Modern C++ Programming

14. Code Conventions

Part I

Federico Busato

2025-01-22

Table of Contents

Table of Contents

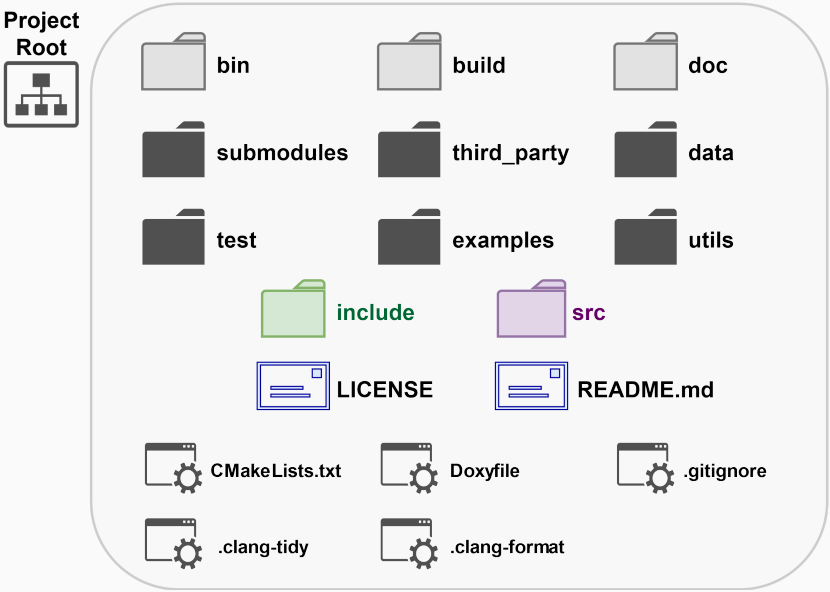
Table of Contents

Table of Contents

9 Structs and Classes

C++ Project Organization

“Common” Project Organization



Fundamental directories

`include` Project *public* header files

`src` Project source/implementation 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.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

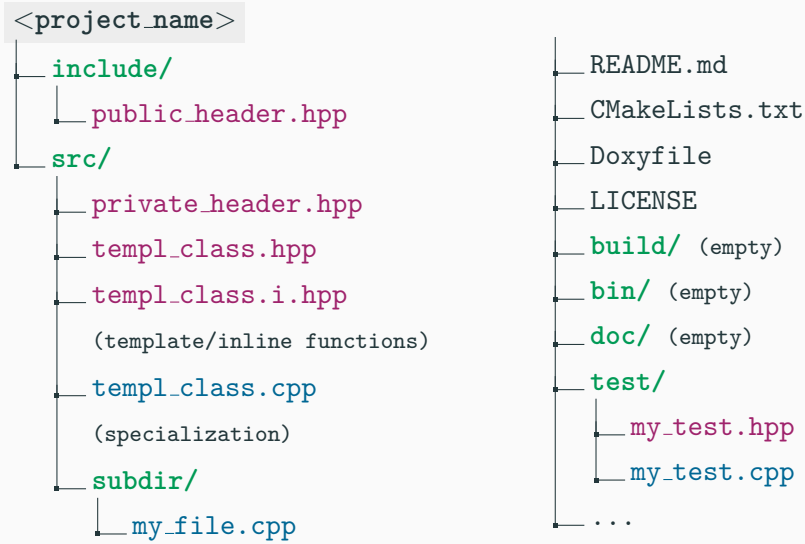
Common C++ file extensions:

- **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” Project Organization Notes

- Public **header(s)** in `include/`
- **source files**, private **headers**, **header implementations** in `src/` directory
- The **main** file (if present) should be placed in `src/` and called `main.cpp`
- **Code tests**, *unit* and *functional* tests can be placed in `test/`. Alternatively, **unit tests** can appear in the same directory of the component under test with the same filename and include `.test` suffix, e.g. `my_file.test.cpp`

"Common" Project Organization Example



"Common" Project Organization - Improvements

The "common" project organization can be improved by adding the *name of the project* as subdirectory of `include/`
Some projects often entirely avoid the `include/` directory

This is particularly useful when the project is used as *submodule* (part of a larger project) or imported as an *external library*

The includes now look like:

```
#include <my_project/public_header.hpp>
```

```
<project_name>
├── include/
│   └── <project_name>/
│       └── public_header.hpp
└── src/
    └── private_file.cpp
```

- *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 `test/` subdirectory

<project_name> (v1)

- <project_name>/
  - public_header.hpp
  - private_header.hpp
  - my_file.cpp
  - my_file.mpp
  - my_file.test.cpp
- test/
  - my_functional_test.cpp
- build/
- doc/
- ...

<project_name> (v2)

- <project_name>/
  - public_header.hpp
  - private/
    - private_header.hpp
    - my_internal_file.cpp
    - my_internal_file.test.cpp
- test/
  - my_functional_test.cpp
- build/
- doc/
- ...

