Modern C++ Programming

11. Code Conventions

Federico Busato

University of Verona, Dept. of Computer Science

2021, v3.04

Table of Context

1. **C++ Project Organization**
   - Project Directories
   - Project Files
   - `src/include` directories
2. **Coding Styles and Conventions**
   - Coding Styles
3. **`#include`**

Table of Context

4 Macro and Preprocessing

5 `namespace`

6 Variables

7 Functions

8 Structs and Classes

Table of Context

9 Control Flow

10 Modern C++ Features

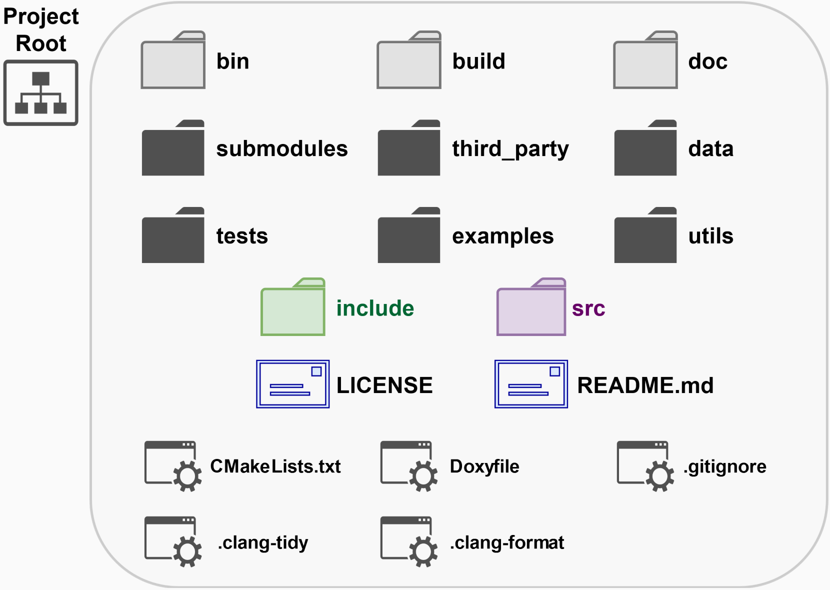
11 Maintainability

12 Naming and Formatting

13 Code Documentation

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

Project Directories 1/2

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

Project Directories 1/2

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 and License

README

- README template:
  - Embedded Artistry README Template
  - Your Project is Great, So Let's Make Your README Great Too

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`

`src/include` directories

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

Common Rules

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_templ_class.hpp`
  - `my_templ_class.i.hpp` (template/inline functions)
  - `my_templ_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

Overview

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

Bjarne Stroustrup

Overview

Most important rule:

BE CONSISTENT!!

“The best code explains itself”

Google

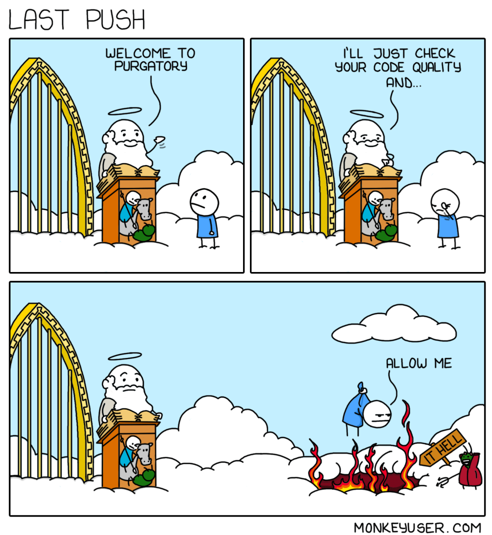
Overview

> *“80% of the lifetime cost of a piece of software goes to maintenance”*
>
> **Unreal Engine**

