

# Modern C++ Programming

## 15. CODE CONVENTIONS

### PART II

---

*Federico Busato*

2025-01-22

## **1** auto

## **2** Templates and Type Deduction

## **3** Control Flow

- Redundant Control Flow
- if/else
- Comparison
- switch
- for/while

# Table of Contents

## 4 namespace

- using namespace Directive
- Anonymous/Unnamed Namespace
- Namespace and Class Design
- Style

## 5 Modern C++

- Keywords
- Features
- Class
- Library

## **6** Maintainability

- Code Comprehension
- Functions
- Template and Deduction
- Library

## **7** Portability

## 8 Naming

- Entities
- Variables
- Functions
- Style Conventions
- Enforcing Naming Styles

## **9** Readability and Formatting

- Horizontal Spacing
- Pointers/References
- Vertical Spacing
- Braces
- Type Decorators
- Reduce Code Verbosity
- Other Issues

## **10** Code Documentation and Comments

- Function Documentation
- Comment Syntax
- File Documentation

auto





- \* **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](#)

lambdas, iterators, template expressions

unreal (only)

- \* **Do not excessively use** `auto` **for variable types**. Use `auto` only when the left type is easy to deduce looking at the right expression

[GOOGLE](#)

- Don't use `auto` when the type would be deduced to be a pointer type

```
auto* v = new int;
```

[CHROMIUM](#)

- Use `auto` for *return type deduction* only with small/simple functions and lambda expressions

[GOOGLE](#) 7/78

# Templates and Type Deduction

---

## ※ Avoid complicated template programming

[GOOGLE](#)

\* Prefer automatic template deduction `f(0)` instead of `f<int>(0)`

- Use *class template argument deduction* (CTAD) only with templates that provide at least one explicit *deduction guide*

[GOOGLE](#)

- Use *trailing return types* only where using the ordinary syntax is impractical or much less readable

[GOOGLE](#), [WEBKIT](#)

`int foo(int x)` instead of `auto foo(int x) -> int`

# Templates and Type Deduction

- Declare *template specializations* in the same file as the primary template they specialize

[HiC](#)

```
template<typename T>
f(); // primary template

template<>
f<int>();
```

- Do not place spaces between the identifier template and its angle brackets

[WebKit](#)

```
template<typename U> struct Bar { };
```

# Control Flow

---

- ※ **Limit control flow complexity** (cyclomatic/cognitive complexity)

HIC,  $\mu$ OS, CLANG-TIDY

- \* **Avoid** `goto`

$\mu$ OS, CORECPP

\* **Avoid redundant control flow** (see next slides)

- Do not use `else` after a `return` / `break`
- Avoid comparing boolean condition to `true/false`
- Avoid `return true/return false` pattern
- Merge multiple conditional statements

[CLANG-TIDY](#), [CORECPP](#)  
[LLVM](#), [WEBKIT](#), [CLANG-TIDY](#)  
[MOZILLA](#)

```
if (condition) {      // BAD
  < body1 >
  return;             // <--
}
else                  // <-- redundant
  < body2 >
```

```
if (condition) {      // GOOD
  < body1 >
  return;
}
< body2 >
```

---

```
if (condition == true) // BAD
```

```
if (condition) // GOOD
```



```
if (condition)    // BAD
    return true;
else
    return false;
```

```
return condition; // GOOD
```

---

```
if (condition1) {
if (condition2) {
if (condition3) { // BAD
```

```
if (condition1 && condition2 && condition3) { // GOOD
```

```
bool condition4 = condition1 && condition2 && condition3;
if (condition4) { // GOOD
```

- \* The `if` and `else` keywords belong on separate lines

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

GOOGLE, WEBKIT

- Don't use the ternary operator ( `?:` ) as a sub-expression

```
(i != 0) ? ((j != 0) ? 1 : 0) : 0;
```

HIC

## Control Flow - Comparison

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

[HIC](#)

(opposite) [MOZILLA](#), [WEBKIT](#), [CORECPP](#)

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

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

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

[CHROMIUM](#)