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

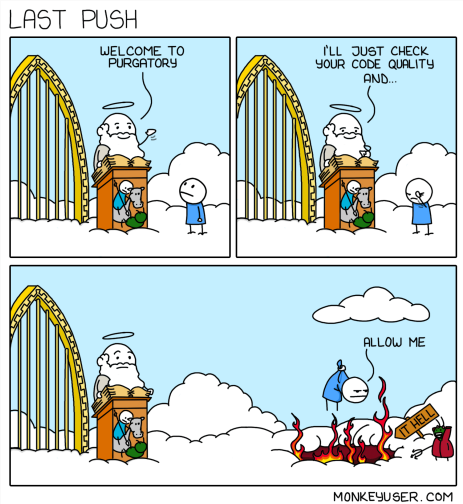
"80% of the lifetime cost of a piece of software goes to maintenance"

Unreal Engine

Code Quality

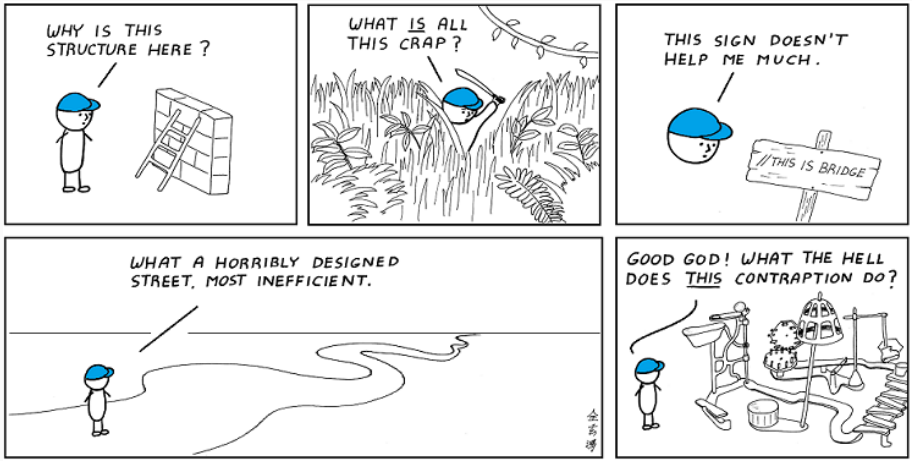
“The worst thing that can happen to a code base is size”

— Steve Yegge



Bad Code

How *my* code looks like for other people?



abstrusegoose.com/432

Coding Styles Overview

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

A **consistent code** base helps developers better understand code organization, focus on program logic, and reduce the time spent interpreting other engineers' intentions

Personal Comment: Don't start a project that involves multiple engineers without establishing clear guidelines that all engineers agree to. This is essential to avoid costly refactoring, personal style discussions, and conflicts later on

This section, including the review of all coding styles, has been updated on October 2024

- ***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`
  The Firefox code base adopts parts of the Google Coding style for C++ code (C++17, 2020), but not all of its rules
- ***Chromium Coding Style***. `chromium.googlesource.com`
  Chromium follows the Google C++ Style Guide with some exceptions

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

- ***μOS++*** (derived from `MISRA 2018` and `JSV`)
  `micro-os-plus.github.io/develop/coding-style`
  `micro-os-plus.github.io/develop/naming-conventions`

More educational-oriented guidelines

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

Secure Coding

- ***High Integrity C++ Coding Standard***. `www.perforce.com/resources`
- ***CERT C++ Secure Coding***. `wiki.sei.cmu.edu`

Critical system coding standards

- ***MISRA C++17, 2023***. `www.misra.org.uk`
- ***Autosar C++14, 2019*** (based on MISRA:2008). `www.autosar.org`
- ***Joint Strike Fighter Air Vehicle (JSV) C++, 2005***. `JSF-AV-rule`

Static Analysis Tools

- ***clang-tidy***
  `clang.llvm.org/extra/clang-tidy/checks/list.html`
- ***PVS-Studio***
  `pvs-studio.com/en/docs/warnings`
- ***SonarSource***
  `rules.sonarsource.com/cpp/`
- ***cpp-checks***
  `sourceforge.net/p/cppcheck/wiki/ListOfChecks/`

Note: each tool also provides the list of checks that are evaluated

Legend

※ → **Important!**
Highlight potential code issues such as bugs, inefficiency, or important readability problems. 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, not very common issue, or hard to enforce

Header Files and `#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, CoreCpp

∗ **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, Mozilla, Clang-Tidy, CoreCpp

∗ **Every source file should have an associated header file** Google, CoreCpp

* `#include` preprocessor should be placed immediately after the header comment and include guard LLVM, μOS, CoreCpp

* **Use C++ headers instead of C headers.** C++ headers define additional functions and their symbols are in the `std` namespace Hic
  - `<cassert>` instead of `<assert.h>`
  - `<cmath>` instead of `<math.h>`, etc.