Code Quality

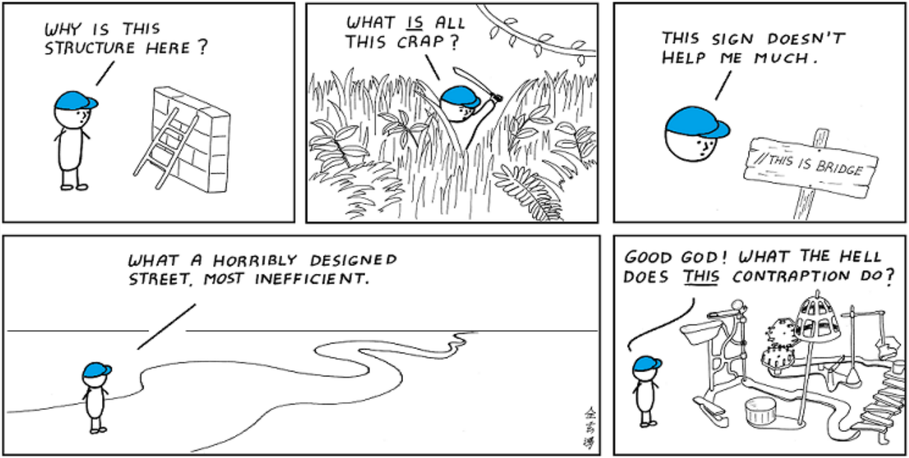
"The worst thing that can happen to a code base is size"

— ***Steve Yegge***



Bad Code

How *my* code looks like for other people?


WHY IS THIS STRUCTURE HERE ?
WHAT IS ALL THIS CRAP ?
THIS SIGN DOESN'T HELP ME MUCH.
//THIS IS BRIDGE
WHAT A HORRIBLY DESIGNED STREET. MOST INEFFICIENT.
GOOD GOD! WHAT THE HELL DOES THIS CONTRAPTION DO?

abstrusegoose.com/432

Coding Styles 1/3

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

- ***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***
  `firefox-source-docs.mozilla.org`

Coding Styles 2/3

- ***Chromium Coding Style***
  `chromium.googlesource.com`
  `c++-dos-and-donts.md`

- ***Unreal Engine - Coding Standard***
  `docs.unrealengine.com/en-us/Programming`

- μ***OS++***
  `micro-os-plus.github.io/develop/coding-style`
  `micro-os-plus.github.io/develop/naming-conventions`

- ***High Integrity C++ Coding Standard***
  `www.perforce.com/resources`

- ***CERT C++ Secure Coding***
  `wiki.sei.cmu.edu`

Coding Styles 3/3

More comprehensive code guidelines

- ***C++ Guidelines***
  `isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`

Critical system coding standards

- ***Misra - Coding Standard***
  `www.misra.org.uk`
- ***Autosar - Coding Standard***
  `www.misra.org.uk`
- ***Joint Strike Fighter Air Vehicle***
  `www.perforce.com/blog/qac/jsf-coding-standard-cpp`

Legend

※ → **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

#include 1/5

※ **Every includes must be self-contained**

- include every header you need directly
- the project must compile with any include order
- do not rely on recursive `#include`

LLVM, Google, Unreal, μOS++, Core

∗ **Include as less as possible, especially in header files**

- do not include unneeded headers
- minimize dependencies
- minimize code in headers (e.g. use forward declarations)
- it is not in contrast with the previous rule

LLVM, Google, Chromium, Unreal, Hic, μOS++

Order of #include LLVM, WebKit, Core

(1) Main Module Header (it is only one)

- space

(2) Local project includes (in alphetical order)

- space

(3) System includes (in alphetical order)

Note: **(2)** and **(3)** can be inverted Google
System includes are self-contained, local includes might not

#include 3/5

Project includes LLVM, Google, WebKit, Hic, Core

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

System includes LLVM, Google, WebKit, Hic

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

#include

- `include guard` vs. `#pragma once`
  - Use `include guard` if portability is a strong requirement LLVM, Google, Chromium, Core
  - `#pragma once` otherwise WebKit, Unreal
- `#include` preprocessor should be placed immediately after the header comment and include guard LLVM

Forward declarations vs. #includes

- *Prefer forward declaration*: reduce compile time, less dependency Chromium
- *Prefer* `#include`: *safer* Google

