Modern C++ Programming

12. Code Conventions

Federico Busato

2023-04-02

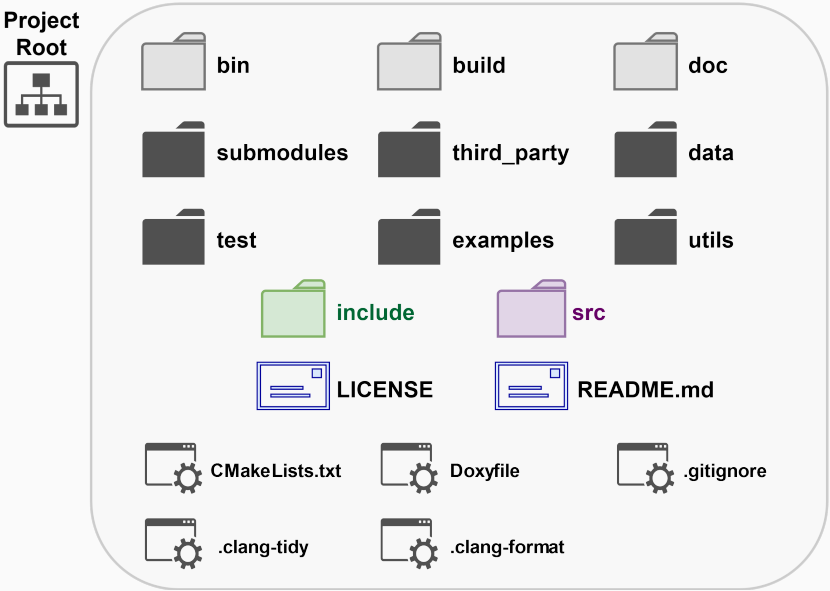
Table of Context

Table of Context

Table of Context

C++ Project Organization

(Common) Project Organization



Fundamental directories

`include` Project *public* header files

`src` Project source files and *private* headers

`test` (or `tests`) Source files for testing the project

Empty directories

`bin` Output executables

`build` All intermediate files

`doc` (or `docs`) Project documentation

Optional directories

`submodules`	Project submodules
`third_party`	(less often `deps`/`external`/`extern`) dependencies or external libraries
`data`	(or `extras`) Files used by the executables or for testing
`examples`	Source files for showing project features
`utils`	(or `tools`, 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.md

- 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

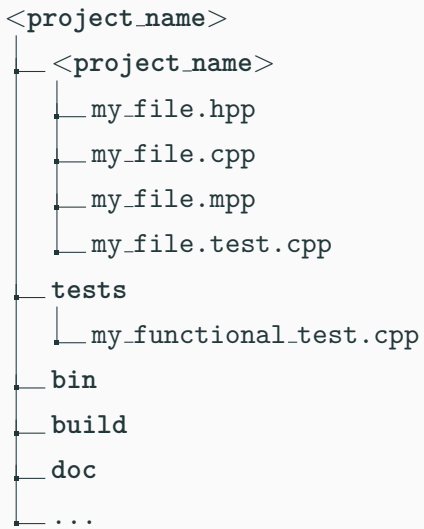
The file should have the same name of the class/namespace that they implement

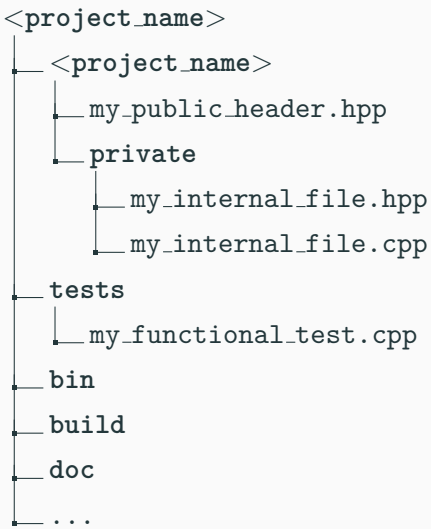
- `class MyClass`
  `my_class.hpp` (`MyClass.hpp`)
  `my_class.i.hpp` (`MyClass.i.hpp`)
  `my_class.cpp` (`MyClass.cpp`)

- `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)
- **tests**
  - test1.cpp
  - test2.cpp


<project_name>
<project_name>
my_file.hpp
my_file.cpp
my_file.mpp
my_file.test.cpp
tests
my_functional_test.cpp
bin
build
doc
...


