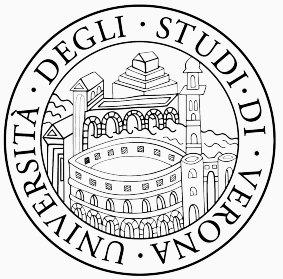
Modern C++ Programming

12. Code Conventions

Federico Busato

University of Verona, Dept. of Computer Science
2019, v2.0

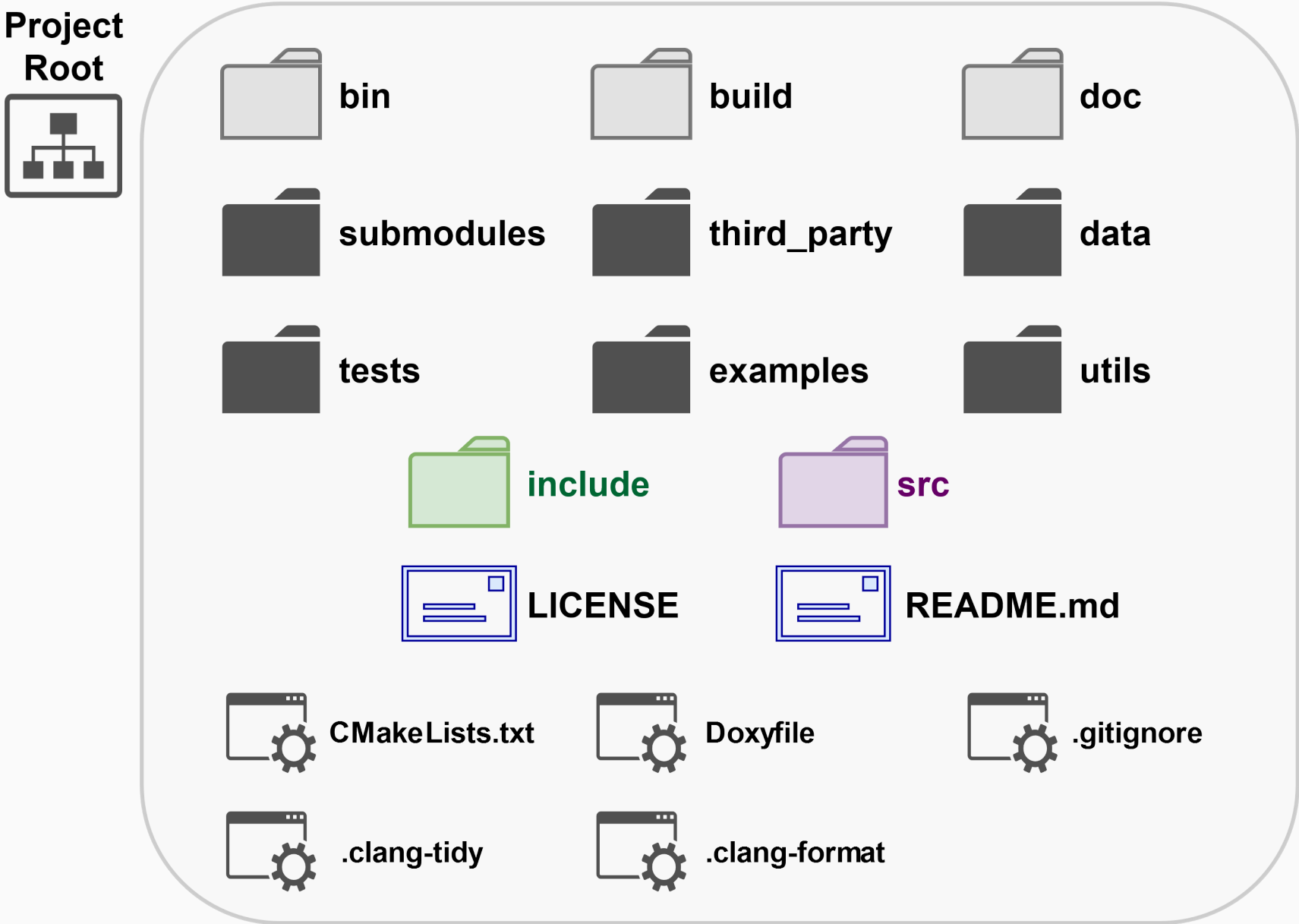


Agenda

- **Coding Style and Conventions**
  - `#include`
  - Namespace
  - Variables
  - Functions
  - Structs and Classes
  - C++11/C++14/C++17 features
  - Control Flow
- **Naming and Formatting**
  - File names and spacing
  - Issues
- **Other Issues**
  - Maintainability
  - Code documentation
- **C++ Guidelines**

