Modern C++ Programming

16. Code Conventions

Part II

Federico Busato

2026-01-06

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

auto

`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

Templates and Type Deduction

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

Control Flow

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

  Hic, μOS, Clang-Tidy

- ∗ **Avoid** `goto`

  μOS, CoreCpp

Redundant Control Flow

* **Avoid redundant control flow** (see next slides) Clang-Tidy, CoreCpp
  - Do not use `else` after a `return` / `break` LLVM, Webkit, Clang-Tidy
  - Avoid comparing boolean condition to `true/false` Mozilla
  - Avoid `return true/return false` pattern
  - Merge multiple conditional statements

Redundant Control Flow

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

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

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

Redundant Control Flow

```
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
```

Control Flow - `if/else`

* **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

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

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

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

Control Flow - `switch`

* 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;
}
```

Control Flow - `for/while`

※ **Use *range-based for loops* whenever possible**
LLVM, Unreal, Clang-Tidy, CoreCpp[1], CoreCpp[2], CoreCpp[3]

∗ 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

Control Flow - `for/while`

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

LLVM, CoreCpp

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

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

Control Flow - `for/while` 3/3

* 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

`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

Namespace and Class Design

※ **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

Style 1/2

* 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

Style

- 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++

Modern C++

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

Modern C++ Keywords

- ※ **Use `constexpr` C++11 variables to define true constants** (instead of *macro*)
  GOOGLE, WEBKIT, CORECPP1, CORECPP2

- ※ **Use `consteval` C++20 function to ensure compile-time evaluation**
  GOOGLE

- ※ **Use `constinit` C++20 to ensure constant initialization for non-constant variables**
  GOOGLE

- ※ `static_assert` **compile-time assertion**
  UNREAL, HIC

Modern C++ Keywords

※ **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

Modern C++ Keywords

* **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

Modern C++ Features

※ **lambda expression** C++11 UNREAL

※ **move semantic** C++11 UNREAL

※ **Use *range-based for loops* whenever possible** C++11
LLVM, UNREAL, CLANG-TIDY, CORECPP1, CORECPP2, CORECPP3

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

Modern C++ Features

* `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

Modern C++ Features - Class

* **Always use `override` C++11 and `final` function member keywords**
  Google, Webkit, Mozilla, Unreal, Hic, Clang-Tidy, CoreCpp

* Use `= default` C++11 constructors

Modern C++ Features - Class

* **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

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

Maintainability

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

  LLVM, Google1, Google2, Mozilla1, Mozilla2, 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

Maintainability 3/3

- ※ **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
```

Maintainability - Constants

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

- Declare an object `const` or `constexpr` unless you want to modify its value later on COREcPP1, COREcPP2, UNREAL

- but **don't `const` all the things** [1], COREcPP
  - Pass by-`const` value: almost useless (copy), ABI break
  - `const` return: useless (copy) CLANG-TIDY, UNREAL
  - `const` data member: disable assignment and copy constructor
  - `const` local variables: verbose, rarely effective

[1] `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

```cpp
#include <cstddef> // std::nullptr_
void f(void*);
void f(std::nullptr_t) = 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, CoreCpp

Maintainability - Template and Deduction

※ **Avoid complicated template programming** GOOGLE

∗ 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&)
```

Maintainability - Library

- ∗ **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 CoreCpp1, CoreCpp2

- Use `std::string_view` to refer to character sequences CoreCpp

`Prefer core-language features over library facilities`

Portability

Portability 1/2

※ **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

Portability 2/2

- ※ 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

Naming

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

Marissa Mayer

General Notes on Naming

* **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()` $\rightarrow$ `save()`

- Use natural word pair, e.g. `create()/destroy()`, `open()/close()`, `begin()/end()`, `source()/destination()`

General Notes on Naming

- Don't overdecorate, e.g. `Base/Impl`, `Factory/Singleton`
- Don't list the content, e.g. `NameAndAddress` $\rightarrow$ `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` $\rightarrow$ `FullName`, `Salary` $\rightarrow$ `AnnualSalary`

Naming is Hard: Let's Do Better, CppCon 2019, Kate Gregory

Entities Naming 1/2

- **Abbreviations are generally bad**, longer names are better in most cases (don't be lazy) μOS
- **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

Entities Naming

- Avoid names that are easily misread: similar or hard to pronounce 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

Literals

- ※ Avoid ambiguous characters, `o/O/0`, `I/l/1`, `s/S/5`, `Z/2`, `N/n/h`, `B/8`
  e.g. `hel1o` Hic, μOS, CoreCpp