- \* Prefer `switch` to multiple `if`-statement

[CORECPP](#)

- \* Don't use default labels in fully covered `switch` over enumerations

[LLVM](#), [CORECPP](#)

- \* In all other cases, `switch` statements should always have a `default` case

[GOOGLE](#), [UNREAL](#), [HIC](#), [CLANG-TIDY](#)

## Control Flow - switch - *Style*

- `case` blocks in `switch` statements are indented twice

GOOGLE

```
switch (var) {  
    case 0: {  
        Foo();  
        break;  
    }  
}
```

- A case label should line up with its `switch` statement. The case statement is indented

WEBKIT

```
switch (var) {  
case 0:  
    Foo();  
    break;  
}
```

- ※ Use *range-based for loops* whenever possible

[LLVM](#), [UNREAL](#), [CLANG-TIDY](#), [CORECPP1](#), [CORECPP2](#), [CORECPP3](#)

- \* Prefer a `for`-statement to a `while`-statement when there is an obvious loop variable

[CORECPP](#)

- \* Prefer a `while`-statement to a `for`-statement when there is no obvious loop variable

[CORECPP](#)

- Avoid `do-while` loop

[CORECPP](#)

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

[LLVM](#), [CORECPP](#)

```
for (<condition1>) {    // BAD
    if (<condition2>)
        ...
}
```

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

- \* Turn predicate loops into predicate functions

[LLVM](#), [CoreCPP](#)

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



# namespace

---

# Namespace

- ※ Always place code in a namespace to avoid *global namespace pollution*

[GOOGLE](#)

- ※ Do not use *namespace aliases* `namespace nsA = other_namespace` at namespace/global scope in header files except in explicitly marked internal-only namespaces

[GOOGLE](#), [MOZILLA](#)

- ※ Do not declare anything in the namespace `std`

[GOOGLE](#), [SEI CERT](#), [CLANG-TIDY](#), [CORECPP](#)

- ※ Do not use `using namespace` declarations of any kind to import names in the `std` namespace

[WEBKIT](#)

- \* Do not use `inline` namespaces

[GOOGLE](#)<sup>21/78</sup>

## using namespace Directive

- ※ **Avoid** using namespace -directives, especially at global scope

LLVM, GOOGLE, WEBKIT, UNREAL, HIC, μOS, CORECPP

```
#include <cmath> // if 'header.hpp' contains
#include "header.hpp" // 'using namespace std;'
auto f(float a) { return abs(a) * 2; } // f(3.5) returns 7 instead of 6
```

- \* **Limit** using namespace -directives at local scope and prefer explicit namespace entities declarations

GOOGLE, UNREAL, HIC, CLANG-TIDY

- using namespace is allowed in implementation files in nested namespaces

WEBKIT

# Anonymous/Unnamed Namespace

## ※ Avoid *anonymous* namespaces/ `static` in headers

[GOOGLE](#), [μOS](#), [SEI CERT](#), [CLANG-TIDY](#), [CORECPP](#)

### ■ `anonymous namespace` vs. `static`

- `anonymous namespaces` instead of `static` everywhere

[HIC](#), [CLANG-TIDY](#), [CORECPP](#)

- `anonymous namespaces` only for `struct` / `class` declaration, `static` otherwise (easy identification)

[LLVM](#), [MOZILLA](#), [μOS](#)

## \* **Anonymous namespaces and `static` in source files:**

Items local to a source file (e.g. `.cpp`) file should be wrapped in an anonymous namespace/marked `static`. Anonymous namespaces/ `static` restrict symbols visibility to the translation unit, improving function call cost and reduce the size of entry point tables

[GOOGLE](#), [CHROMIUM](#), [CORECPP](#), [HIC](#), [μOS](#)<sub>23/78</sub>

- ※ **All helper functions and operators of a class need to belong to the same namespace of the class**
- \* **Prefer free functions in namespaces instead of classes**, avoid global scope functions

GOOGLE

- \* The content of namespaces is not indented

LLVM, GOOGLE, WEBKIT

```
namespace ns {  
  
void f() {}  
  
}
```

- \* Close namespace declarations

LLVM, GOOGLE, WEBKIT, CLANG-TIDY

```
} // namespace <namespace_identifier>  
} // namespace (for anonymous namespaces)
```