C++ Project Organization

Project Organization



Project Directories

`bin`	Output executables
`build`	All intermediate files
`doc`	Project documentation
`submodules`	Project submodules
`third_party`	(less often `deps`/`external`/`extern`) dependencies or external libraries
`data`	Files used by the executables
`tests`	Source files for testing the project
`examples`	Source files for showing project features
`utils`	(or `script`) Scripts and utilities related to the project
`cmake`	CMake submodules (.cmake)

Project Files

`include`	Project header files
`src`	Project source 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`, `main.cpp` (program entry point), etc.

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

File extensions

Common C++ file extensions:

- **header** `.h .hh .hpp .hxx`
- **header implementation** `.i.h .i.hpp` EDAlab
- **src** `.c .cc .cpp .cxx`
- **textually included at specific points** `.inc` Google

Common conventions:

- `.h .c .cc` Google
- `.hh .cc`
- `.hpp .cpp`
- `.hxx .cxx`

`src/include` directories

`src/include` directories should present exactly the same directory structure

Every directory included in `src` should be also present in `include`

Organization:

- **headers** and **header implementations** in `include`
- **source files** in `src`
- The **main** file (if present) can be placed in `src` and called `main.*` or placed in the project root directory with a generic name

Common Rules

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

- `MyClass.hpp`, `MyClass.i.hpp`, `MyClass.cpp` with `class MyClass`
- `MyNP.hpp` (`my_np.hpp`),
  `MyNP.i.hpp` (`my_np.i.hpp`),
  `MyNP.cpp` (`my_np.cpp`) with `namespace my_np`

All code should be included in a **`namespace`**
$\rightarrow$ avoid *global namespace pollution*

Code Organization Example

- **include**
  - `MyClass1.hpp`
  - `MyTemplClass.hpp`
  - `MyTemplClass.i.hpp`
  - **subdir1**
    - `MyLib.hpp`
    - `MyLib.i.hpp`
      (template/inline functions)
- **src**
  - `MyClass1.cpp`
  - `MyTemplClass.cpp`
    (specialization)
  - **subdir1**
    - `MyLib.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

Most important rule:

BE CONSISTENT!!