#include

- **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
```

Macro and Preprocessing

Macro and Preprocessing 1/3

- ※ **Avoid defining macros**, especially in headers GOOGLE
  - Do not use macro for enumerators, constants, and functions WEBKIT, GOOGLE
- ※ **Use a prefix for all macros** related to the project `MYPROJECT_MACRO` GOOGLE, UNREAL
- ※ `#undef` **macros wherever possible** GOOGLE
  - Even in the source files if *unity build* is used

Macro and Preprocessing

※ **Always use curly brackets for multilines macro**

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

※ **Always put macros after `#include`** Hic

- Put macros outside namespaces

Macro and Preprocessing 3/3

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`

`namespace`

Namespaces

※ **Avoid `using namespace`-directives at global scope**
LLVM, Google, WebKit, Unreal, Hic, μOS++

∗ **Limit `using namespace`-directives at local scope** and prefer explicit namespace specification
Google, WebKit, Unreal

※ **Always place code in a namespace** to avoid *global namespace pollution* Google, WebKit

∗ **Avoid *anonymous* namespaces in headers** Google, Cert

- `anonymous namespace` vs. `static`
  - Prefer *anonymous* namespaces instead of `static` variables/functions Google, Core
  - Use *anonymous* namespaces only for inline class declaration, `static` otherwise LLVM, static

Namespaces

Style guidelines:

- The content of namespaces is not indented

  LLVM, 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, Core, Hic

Variables

Variables 1/3

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

  Google, Isocpp, Mozilla, Hic, *mu*OS, Cert

- ∗ **Avoid static (non-const) global variables**

  LLVM, Google, Core, Hic

- Use assignment syntax `=` when performing “simple” initialization Chromium

- 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

  WebKit, Mozilla, Chromium, Unreal

  - `char* c;`
  - `char *c;`
  - `char * c;`

Variables 2/3

- ※ **Use fixed-width integer type** (e.g. `int64_t`, `int8_t`, etc.). Exception: `int` and `unsigned` Google, Unreal

- * `size_t` vs. `int64_t`
  - 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 (constant) arithmetic types (narrowing) e.g. `int64_t{x}` Google

- * Use `true`, `false` for boolean variables instead numeric values `0, 1` WebKit

Variables 3/3

※ **Do not shift** `≪` **signed operands** Hic, Core, μOS

※ **Do not directly compare floating point** `==`, `<`, etc. Hic

※ **Use signed types for arithmetic** Core

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

Functions

Functions 1/3

∗ **Limit overloaded functions**. Prefer default arguments

Google, Core

∗ **Split up large functions** into logical sub-functions for improving readability and compile time

Unreal, Google, Core

▪ Use `inline` only for small functions (e.g. $<$ 10 lines)

Google, Hic

※ **Never return pointers for new objects**. Use `std::unique_ptr` instead

Chromium, Core

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

Functions - *Parameters* 2/3

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

∗ **Pass 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 returning values** rather than output parameters GOOGLE

∗ **Do not declare functions with an excessive number of parameters**. Use a wrapper structure instead HIC, CORE

Functions - *Style* 3/3

- 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) LLVM

- Do not separate declaration and definition for template and inline functions GOOGLE

Structs and Classes

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

- ※ **Objects are fully initialized by constructor call** GOOGLE, WEBKIT, CORE

- ∗ **Prefer in-class initializers to member initializers** CORE

- Use delegating constructors to represent common actions for all constructors of a class CORE

- Initialize member variables in the order of member declaration CORE, HIC

Structs and Classes

- * **Do not define implicit conversions**. Use the `explicit` keyword for conversion operators and constructors
  GOOGLE, CORE
- Prefer `= default` constructors over user-defined / implicit default constructors MOZILLA, CHROMIUM, CORE, HIC
- Use `= delete` to mark deleted functions CORE, HIC
- Mark destructors `noexcept` CORE
- 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

Structs and Classes - *Inheritance* 3/6

- ※ **Avoid virtual method calls in constructors**
  GOOGLE, CORE, CERT

- ※ **Default arguments are allowed only on *non-virtual* functions** GOOGLE, CORE, HIC

- ∗ ***Multiple inheritance* and *virtual inheritance* are discouraged** GOOGLE, CHROMIUM

- ∗ **Prefer *composition* over *inheritance*** GOOGLE

- ∗ **A polymorphic class should suppress copying** CORE

- ∗ **A class with a *virtual function* should have a *virtual or protected destructor*** (e.g. interfaces and abstract classes)
  CORE

Structs and Classes - *Style* 4/6

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

- Trailing underscore (e.g. `member_var_`) GOOGLE, μOS, CHROMIUM
- Leading underscore (e.g. `_member_var`) .NET
- Public members (e.g. `m_member_var`) WEBKIT

- Class inheritance declarations order: **`public`**, **`protected`**, **`private`** GOOGLE, μOS

- 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

Structs and Classes - *Style* 5/6