- Namespaces should have unique names based on the project name

GOOGLE

- Prefer single-line nested namespace declarations `ns1::ns2` [C++17](#)  
[GOOGLE](#), [MOZILLA](#)
- Minimize use of nested namespaces [CHROMIUM](#)
- Namespaces can match hierarchy with file system hierarchy for consistency

```
include/  
├── my_project/  
│   ├── core.hpp  
│   └── detail/  
│       └── helper.hpp
```

```
namespace my_project::detail
```

# Modern C++

---



Use C++ over pure C and  
use *modern* C++ wherever possible

- ※ Use `constexpr` C++11 variables to define true constants (instead of *macro*)  
[GOOGLE](#), [WEBKIT](#), [CORECPP1](#), [CORECPP2](#)
- ※ Use `constexpr` C++20 function to ensure compile-time evaluation  
[GOOGLE](#)
- ※ Use `constexpr` C++20 to ensure constant initialization for non-constant variables  
[GOOGLE](#)
- ※ `static_assert` compile-time assertion  
[UNREAL](#), [HIC](#)

- ※ Prefer `enum class` C++11 instead of plain `enum` C++11

UNREAL, μOS, CORECPP

- \* Use `auto` C++11 to avoid type names that are noisy, obvious, or unimportant

```
auto array = new int[10];
```

```
auto var    = static_cast<int>(var);
```

LLVM, GOOGLE, HIC, CLANG-TIDY, CORECPP

(only for lambdas, iterators, template expressions)

UNREAL

- ※ `nullptr` C++11 instead of `0` or `NULL` for pointers

GOOGLE, UNREAL, WEBKIT, MOZILLA, HIC, μOS, CLANG-TIDY, CORECPP

- \* Use the `explicit` keyword for conversion operators C++11 and constructors. Do not define implicit conversions [GOOGLE](#), [MOZILLA](#), [μOS](#)
- \* Use `using` C++11 instead `typedef` [MOZILLA](#), [CLANG-TIDY](#), [CORECPP](#)
- \* Avoid `throw` function specifier. Use `noexcept` C++11 instead [MICROSOFT BLOG](#)

※ lambda expression C++11

UNREAL

※ move semantic C++11

UNREAL

※ Use *range-based for loops* whenever possible C++11

LLVM, UNREAL, CLANG-TIDY, CORECPP<sub>1</sub>, CORECPP<sub>2</sub>, CORECPP<sub>3</sub>

\* Prefer uniform (brace) initialization C++11 when it cannot be confused with  
`std::initializer_list`

CHROMIUM

- \* `static_cast`, `reinterpret_cast`, `const_cast`, `std::bit_cast` C++20,  
instead of *old style cast* `(type)` [LLVM](#), [GOOGLE](#), [μOS](#), [HIC](#), [CLANG-TIDY](#)
- \* Use `[[deprecated]]` C++14 / `[[noreturn]]` C++11 / `[[nodiscard]]` C++17 to indicate deprecated functions / that do not return / result should not be discarded [CLANG-TIDY](#)
- \* Use `= delete` C++11 to mark deleted functions
  - Replace SFINAE with concepts C++20 [CLANG-TIDY](#)
  - Use structure binding C++17

- \* Always use `override` C++11 and `final` function member keywords  
GOOGLE, WEBKIT, MOZILLA, UNREAL, HIC, CLANG-TIDY, CORECPP
- \* Use `= default` C++11 constructors

- \* Use braced *direct-list-initialization* or *copy-initialization* C++11 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  
};
```

- Replaces explicit calls to the constructor in a return with a braced initializer list

CLANG-TIDY