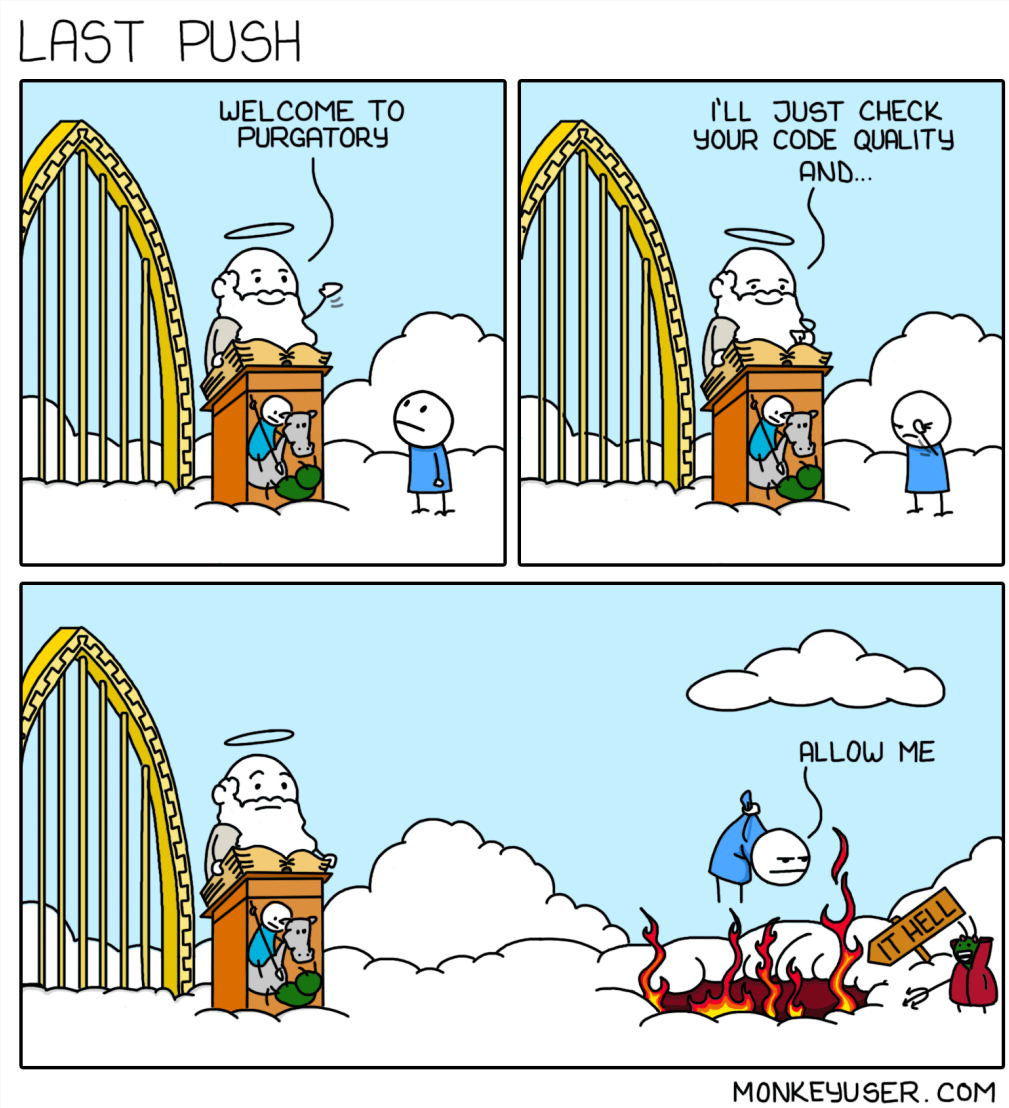
“The best code explains itself”