- Use uppercase for post-fix literals, `1234L`, `1234ULL` SEI Cert

- Hexadecimal constants should be uppercase, `0x1BA7` Autosar, A2-13-5
  (personal) exception: do not mix with uppercase for post-fix literals, `0x1BAcULL`

- Make literals readable CoreCpp

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

Variables Naming

* 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

Functions Naming

* **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()` $\rightarrow$ `top_n()`

* **Functions that return boolean values should start with boolean verbs**, like `is`, `has`, ~~`should`, `does`~~ μOS
  `empty()` $\rightarrow$ `is_empty()`

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` — LLVM, Unreal
- Snake style e.g. `my_var` — Google, Webkit, Std, μOS
- Global variable with `g` prefix, e.g. `gVar` — Mozilla
- Arguments with `a` prefix, e.g. `aVar` — Mozilla

Constant

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

Naming Style Conventions - Type/Macro/File

Namespace

- Snake style, e.g. `my_namespace` — GOOGLE, LLVM, STD
- Capital style, e.g. `MyNamespace` — WEBKIT, UNREAL

Macro Macro style, e.g. `MY_MACRO`
GOOGLE, STD, UNREAL, WEBKIT, MOZILLA, CORECPP

Macro style should be used only for macros
CORECPP[1], CORECPP[2], CORECPP[3], CORECPP[4]

File

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

Personal Comment

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 `typenames` 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

Horizontal Spacing

※ **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`

Horizontal Spacing 2/3

※ **Use always the same indentation style**

- tab → 2 spaces — GOOGLE, μOS
- tab → 4 spaces — LLVM, WEBKIT, HIC, PYTHON
- (actual) tab = 4 spaces — 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

Horizontal Spacing 3/3

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

LLVM, GOOGLE$_1$, GOOGLE$_2$, 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

Pointers/References

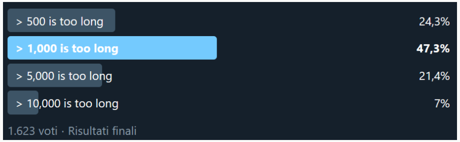
- Declaration of pointer/reference variables or arguments may be placed with the asterisk/ampersand *adjacent* to either the *type* or to the *variable name* for all symbols in the same way GOOGLE
  - `char* c;` WEBKIT, CHROMIUM, UNREAL, CORECPP
  - `char *c;`
  - `char * c;`

- 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
```

Vertical Spacing

* 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

* Each statement should get its own line

Webkit, μOS, CoreCpp$_1$, CoreCpp$_2$, Hic, Google

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

What is your threshold for a long source file?

Vertical Spacing 2/2

* 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 (C98 compatibility) Unreal

Braces

* Multi-lines statements and complex conditions require curly braces. Use an additional boolean variable if possible GOOGLE$_1$, GOOGLE$_2$, 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

Braces

* **Use always the same style for braces**
  - Same line, aka `Kernigham & Ritchie`

    Google1, Google2

    Webkit (function only), CoreCpp (expect for function)

  - Its own line, aka `Allman`

    Unreal, Webkit (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* GOOGLE, CORECPP
  - `int const*` *East notation* 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. WEBKIT
  `unsigned` instead of `unsigned int`
  `long long` instead of `long long int`

- **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

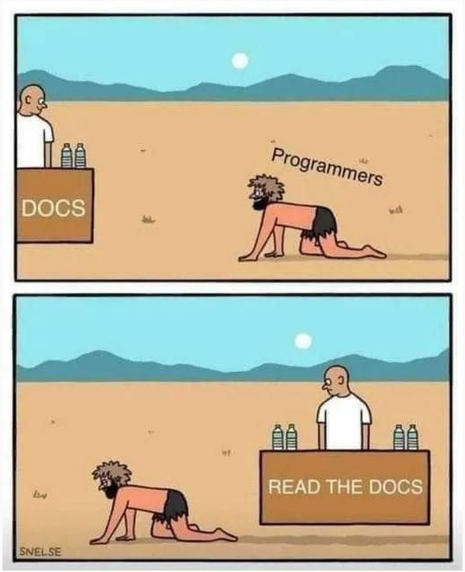
- Boolean expressions at the same nesting level that span multiple lines should have their operators on the left side of the line instead of the right side WEBKIT

```
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



Code 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

File Documentation

∗ **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