```
Foo bar() { return Foo(3); }  
Foo bar() { return {3};    }
```



# Modern C++ Library

- ✧ Avoid C-Style memory management `malloc()/free()` and use `new/delete`  
[CORECPP](#), [CLANG-TIDY](#)
- ✧ Except `int`, Use fixed-width integer type C++11 (e.g. `int64_t`, `int8_t`, etc.)  
[CHROMIUM](#), [UNREAL](#), [GOOGLE](#), [HIC](#), [μOS](#), [CLANG-TIDY](#)
- Use `std::print` C++23 [CLANG-TIDY](#)
- Uses modern type traits C++17 [CLANG-TIDY](#)

```
std::is_integral<T>::value;          // --> std::is_integral_v
std::make_signed<unsigned>::type;    // --> std::std::make_signed_t
```

# Maintainability

---

- ※ **Document code** (See code documentation section)

- ※ **Don't optimize without reason**

CORECPP

- \* **Address compiler warnings.** Compiler warning messages mean something is wrong

UNREAL

- \* **Compile-time and link-time errors should be preferred over run-time errors**

μOS, CORECPP

\* **Avoid *RTTI* (`dynamic_cast`) and *exceptions***

[LLVM](#), [GOOGLE<sub>1</sub>](#), [GOOGLE<sub>2</sub>](#), [MOZILLA<sub>1</sub>](#), [MOZILLA<sub>2</sub>](#), [HIC](#)

※ **Do not use reserved names**

[SEI CERT](#), [CLANG-TIDY](#)

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

- The `goto` statement shall not be used

[μOS](#), [CLANG-TIDY](#)

- Code that is not used (commented out) should be deleted

[μOS](#)

- Code should not include unnecessary constructs: variables, types, unreachable code

[μOS](#)<sup>37/78</sup>

## ※ Do not depend on the order of evaluation for side effects

[SEI CERT](#)

```
f(i++, i++);  
a[i++] = i;
```

- Do not perform assignments in conditional statements [SEI CERT](#), [CLANG-TIDY](#)

```
if (a = b)
```

\* Prefer `sizeof(variable/value)` instead of `sizeof(type)`[GOOGLE](#)\* Avoid octal numbers, e.g. `int v = 0010; //8`[HIC](#), [μOS](#)

# Maintainability - Code Comprehension

## ※ Write self-documenting code

e.g. `(x + y - 1) / y` → `ceil_div(x, y)`

UNREAL

## ※ Use symbolic names instead of literal values in code (don't use magic numbers)

HIC, CLANG-TIDY, CORECPP

```
double    area1 = 3.14 * radius * radius; // BAD
constexpr auto Pi    = 3.14;                // correct
double    area2 = Pi * radius * radius;
```

- Use parentheses in expressions to specify the intent of the expression, especially with mixed operators

HIC, μOS, CLANG-TIDY, CORECPP

```
int r = i + j * k - 4 / 5;                // BAD
if ((i != 0) && (j != 0) || (k != 0)) // correct
```

✧ Enforce `const`-correctness

- Pass function arguments by `const` pointer or reference
- Function members
- Use `const` iteration over containers if the loop isn't intended to modify the container

UNREAL  
CORECPP  
CORECPP

- Declare an object `const` or `constexpr` unless you want to modify its value later on

CORECPP<sub>1</sub>, CORECPP<sub>2</sub>, UNREAL

- but **don't** `const` all the things

<sup>1</sup>, CORECPP

- Pass by-`const` value: almost useless (copy), ABI break
- `const` return: useless (copy)
- `const` data member: disable assignment and copy constructor
- `const` local variables: verbose, rarely effective

CLANG-TIDY, UNREAL

---

<sup>1</sup> Don't const all the things

# Maintainability - Functions

- ※ Use `assert` to document preconditions and assumptions

[LLVM](#), [CORECPP](#)

- Ensure that all statements are reachable for at least one combination of function inputs

[HIC](#)

- Prevent using functions that don't accept `nullptr`

[CORECPP](#)

```
void f(void*);  
void f(nullptr) = delete;  
// f(nullptr) // compile error
```



# Maintainability - Object Semantic

- \* Prefer RAII instead of manual resource management

[CORECPP1](#), [CORECPP2](#)