```
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();
};
```

Structs and Classes - *Style* 6/6

- 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) {
```

Control Flow

Control Flow 1/6

- ※ **Avoid redundant control flow** (see next slide)
  - Do not use `else` after a `return` / `break`
    LLVM, Mozilla, Chromium, WebKit
  - Avoid `return true/return false` pattern
  - Merge multiple conditional statements

- ∗ **Prefer `switch` to multiple `if`-statement** Core

- Avoid `do-while` loop Core

- Avoid `goto` μOS, Core

- Do not use default labels in fully covered switches over enumerations LLVM

Control Flow - *if/else* 2/6

```
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
```

Control Flow - *Loops* 3/6

- 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;             //
   }                      //
}                         //
```

Control Flow - *Comparison* 4/6

※ **Tests for `null/non-null`, and `zero/non-zero` should all be done with equality comparisons** CORE, WEBKIT

(opposite) 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

- Do not compare to `true/false`, e.g. `if (x == true)`

Control Flow

※ **Do not mix `signed` and `unsigned` types** HIC

- Prefer signed integer (better 64-bit) for loop indices CORE
- Prefer `enum` to `bool` on function parameters
- Prefer `empty()` method over `size()` to check if a container has no items MOZILLA
- Ensure that all statements are reachable HIC

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

Control Flow - *Style* 6/6

※ **The `if` and `else` keywords belong on separate lines**

```
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, WEBKIT

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

- Boolean expression longer than the standard line length requires to be consistent in how you break up the lines GOOGLE

Modern C++ Features

Modern C++ Features 1/4

Use modern C++ features wherever possible

- `static_cast` `reinterpret_cast` instead of *old style cast* `(type)` Google, μOS, Hic

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

Modern C++ Features - *C++11/14/17* 2/4

- ※ **Use `constexpr` instead of *macro*** GOOGLE, WEBKIT
- ※ **Use `using` instead `typedef`**
- ※ **Prefer `enum class` instead of plain `enum`** UNREAL, μOS
- ※ **`static_assert` compile-time assertion** UNREAL, HIC
- ※ **`lambda` expression** UNREAL
- ※ **`move` semantic** UNREAL
- ※ **`nullptr` instead of `0` or `NULL`**
  LLVM, GOOGLE, UNREAL, WEBKIT, MOZILLA, HIC, μOS

Modern C++ Features - *C++11/14/17* 3/4

※ **Use *range-for* loops whatever possible**

LLVM, WebKit, Unreal, Core

※ **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]]` / `[[nodiscard]]` to indicate deprecated functions / that do not return / result should not be discarded