<project_name>
<project_name>
my_public_header.hpp
private
my_internal_file.hpp
my_internal_file.cpp
tests
my_functional_test.cpp
bin
build
doc
...

- *Header* and *source files* (or *module interface* and *implementation files*) are next to each other (no `include/` and `src/` split)
- *Headers* are included with `<>` and contain the project directory prefix, for example, `<hello/hello.hpp>` (no need of `""` syntax)
- *Header* and *source file* extensions are `.hpp` / `.cpp` (`.mpp` for module interfaces). No special characters other than `_` and `-` in file names with `.` only used for extensions
- A source file that implements a *module's unit tests* should be placed next to that *module's files* and be called with the module's name plus the `.test` second-level extension
- A project's functional/integration tests should go into the tests/ subdirectory

References

- Kick-start your C++! A template for modern C++ projects
- The Pitchfork Layout
- Canonical Project Structure

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

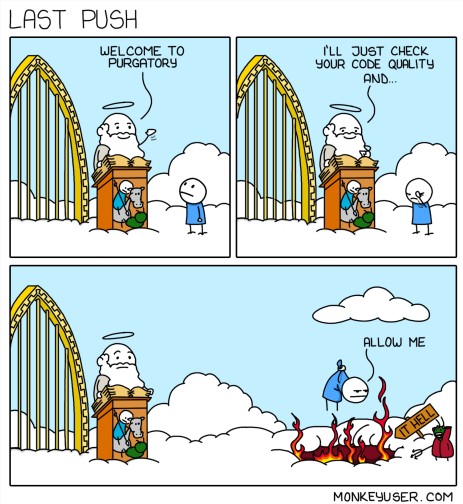
"80% of the lifetime cost of a piece of software goes to maintenance"

Unreal Engine

Code Quality

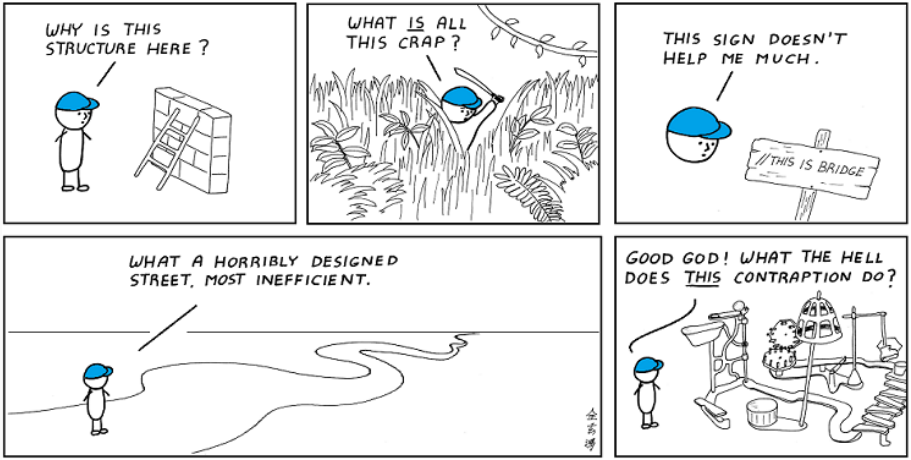
“The worst thing that can happen to a code base is size”

— Steve Yegge



Bad Code

How *my* code looks like for other people?



abstrusegoose.com/432

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`

- ***Chromium Coding Style***
  `chromium.googlesource.com`
  `c++-dos-and-donts.md`
- ***Unreal Engine - Coding Standard***
  `docs.unrealengine.com/en-us/Programming`
- ***μOS++***
  `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`

More educational-oriented 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 and can help to prevent bugs. Should be followed if possible

▪ → **Minor / Obvious**
Style choice or not very common issue

#include

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

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

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)

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

Order of #include LLVM, WebKit, Core

(1) Main module/interface header, if exists (it is only one)

- space

(2) Local project includes (in lexicographic order)

- space

(3) System includes (in lexicographic order)