```
void f(char* name) {  
    FILE* input = fopen(name, "r"); // use "ifstream input {name};" instead  
    if (something) return;          // BAD: if something == true,  
    // ...                          // a file handle is leaked  
    fclose(input);  
}
```

- ✱ Never transfer ownership by a raw pointer (T\*) or reference (T&). Use object semantics, `unique_ptr`, etc. [CORECPP](#)

- \* **Avoid singletons.** Use a `static` member function named `singleton()` to access the instance of the singleton instead of a free function [WEBKIT](#), [CORECPP42/78](#)

## ※ Avoid complicated template programming

### \* Be aware of bug-prone deductions

```
template<typename T, int N>
void f(const T&);

template<typename T>
void f(T); // same of f(T*)

int array[3];
f(array); // call the second funtion, not f(T&)
```

- \* **Do not pass an array as a single pointer.** Prefer `std::span`, `std::mdspan`  
[CORECPP](#)
- \* **Prefer core-language features** over library facilities, e.g. `uint8_t` vs. `std::byte`
- Prefer `std::array` over plain array. It can be also used to return multiple values of the same type from a function  
[CORECPP<sub>1</sub>](#), [CORECPP<sub>2</sub>](#)
- Use `std::string_view` to refer to character sequences  
[CORECPP](#)

# Portability

---

- ※ **Ensure ISO C++ compliant code. Do not use non-standard extensions**  
see `-Wpedantic` [HIC](#), [GOOGLE1](#), [GOOGLE2](#), [μOS](#), [CORECPP](#)
- ※ Do not use deprecated C++ features, or asm declarations, e.g. `register`, `__attribute__`, `throw` (function qualifier) [HIC](#)
- ※ **Do not use `reinterpret_cast` or `union` for type punning**  
Prefer `std::bit_cast` or `std::memcpy` [CORECPP1](#), [CORECPP2](#), [HIC](#)
- ※ Except `int`, use fixed-width integer type (e.g. `int64_t`, `int8_t`, etc.)  
[CHROMIUM](#), [UNREAL](#), [GOOGLE](#), [HIC](#), [μOS](#), [CLANG-TIDY](#)

✖ Don't use `long double`

\* **Do not use UTF characters\*** for portability, prefer ASCII [GOOGLE](#), [μOS](#)

\* If UTF is needed, **prefer UTF-8 encoding for portability** [GOOGLE](#), [CHROMIUM](#)

\* **Use the same line ending** (e.g. `'\n'`) for all files [MOZILLA](#), [CHROMIUM](#)

---

\* Trojan Source attack for introducing invisible vulnerabilities

# Naming

---

*“Beyond basic mathematical aptitude, the difference between good programmers and great programmers is verbal ability”*

***Marissa Mayer***



- \* **Naming is hard.** *Most of the time, code is shared with other developers. It is worth spending a few seconds to find the right name*
- \* **Think about the purpose to choose names**
- \* **Adopt names commonly used in real contexts** (outside the code)
- \* **Don't use the same name for different things.** Use a specific name everywhere
  - Prefer single **English** word to implementation-focused, e.g.  
`UpdateConfigFile()` → `save()`
  - Use natural word pair, e.g. `create()/destroy()`, `open()/close()`,  
`begin()/end()`, `source()/destination()`

- Don't overdecorate, e.g. `Base/Impl` , `Factory/Singleton`
- Don't list the content, e.g. `NameAndAddress` → `ContactInfo`
- Don't repeat class/enum names, e.g. `Employee::EmployeeName`
- Avoid temporal attributes, e.g. `PreLoad()` , `PostLoad()`
- Use adjectives to enrich a name, e.g. `Name` → `FullName` , `Salary` → `AnnualSalary`

- \* **Abbreviations are generally bad**, longer names are better in most cases (don't be lazy)  $\mu$ OS
- \* **Use whole words**, except in the rare case where an abbreviation would be more canonical and easier to understand, e.g. `tmp` WEBKIT
- \* **Avoid short and very long names**. Remember that the average word length in English is 4.8 CLANG-TIDY

- Avoid names that are easily misread: similar or hard to pronounce [CORECPP](#)

✘ Avoid ambiguous characters, o/0/0, I/1/1, s/S/5, Z/2, N/n/h, B/8  
e.g. hel1o [HIC](#), [μOS](#), [CORECPP](#)

- Do not abbreviate by deleting letters within a word [GOOGLE](#)

- If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme [GOOGLE](#)

- \* The length of a variable should be **proportional to the size of the scope** that contains it. For example, `i` is fine within a loop

`GOOGLE, CORECPP1, CORECPP2`

- Names can be made singular or plural depending on whether they hold a single value or multiple values, thus arrays and collections should be plural

`μOS`