GOOGLE

Code Quality

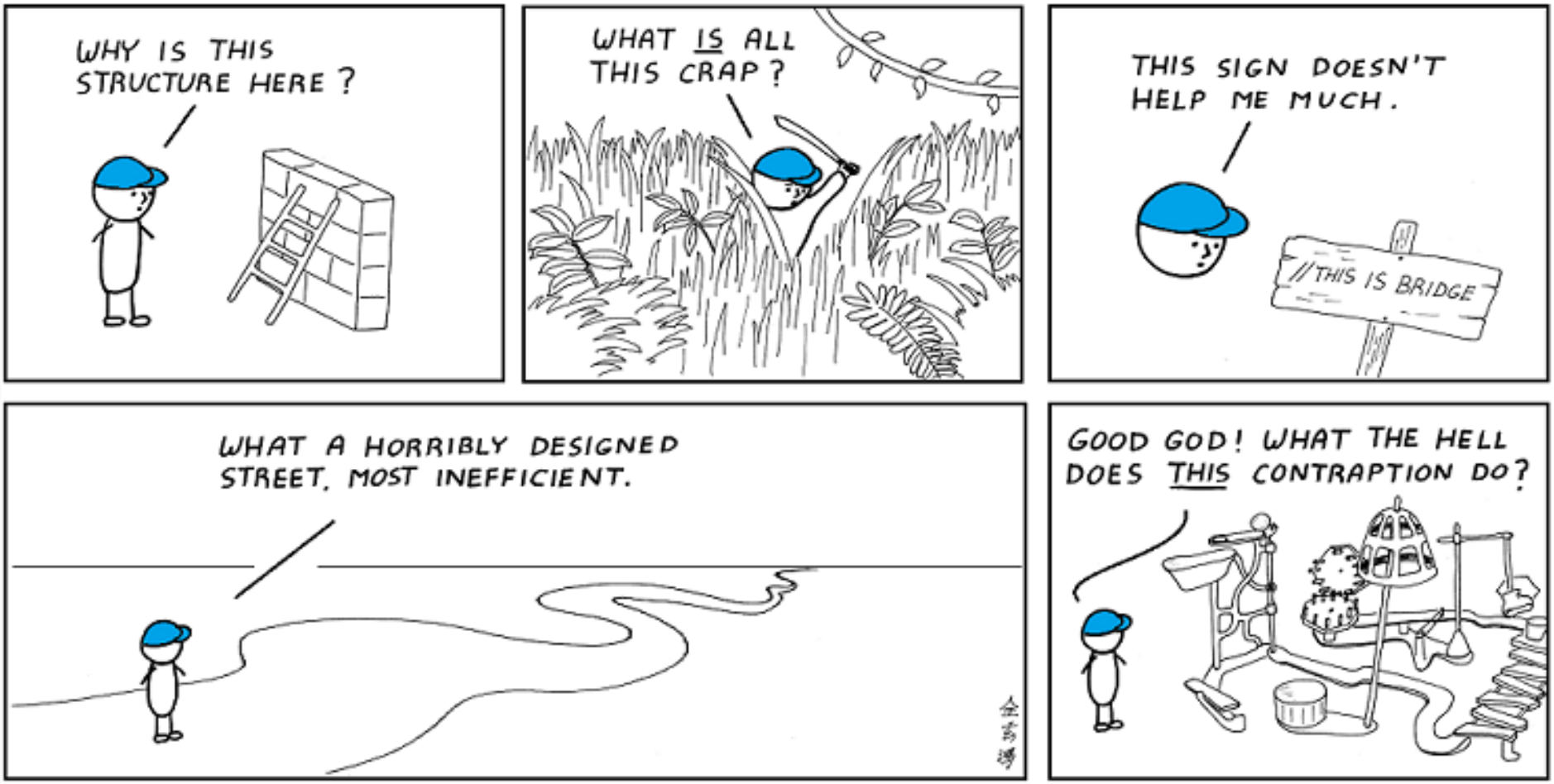
“The worst thing that can happen to a code base is size”

— *Steve Yegge*



Bad Code

How *my* code looks like for other people?



Coding Styles

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`

Minors:

- ***Webkit Coding Style*** `webkit.org/code-style-guidelines`
- ***Mozilla Coding Style*** `developer.mozilla.org`
- ***Chromium Coding Style*** `chromium.googlesource.com`

`#include` and `namespace`

- `#include` preprocessor should be placed immediately **after** the header file comment and include guards LLVM
- **Include as less as possible, especially in header files** LLVM, Google
- **Every includes must be self-contained** (the project must compile with every include order)
- Use `include guard` instead `#pragma once` Google

Order of #include — LLVM, Google

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

System includes are self-contained, local includes might not

Project includes — LLVM, Google

- should be indicated with `""` syntax
- should be absolute paths from the project include root
  e.g. `#include "directory1/header.hpp"`

