunc()`

LLVM

- Uppercase Camel style for standard functions
e.g. `MyFunc()`

GOOGLE, MOZILLA, UNREAL

- Snake style for cheap functions
e.g. `my_func()`

GOOGLE, STD

Macro Macro style
e.g. MY_MACRO

GOOGLE

File

- Snake style (my_file)
- Camel style (MyFile)

GOOGLE

LLVM

Naming and Formatting Issues

※ Reserved names (do not use):

- double underscore followed by any character `__var`
- single underscore followed by uppercase `_VAR`

■ Use common loop variable names

- `i, j, k, l` used in order
- `it` for iterators

■ Prefer consecutive alignment

```
int          var1 = ...  
long long int var2 = ...
```

Naming and Formatting Issues

- ※ Use the same line ending (e.g. `'\n'`) for all files
MOZILLA, CHROMIUM
- ※ Use always the same style for braces
 - Same line
WEBKIT (others), MOZILLA
 - Its own line
UNREAL, WEBKIT (function)
MOZILLA (Class)
- * Do not use UTF characters for portability
- * Use UTF-8 encoding for portability
CHROMIUM
- Close files with a blank line
MOZILLA, UNREAL

```
int main() {  
    code  
}
```

```
int main  
{  
    code  
}
```

Maintainability and Code Documentation

Maintainability

※ Avoid complicated template programming GOOGLE

※ Use the `assert` to document preconditions and assumptions LLVM

▪ Prefer `sizeof(variable/value)` instead of `sizeof(type)` GOOGLE

▪ Avoid *RTTI* (`dynamic_cast`) or *exceptions* if possible LLVM, GOOGLE

▪ Only one space between statement and comment WEBKIT

▪ Address compiler warnings. Compiler warning messages mean something is wrong UNREAL^{54/61}

- * Any file start with a license LLVM, UNREAL
- * Each file should include
 - `@author` name, surname, affiliation, email
 - `@version`
 - `@date` e.g. year and month
 - `@file` the purpose of the filein both header and source files
- Document methods/classes/namespaces only in header files
- ※ Include `@param[in]` , `@param[out]` , `@param[in,out]` ,
`@return` tags
- The first sentence (beginning with `@brief`) is used as an abstract

- Use always the same style of comment
- Be aware of the comment style, e.g.

- Multiple lines

```
/**  
 * comment1  
 * comment2  
 */
```

- single line

```
/// comment
```

- Prefer `///` comment instead of `/* */` → allow string-search tools like `grep` to identify valid code lines

C++ Guidelines

C++ Core Guidelines

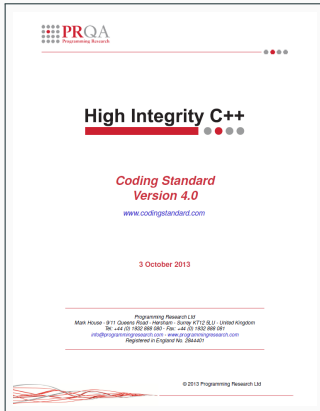
Authors: Bjarne Stroustrup, Herb Sutter



CORE GUIDELINES

The guidelines are focused on relatively high-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today

High Integrity C++ Coding Standard (HIC++)

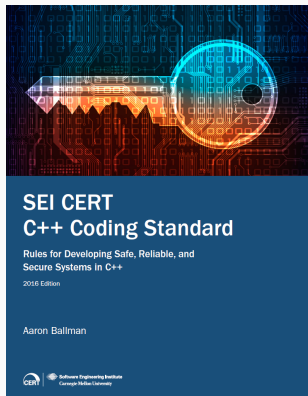


This document defines a set of rules for the production of high quality C++ code.

The guiding principles of this standard are maintenance, portability, readability and robustness

CERT C++ Secure Coding

Author: Aaron Ballman



This standard provides rules for secure coding in the C++ programming language.

The goal of these rules is to develop safe, reliable, and secure systems, for example by eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities

MISRA C++ Coding Standard



MISRA C++ provides coding standards for developing safety-critical systems.

The standard has been accepted worldwide across all safety sectors where safety, quality or reliability are issues of concern including Automotive, Industrial, Medical devices, Railways, Nuclear energy, and Embedded systems

AUTOSAR C++ Coding Standard



AUTOSAR C++ was designed as an addendum to MISRA C++:2008 for the usage of the C++14 language.

The main application sector is automotive, but it can be used in other embedded application sectors