```
int value;  
int values[N];
```

- Use common loop variable names

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

- Make literals readable

CORECPP

```
auto c          = 299'792'458; // digit separation  
auto interval = 100ms;         // using <chrono>
```

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

WEBKIT

- \* **Should describe their action or effect instead of how they are implemented**, e.g. `partial_sort()` → `top_n()`

- \* **Functions that return boolean values should start with boolean verbs**, like

`is`, `has`, `should`, ~~`does`~~

`empty()` → `is_empty()`

μOS

# Naming Style Conventions

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

```
CapitalStyle
```

**Camel-Back** Uppercase first word letter except the first one (less readable, shorter names)

```
camelBack
```

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

```
snake_style
```

**Macro** Upper case words separated by single underscore (sometimes called *All Capitalized* or *Screaming style*) (best readability, longer names)

```
MACRO_STYLE
```



# Naming Style Conventions - Variables/Constant

## Variable Variable names should be nouns

- Capital style e.g. `MyVar`
- Snake style e.g. `my_var`
- Global variable with `g` prefix, e.g. `gVar`
- Arguments with `a` prefix, e.g. `aVar`

LLVM, UNREAL

GOOGLE, WEBKIT, STD, μOS

MOZILLA

MOZILLA

## Constant

- Capital style + `k` prefix, e.g. `kConstantVar`
- Snake style e.g. `my_var`
- Macro style e.g. `CONSTANT_VAR`

GOOGLE, MOZILLA

μOS

OPENSTACK

# Naming Style Conventions - Function

- Camel-back style, e.g. `myFunc()` LLVM
- Capital style, e.g. `MyFunc()` GOOGLE, CHROMIUM, MOZILLA, UNREAL
- Snake style, e.g. `my_func()` WEBKIT, STD,  $\mu$ OS
- Snake style for accessor and mutator methods GOOGLE, CHROMIUM

# Naming Style Conventions - Enum/Namespace

## Enum

- Capital style + `k`

GOOGLE

e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }`

- `e` prefix

MOZILLA

e.g. `enum MyEnum { eVar1, eVar2 }`

- Capital style

LLVM, WEBKIT, UNREAL

e.g. `enum MyEnum { EnumVar1, EnumVar2 }`

- Snake style

μOS

e.g. `enum MyEnum { enum_var1, enum_var2 }`

## Type Should be nouns

- Capital style (including classes, structs, enums, typedefs, template, etc.)

e.g. `HelloWorldClass`

LLVM, GOOGLE, WEBKIT, UNREAL

- Snake style

μOS (class), STD<sub>58/78</sub>

# Naming Style Conventions - Type/Macro/File

## Namespace

- Snake style, e.g. `my_namespace`
- Capital style, e.g. `MyNamespace`

GOOGLE, LLVM, STD

WEBKIT, UNREAL

**Macro** Macro style, e.g. `MY_MACRO`

GOOGLE, STD, UNREAL, WEBKIT, MOZILLA, CORECPP

Macro style should be used only for macros

CORECPP<sub>1</sub>, CORECPP<sub>2</sub>, CORECPP<sub>3</sub>, CORECPP<sub>4</sub>

## File

- Snake style ( `my_file` )
- Capital style ( `MyFile` ), could lead Windows/Linux conflicts

GOOGLE

LLVM

PERSONAL COMMENT: **Macro style** needs to be used only for macros to avoid subtle bugs. I prefer **snake style** for almost everything because it has the best readability. On the other hand, I don't want to confuse `typename`s and variables, so I use **camel style** for the former ones. Finally, I also use **camel style** for compile-time constants because they are very relevant in my work and I need to quickly identify them

# Enforcing Naming Styles

Naming style conventions can be also enforced by using tools like

`clang-tidy: readability-identifier-naming` [↗](#)

.clang-tidy configuration file