System includes — LLVM, Google

- should be indicated with `<>` syntax
  e.g. `#include <iostream>`

- **Report at least one function used for each include**

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

- **Use C++ headers instead of C headers:**

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

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

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
```

Namespace guidelines:

- Avoid `using namespace`-directives at global scope
  LLVM, Google
- Limit `using namespace`-directives at local scope and prefer explicit namespace specification Google
- Always place code in a namespace Google
- Avoid *anonymous* namespaces in headers Google
- Prefer *anonymous* namespaces instead of `static` variables Google

Style guidelines:

- The content of namespaces are not indented — Google
- Close namespace declarations with
  `} // namespace <namespace_identifier>` — LLVM
- Close anonymous namespace declarations with
  `} // namespace` — Google

Entities

Variables

- Avoid static and global variables — LLVM, Google
- Place a variables in the narrowest scope possible, and initialize variables in the declaration — Google, Isocpp
- Declaration of pointer variables or arguments may be placed with the asterisk *adjacent* to either the *type* or to the variable *name* for all in the same way

  `char* c;` `char *c;` — Google
- Use fixed-width integer type (e.g. `int64_t`) — Google
- Use brace initialization to convert arithmetic types (narrowing) e.g. `int64_t{x}` — Google

Code guidelines:

- *Do not return pointers to local initialized heap memory!*
- Prefer return values rather than output parameters GOOGLE
- Limit overloaded functions GOOGLE
- Default arguments are allowed only on *non*-virtual functions GOOGLE
- Do not pass `by-const value`
- Prefer pass `by-reference` instead `by-value` except for raw arrays and built-in types

Style guidelines:

- All parameters should be aligned if possible (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 → `.i.hpp` files) LLVM

Code guidelines:

- Use a `struct` only for passive objects that carry data; everything else is a `class` Google
- Objects that are fully initialized by constructor call Google

Minors:

- Use braced initializer lists for aggregate types `A{1, 2};` LLVM, Google
- Do not use braced initializer lists for constructors LLVM
- Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors Google

Style guidelines:

- Class inheritance declarations order:
  `public`, `protected`, `private` — Google
- First data members, then function members
- Declare class data members in special way*. Examples:
  - Trailing underscore (e.g. `member_var_`) — Google
  - Leading underscore (e.g. `_member_var`) — EDAlab, .NET
  - Public members (e.g. `m_member_var`)
- **Avoid** use of `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

```
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();
};
```

Modern C++ Features

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

- Use `constexpr` instead of *macros* GOOGLE
- `static_cast` `reiterpreter_cast` instead of *old style cast* `(type)` (< C++11) GOOGLE
- Use *range-for* loops whatever possible LLVM
- 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

- `nullptr` instead `0` or `NULL` LLVM, Google
- Use `[[deprecated]]` to indicate deprecated functions
- Use `[[noreturn]]` to indicate functions that do not return
- Use `using` instead `typedef`

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

- Use *defaulted* default constructor `= default`
- Use always `override/final` function member keyword
- Use `= delete` to mark deleted functions
- Use braced *direct-list-initialization* or *copy-initialization* for setting default data member value

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

Control Flow

- Multi-lines statements and complex conditions require curly braces GOOGLE
- Boolean expression longer than the standard line length requires to be consistent in how you break up the lines GOOGLE
- Curly braces are not required for single-line statements (but allowed) (`for, while, if`) GOOGLE
- The `if` and `else` keywords belong on separate lines GOOGLE

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

if (c2) {  // required
    <statement1>
    <statement2>
}
```

```
if (complex_condition1 &&
    complex_condition2) { // required
    <statement1>
}

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

- Do not use `else` after a `return` LLVM

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

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

- Use *early exits* (`continue` , `break` , `return`) to simplify the code LLVM
- Turn predicate loops into predicate functions LLVM
- Merge multiple conditional statements

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

```
if (<condition1>) { // error!!
    if (<condition2>)
        <statement>
}

if (<condition1> && <condition2>) // correct
    <statement>
```

Naming and Formatting

Spacing

- Never use tab — LLVM, Google
  - tab → 2 spaces — Google
  - tab → 4 spaces — LLVM

- Never put trailing whitespace at the end of a line — Google

- Separate commands, operators, etc., by a space — LLVM, Google

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

- Line length (width) should be at most **80 characters** long (help code view on a terminal) — LLVM, Google

Naming Conventions

Camel style Uppercase first word letter (sometimes called *Pascal style*) (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 (good readability, longer names)