▪ Avoid `throw()` expression. Use `noexpect` instead Hic

Modern C++ Features for Classes 4/4

- ※ **Use always `override/final` function member keyword**

  WebKit, Mozilla, Unreal, Chromium, Hic

- ∗ **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)
};
```

- Use `= default` constructors
- Use `= delete` to mark deleted functions
- Prefer *uniform initialization* when it cannot be confused with `std::initializer_list`

  Chromium

Maintainability

Readability

- ※ **Write all code in English**
- ※ **Avoid complicated template programming** GOOGLE
- ※ **Use the `assert` to document preconditions and assumptions** LLVM
- ∗ **Use symbolic names** instead of literal values in code HIC
- ※ **Do not overload operators with special semantics `&&`** HIC

Readability

- Prefer consecutive alignment

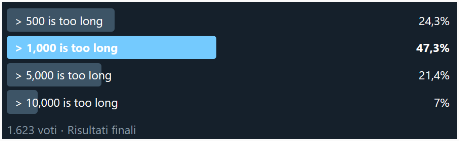
```
int           var1 = ...
long long int var2 = ...
```

∗ **Write self-documenting code**, e.g. `(x + y - 1) / y` → `ceil_div(x, y)` UNREAL

- Minimize the number of empty rows

- Do not use more than one empty line GOOGLE

- Do not write excessive long file



> 500 is too long	24,3%
> 1,000 is too long	**47,3%**
> 5,000 is too long	21,4%
> 10,000 is too long	7%

1.623 voti · Risultati finali

`What is your threshold for a long source file?`

Spacing

- Use always the same style for braces
  - Same line WebKit (func. only), Mozilla
  - Its own line Unreal, WebKit (function)
    Mozilla (Class)

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

```
int main
{
    code
}
```

Spacing

※ **Use always the same indentation style**

- tab → 2 spaces — Google, Mozilla, Hic, μOS
- tab → 4 spaces — LLVM, Webkit, Hic, μOS
- tab = 4 spaces — Unreal

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

LLVM, Google, WebKit

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

※ **Limit line length (width)** to be at most **80 characters** long (or 120) → help code view on a terminal

LLVM, Google, Mozilla, μOS

Maintainability

- ※ **Do not use `reinterpret_cast` or `union` for type punning** Core, Hic
- Address compiler warnings. Compiler warning messages mean something is wrong Unreal
- Ensure ISO C++ compliant code and avoid non-standard extension, deprecated features, or asm declarations, e.g. `register`, `__attribute__` Hic
- Prefer `sizeof(variable/value)` instead of `sizeof(type)` Google
- Enforce const-correctness Unreal

Naming and Formatting

Naming Conventions

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
```

Entity Names 1/2

Variable Variable names should be nouns

- Camel style e.g. `MyVar` LLVM, Unreal
- Snake style e.g. `my_var` Google, μOS

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

Entity Names 2/2

Namespace

- Snake style, e.g. `my_namespace` GOOGLE, LLVM
- Camel style, e.g. `MyNamespace` WEBKIT

Typename Should be nouns

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

  e.g. `HelloWorldClass` LLVM, GOOGLE, WEBKIT
- Snake style μOS (class)

Functions

- **Should be descriptive verb** (as they represent actions) WebKit

- **Functions that return boolean values should start with boolean verbs**, like `is, has, should, does` μOS

- 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 and Files

Macro Macro style

e.g. `MY_MACRO` Google

File

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

Other Naming Issues

※ **Do not use reserved names** Cert

  - 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

Other Formatting Issues 1/2

- Never put trailing white space or tabs at the end of a line Google, Mozilla
- Declare each identifier on a separate line in a separate declaration Hic
- Only one space between statement and comment WebKit

Other Formatting Issues 2/2

- ※ **Use the same line ending** (e.g. `'\n'`) for all files MOZILLA, CHROMIUM
- ∗ **Do not use UTF characters for portability**, prefer ASCII
- ∗ If UTF is needed, **prefer `UTF-8` encoding for portability** CHROMIUM
- Close files with a blank line MOZILLA, UNREAL

Code Documentation

Code Documentation 1/2

- **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 file

  in both header and source files

- Document methods/classes/namespaces only in header files

- Include `@param[in]`, `@param[out]`, `@param[in,out]`, `@return` tags

- Document ranges, impossible values, status/return values meaning Unreal

Code Documentation 2/2

- 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

  Hic, μOS

- Use anchors for indicating special issues: `TODO`, `FIXME`, `BUG`, etc.

  WebKit, Chromium