```
Checks: 'readability-identifier-naming'
HeaderFileExtensions: [' ', 'h', 'hh', 'hpp', 'hxx']
ImplementationFileExtensions: ['c', 'cc', 'cpp', 'cxx']
CheckOptions:
  readability-identifier-naming.ClassCase: 'lower_case'
  readability-identifier-naming.MacroDefinitionCase: 'UPPER_CASE'
```

```
class MyClass {}; // before
#define my_macro
class my_class {}; // after
#define MY_MACRO
```

# Readability and Formatting

---

- ※ **Limit line length (width)** to be at most **80 characters** long (or 100, or 120) → help code view on a terminal `LLVM` (80), `GOOGLE` (80), `µOS`(120)

PERSONAL COMMENT: I was tempted several times to use a line length  $> 80$  to reduce the number of lines, and therefore improve the readability. Many of my colleagues use split-screens or even the notebook during travels. A line length of **80 columns** is a good compromise for everyone

- 
- Is the 80 character limit still relevant in times of widescreen monitors?
  - Linus Torvalds on 80 column limit



## ※ Use always the same indentation style

- tab → 2 spaces
- tab → 4 spaces
- (actual) tab = 4 spaces

GOOGLE,  $\mu$ OS  
LLVM, WEBKIT, HIC, PYTHON  
UNREAL

PERSONAL COMMENT: I worked on projects with both two and four-space tabs. I observed less bugs due to indentation and better readability with **four-space tabs**. 'Actual tabs' breaks the line length convention and can introduce tabs in the middle of the code, producing a very different formatting from the original one

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

LLVM, GOOGLE<sub>1</sub>, GOOGLE<sub>2</sub>, WEBKIT, CORECPP

```
if(a*b<10&&c)           // BAD
if (a * c < 10 && c)     // good
```

- \* Prefer consecutive alignment

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

- Do not place spaces around unary operators `i ++`

WEBKIT

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

GOOGLE  
64/78

# Pointers/References

- 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 symbols in the same way

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

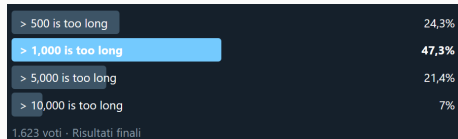
[GOOGLE](#)  
[WEBKIT](#), [CHROMIUM](#), [UNREAL](#), [CORECPP](#)

- Pointer and reference types and variables have no space after the `*` or `&`

[GOOGLE](#)

```
char * v;      // BAD
auto & v = w;   // BAD
* p = 3;       // BAD
v. x + 2;      // BAD
x = r-> y;     // BAD
```

- \* Do not write excessive long file



- \* Each statement should get its own line

WEBKIT, μOS, CORECPP<sub>1</sub>, CORECPP<sub>2</sub>, HIC, GOOGLE

```
x++;  
y++;  
if (condition)  
    doIt();
```

- \* Minimize the number of empty rows. **The more code that fits on one screen, the easier it is to follow and understand the control flow of the program**

GOOGLE

- Close files with a blank line

UNREAL

- \* Multi-lines statements and complex conditions require curly braces. Use an additional boolean variable if possible

[GOOGLE](#)<sub>1</sub>, [GOOGLE](#)<sub>2</sub>, [WEBKIT](#)

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

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

[LLVM](#), [GOOGLE](#), [WEBKIT](#)

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

- Always use brace for all control statements

[MOZILLA](#), [CHROMIUM](#), [μOS](#)

### \* Use always the same style for braces

- Same line, aka Kernigham & Ritchie

`GOOGLE1`, `GOOGLE2`  
`WEBCIT` (function only), `CORECPP` (expect for function)

- Its own line, aka Allman

`UNREAL`, `WEBCIT` (class, namespace, control flow)