```
MACRO_STYLE
```

General rule: *avoid abbreviations and very long names*

Variable Variable names should be nouns

- Camel style e.g. `MyVar` — LLVM
- Snake style e.g. `my_var` — Google

Constant

- Camel style + `k` prefix, e.g. `kConstantVar` — Google
- Macro style e.g. `CONSTANT_VAR`

Enum

- Camel style + `k` prefix
  e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }`
  Google
- Camel style
  e.g. `enum MyEnum { EnumVar1, EnumVar2 }` — LLVM
  - prefer `enum class`

Namespace Snake style

e.g. `my_namespace` Google, LLVM

Typename Camel style (including `classes`, `structs`, `enums`, `typedefs`, etc.)

e.g. `HelloWorldClass` LLVM, Google

Function Should be verb phrases (as they represent actions)

- Lowercase Camel style, e.g. `myFunc()` LLVM
- Uppercase Camel style for standard functions

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

  e.g. `my_func()` Google, Std

Macro and Files

Macro Macro style

e.g. `MY_MACRO` — Google

- do not use macro for enumerator, constant, and functions

File

- Snake style (`my_file`) — Google
- Camel style (`MyFile`) — LLVM

Naming and Formatting Issues

- **Reserved names**:
  - 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

- Use `true`, `false` for boolean variables instead numeric value `0, 1`

- Prefer consecutive alignment

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

Naming and Formatting Issues

- Use the same line ending (e.g. `'\n'`) for all files
- Use `UTF-8` encoding for portability
- Close files with a blank line
- The hash mark that starts a preprocessor directive should always be at the beginning of the line GOOGLE

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

Other Issues

Maintainability

- Avoid defining macros, especially in headers — Google
- `#undef` macros wherever possible
- Prefer `sizeof(variable/value)` instead of `sizeof(type)` — Google
- Avoid complicated template programming — Google
- Use the **`assert`** to document preconditions and assumptions — LLVM
- Do not use *RTTI* (`dynamic_cast`) or *exceptions* — LLVM, Google

- Each file should start with a license LLVM
- Each file should include
  - `@author` name, surname, affiliation, email
  - `@version`
  - `@date` e.g. year and month
  - `@file` the purpose of the file

  in both header and source files

- Use always the same style of comment
- Comment methods/classes/namespaces only in header files
- Be aware of the comment style, e.g.
  - Multiple lines

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

  - single line