#include Guard

※ **Always use an include guard**

LLVM, Google, Chromium, Unreal, CoreCpp

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

    LLVM, Google, Chromium, CoreCpp, Mozilla, Hic
  - `#pragma once` otherwise

    Webkit, Unreal

※ **Ensure a unique name for the include guard**, e.g. project_name + path

Googleq

#include Syntax

`""` syntax

* Should be absolute paths from the project include root GOOGLE, MOZILLA, HIC
  e.g. `#include "directory1/header.hpp"`

`<>` syntax

- Any external code WEBKIT
- Only where strictly required GOOGLE, HIC, MOZILLA, CORECPP
  C/C++ standard library headers `#include <iostream>`
  POSIX/Linux/Windows system headers (e.g. `<unistd.h>` and `<windows.h>`

Order of #include

LLVM, Webkit, Mozilla, CoreCpp

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

- space

(2) Current project includes

- space

(3) Third party includes

- space

(4) System includes

Motivation: System/third party includes are self-contained, local includes might not

Google: **(4)** → **(3)** → **(2)**

Note: headers within each section are lexicographic ordered

Google, Webkit

#include - Other Issues

- **Report at least one function used for each include**. It helps to identify unused headers

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

- **Forward declarations vs. #includes**
  - *Prefer forward declaration*: reduce compile time, less dependency CHROMIUM
  - *Prefer* `#include` : *safer* GOOGLE

Common Header/Source Filename Conventions

- `.h` `.c` `.cc` — GOOGLE, μOS(`.h`)
- `.hh` `.cc` — (rare)
- `.hpp` `.cpp` — μOS(`.cpp`)
- `.hxx` `.cxx` — (rare)

Example

```
// [ LICENSE ]
#ifndef PROJECT_A_MY_HEADER
#define PROJECT_A_MY_HEADER

#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

// ..

#endif // PROJECT_A_MY_HEADER
```

Preprocessing

- **Avoid defining macros**, especially in headers — Google
  - Do not use macro for enumerators, constants, and functions — μOS, CoreCpp$_1$, CoreCpp$_2$
- **Always put macros after** `#include` statements — μOS
- **Macros should be unique names**, e.g. use a prefix for all macros related to a project `MYPROJECT_MACRO` — Google, Unreal, CoreCpp
- `#undef` **macros wherever possible** — Google
  - Even in the source files if *unity build* is used (merging multiple source files to improve compile time)

※ **Always use curly brackets for multi-line macro** CLANG-TIDY

```
#define INCREMENT_TWO(x, y) (x)++; (y)++
if (do_increment)
  INCREMENT_TWO(a, b);  // (b)++ will be executed unconditionally
//-----------------------------------------------------------------
#define INCREMENT_TWOO(x, y) \
{                            \
    (x)++;                   \
    (y)++;                   \
}
```

※ Macro shall not have side effect CLANG-TIDY

```
#define MIN(X, Y) (X < Y ? X : Y)  // MIN(i++) -> increased twice
```

※ In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses to prevent unexpected expressions μOS, CLANG-TIDY

```
#define ADD(x, y) ((x) + (y))
```

∗ **Prefer checking macro values**. It prevents mistakes deriving from missing headers

```
#define MACRO 1 // defined in another header
//---------------------------------------
#if MACRO       // instead of #if defined(MACRO)
```

▪ Put macros outside namespaces as they don't have a scope

* Close `#endif` with a comment 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
```

* Avoid conditional `#include` when possible — Mozilla, Chromium

- Prefer `#if defined(MACRO)` instead of `#ifdef MACRO`

  Improve readability, help grep-like utils, and it is uniform with multiple conditions

```
#if defined(MACRO1) && defined(MACRO2)
```

- Place the `\` rightmost for multi-line preprocessing statements

```
#define MACRO2                  \
    macro_def...
```

Variables

※ ***Always initialize*** **variables in the declaration**

Google, CoreCpp, Hic, μOS, SEI Cert, Clang-Tidy

※ **Place variables in the *narrowest scope* possible**. Declare variables close to the first use Google, CoreCpp$_1$, CoreCpp$_2$, CoreCpp$_3$

- It is allowed to declare multiple variables in the same line for improving the readability, except for pointer or reference Google

  (only one declaration per line) CoreCpp

- Use assignment syntax `=` when performing "simple" initialization, `{}` otherwise
  Chromium, CoreCpp
- Initialize variables with `=`, constructors with `{}`
  Mozilla
- Variables with narrow scope need by `if`, `while`, `for` statements should normally be declared within those statements `if (int* ptr = f())`
  Google

* Precede boolean values with words like `is` and `did`
  Webkit, Chromium

- Use `\0` to indicate the null character
  Google