```
//Kernigham & Ritchie  
int main() {  
    code  
}
```

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

PERSONAL COMMENT: C++ is a very verbose language. **Same line** convention helps to keep the code more compact, improving the readability

# Type Decorators

- The same concept applies to `const`
  - `const int*` *West notation*
  - `int const*` *East notation*

GOOGLE, CORECPP  
AUTOSAR (RULE A7-1-3)

PERSONAL COMMENT: I prefer **West notation** to prevent unintentional cv-qualify (const/volatile) of a reference or pointer types `char &const p`, see DCL52-CPP. Never qualify a reference type with `const` or `volatile`

- Prefer the common order of declaration `static constexpr int var`

μOS



# Reduce Code Verbosity

- Use the **short name version** of built-in types, e.g.

`unsigned` instead of `unsigned int`

`long long` instead of `long long int`

WEBCIT

- **Don't** `const` **all the things**. Avoid Pass by-`const` , `const` return, `const` data member, `const` local variables

## Other Issues

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

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

[WEBKIT, CLANG-TIDY](#)

- Boolean expressions at the same nesting level that span multiple lines should have their operators on the left side of the line instead of the right side [WEBKIT](#)

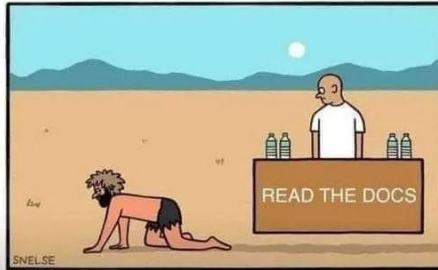
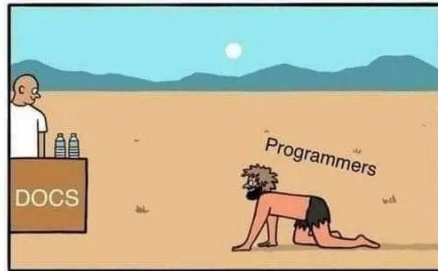
```
return attribute.name() == srcAttr  
    || attribute.name() == lowsrcAttr;
```

**Final note**: Most of the formatting guidelines can be forced by using `clang-tidy` [↗](#)  
and `clang-format` [↗](#)

# **Code Documentation and Comments**

---

# Programmers vs. Documentation



## ※ Comment *what* the code does and *why*

[LLVM](#), [CORECPP](#)

- Avoid *how* it is implemented at low level
- All files should report a brief description of their purpose
- Describe classes and methods

## \* Don't say in comments what can be clearly stated in code

[CORECPP](#)

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

# Function Documentation

- \* **The first sentence** (beginning with `@brief`) is used as an abstract
- \* **Document the inputs:** `@param[in]`, `@param[in,out]`, , and template parameters `@tparam`
- \* **Document outputs:** return value `@return` and output parameters `@param[out]` [GOOGLE](#), [UNREAL](#)
- \* **Document preconditions:** input ranges, impossible values (e.g. `nullptr`), status/return values meaning [UNREAL](#)
- \* **Document program state changes** (e.g. `static`), **arguments with lifetime** beyond the duration of the method call (e.g. constructors), **performance implications** [GOOGLE](#), [UNREAL](#)

# Comment Syntax

- \* Prefer `//` comment instead of `/* */` → prevent bugs and allow string-search tools like `grep` to identify valid code lines [HIC](#), [μOS](#)
- Use the same style of comment `//`, `///`, `/**`, `/*!`, etc.
- Multiple lines and single line comments can have different styles

```
/**  
 * comment1  
 * comment2  
 */  
/// single line
```

- 
- [μOS++ Doxygen style guide link](#)
  - [Teaching the art of great documentation, by Google](#)

## Other Comment Issues

- Use anchors for indicating special issues: `TODO` , `FIXME` , `BUG` , etc.  
[WEBKIT](#), [CHROMIUM](#)
- Only one space between statement and comment  
[WEBKIT](#)



- \* **Any file start with a license** (even scripts)

[GOOGLE](#), [LLVM](#)

- **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