```
/// comment
```

- The first sentence (beginning with `@brief`) is used as an abstract
- Include `@param[in]`, `@param[out]`, `@param[in,out]`, `@return` tags

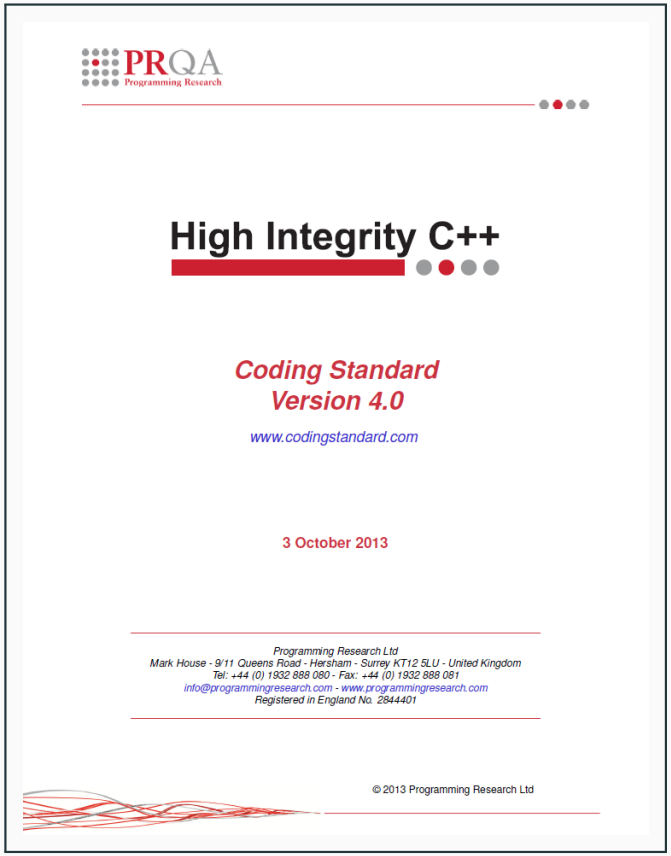
C++ Guidelines

C++ Core Guidelines

`Authors:` Bjarne Stroustrup, Herb Sutter

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++)


PRQA
Programming Research

High Integrity C++

Coding Standard
Version 4.0

www.codingstandard.com

3 October 2013

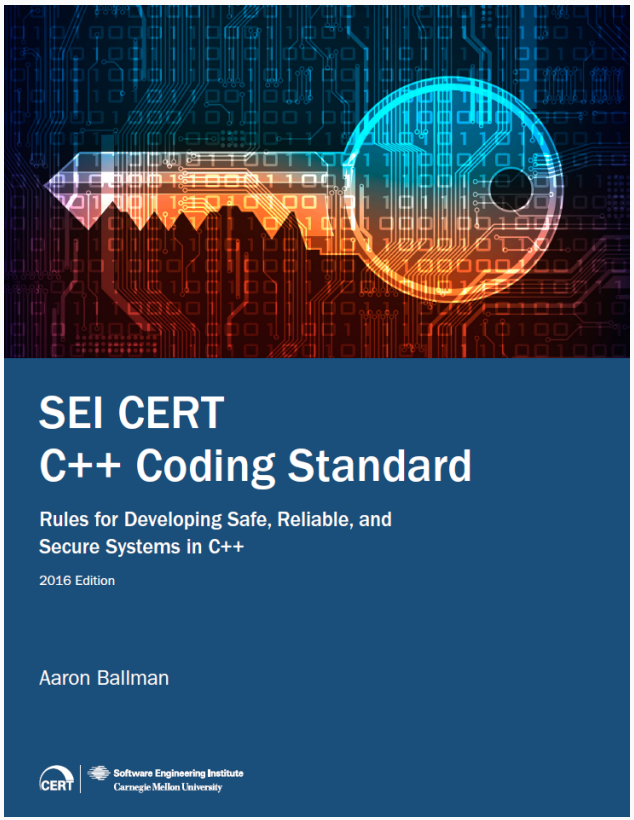
Programming Research Ltd
Mark House - 9/11 Queens Road - Hersham - Surrey KT12 5LU - United Kingdom
Tel: +44 (0) 1932 888 080 - Fax: +44 (0) 1932 888 081
info@programmingresearch.com - www.programmingresearch.com
Registered in England No. 2844401

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

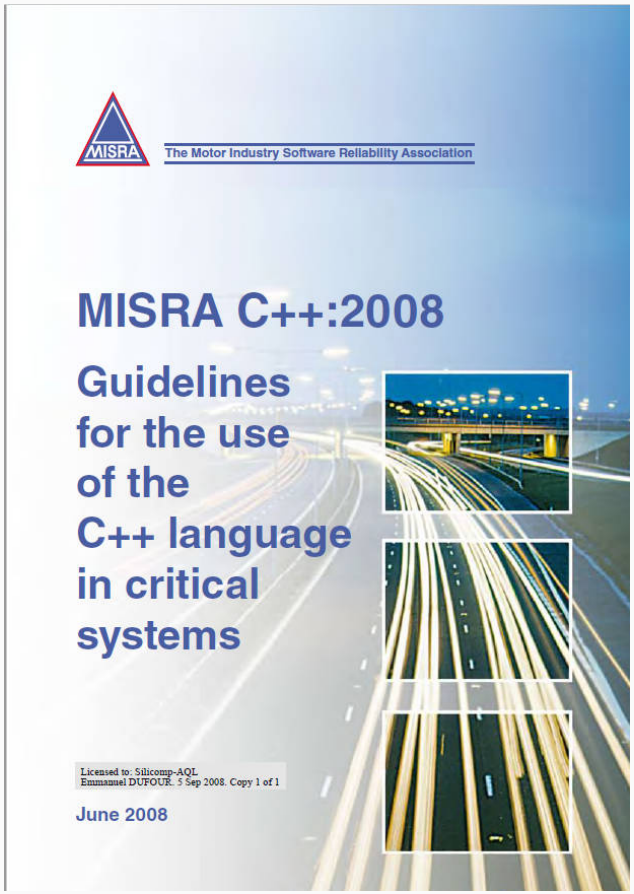
CERT C++ Secure Coding

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