Note: **(2)** and **(3)** can be swapped Google

System includes are self-contained, local includes might not

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>`

※ **Always use an include guard**

- `macro include guard` vs. `#pragma once`
  - Use `macro include guard` if portability is a very 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

* **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`

```
#include "my_class.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 <cmath>                   // std::fabs()
#include <iostream>                // std::cout
#include <vector>                  // std::vector
```

Macro and Preprocessing

※ **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

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

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

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

- Put macros outside namespaces

- 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

※ **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

* **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

- The content of namespaces is not indented — LLVM, Google, WebKit
- Close namespace declarations
  - `} // namespace <namespace_identifier>` — LLVM
  - `} // namespace` (for anonymous namespaces) — Google

Variable

※ **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

※ **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

- ※ **Do not shift $\ll$ 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

* **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
```

- ※ **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

- Prefer `enum` to `bool` on function parameters
- 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

* **Initialize member variables in the order of member declaration** Core, Hic

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

* **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` for 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

※ **Avoid virtual method calls in constructors** Google, Core, Cert

※ **Default arguments are allowed only on *non-virtual* functions**
Google, Core, Hic

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

- Does not use `virtual` with `final/override` (implicit)

see A hole in Clang's -Wsuggest-override

* ***Multiple inheritance*** **and** ***virtual inheritance*** **are discouraged**
  GOOGLE, CHROMIUM
* **Prefer** ***composition*** **over** ***inheritance***
  GOOGLE
* **A polymorphic class should suppress copying**
  CORE

- **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

```
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} {
```

Control Flow

※ **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 `goto`** μOS, Core

▪ Avoid `do-while` loop Core

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

```
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 (<condition2>)
        ...
}
//-----------------------------
for (<condition1>) {    // Correct
    if (!<condition2>)
        continue;
     ...
}
```

- Turn predicate loops into predicate functions LLVM

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

※ **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)`

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

∗ **Prefer signed integer for loop indices** (better 64-bit) CORE

- 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

※ **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

```
if (c1 && ... &&
    c2 && ...) {  // correct
    <statement>
}
```

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

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

Modern C++ Features

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

- **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

※ **Use *range-based 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

※ **Always use `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

※ **Write all code in English**, comments included

※ **Avoid complicated template programming** GOOGLE

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

∗ **Use symbolic names** instead of literal values in code HIC

```
double area1 = 3.14 * radius * radius; // wrong!!
```

```
constexpr auto Pi    = 3.14; // correct
double         area2 = Pi * radius * radius;
```

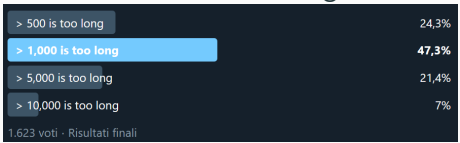
* **Prefer consecutive alignment**

```
int           var1     = ...
long long int longvar2 = ...
```

- Minimize the number of empty rows
- Do not use more than one empty line

GOOGLE

* Do not write excessive long file



What is your threshold for a long source file?

* **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
}
```

- 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;`

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

- **Do not use `reinterpret_cast` or `union` for type punning** — Core, Hic
- **Enforce const-correctness** — Unreal
- **Do not overload operators with special semantics `&&`, `^`** — Hic
- **Use `assert` to document preconditions and assumptions** — LLVM

* **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

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
```

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

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 methods (`const`) without parameters, e.g. (`size()`) — 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

- 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

* **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

* **Any file start with a license** LLVM, Unreal

* **Each file should include**
  - `@author` name, surname, affiliation, email
  - `@date` e.g. year and month
  - `@file` the purpose of the file

  in both header and source files

- Document each entity (functions, classes, namespaces, definitions, etc.) and only in the declarations, e.g. header files

- The first sentence (beginning with `@brief`) is used as an abstract
- Document the input/output parameters `@param[in]`, `@param[out]`, `@param[in,out]`, return value `@return`, and template paramenters `@tparam`
- Document ranges, impossible values, status/return values meaning — Unreal
- Use always the same style of comment
- Use anchors for indicating special issues: `TODO`, `FIXME`, `BUG`, etc. — WebKit, Chromium

- 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

- μOS++ Doxygen style guide link
- Teaching the art of great documentation, by Google