```
char n = '\0';
```

`static` Global Variables

* **Avoid `static` global variables unless they are trivially destructible** Google
  e.g. `std::string str =` is not trivially destructible
  - `static` local variables with dynamic initialization are allowed

* **Avoid `static` global variables unless they are trivially constructible and destructible** LLVM

* **Avoid non-`const` `static` global variables** Hic, Mozilla, CoreCpp

- *Constant initialization* of `static` global variables should be marked with `constexpr` or `constinit` Google, Clang-Tidy

- `static` global variables should only be initialized by constant expressions (e.g. `constexpr` functions/lambdas) Google, Clang-Tidy

Conversions

* Use `static_cast` instead of old-style cast — Google

* Use `const_cast` to remove the `const` qualifier only for pointers and references — Google

- Avoid `const_cast` to remove `const`, except when implementing non-`const` getters in terms of `const` getters — Chromium

- Use `reinterpret_cast` to do unsafe conversions between pointer types, and from/to integer types — Google

* Use `std::bit_cast` to interpret the raw bits of a value using a different type of the same size — Google

Enumerators

Enumerators

※ Prefer enumerators over macros — CoreCpp

∗ Prefer `enum class` over plain `enum` — Unreal, µOS, CoreCpp

- Specify the *underlying type* and *enumerator values* only when necessary — CoreCpp1, CoreCpp2

```
enum class MyEnum : int16_t { Abc = 1, Def = 2 }; // bad
```

- Do not cast an expression to an enumeration type

  `Color c = static_cast<Color>(3)` — Hic

- Don't use `ALL_CAPS` for enumerators — CoreCpp

Arithmetic Types

Signed vs. Unsigned Integral Types

※ **Don't mix signed and unsigned arithmetic** — CoreCpp, μOS

∗ **Prefer *signed* integers whatever possible** — Google, μOS, CoreCpp,

∗ **Use *unsigned* integer only for bitwise operations** — Google, μOS, CoreCpp

∗ **Do not shift $\ll$ signed operands** — Hic, μOS, Clang-Tidy

∗ `size_t` vs. `int64_t`
  - Use `int64_t` instead of `size_t` for object counts and loop indices — Google
  - Use `size_t` for object and allocation sizes, object counts, array and pointer offsets, vector indices, and so on (to avoid overflow undefined behavior) — Chromium

▪ Do not apply unary minus to operands of `unsigned` type, e.g. `-1u` — Hic

Integral Types Conversion

* Avoid silent narrowing conversions, e.g, `int i += 0.1;` — Clang-Tidy

- Use brace initialization to convert/define *constant* arithmetic types (narrowing) e.g. `int64_t{MyConstant}` — Google
- Use `intptr_t` to convert raw pointers to integers — Google
- Be aware of implicit cast to `int`

Integral Types: Size and Other Issues

Size:

※ Except `int`, **use fixed-width integer type** (e.g. `int64_t`, `int8_t`, etc.)
Chromium, Unreal, Google, Hic, μOS, Clang-Tidy

∗ **Prefer *32/64-bit* signed integers over smaller data types** Google

- 64-bit integers add no/little overhead on 64-bit platforms

Other issues:

- Avoid redundant type, e.g. `unsigned int`, `signed int` Webkit

* **Floating point numbers shall not be converted to integers** except through use of standard library functions `std::floor`, `std::ceil` μOS, HIC

```
double d = ...;
int    i = d; // BAD, prefer std::floor(d)
```

* **Don't convert an expression of wider floating-point type to a narrower floating-point type** HIC

```
float f1 = 1.0;  // Bad
float f2 = 1.0F; // Ok
```

※ **Do not directly compare floating point** `==`, $<$, etc. HIC, μOS

- Floating-point literals should always have a radix point, with digits on both sides, even if they use exponential notation `2.0f` GOOGLE, WEBKIT (opposite)

Functions

※ **A function should perform a single logical operation** to promote simple understanding, testing, and reuse CoreCpp

※ **Split up large functions** (≥ 40) into logical sub-functions for improving readability and compile time Unreal, Google, CoreCpp, Clang-Tidy

∗ **Prefer pure functions**, namely functions that always returns the same result given the same input arguments (no external dependencies) and does not modify any state or have side effects outside of returning a value CoreCpp

* **Limit overloaded functions**. Prefer default arguments GOOGLE, CORECPP
  (don't use default arguments) HIC

* **Overload a function when there are no semantic differences between variants** GOOGLE

※ **Don't declare functions with an excessive number of parameters**. Use a wrapper structure instead — HIC, CORECPP, UNREAL, μOS

∗ **Specify all input-only parameters before any output parameters** — GOOGLE

∗ **Avoid adjacent parameters of the same type** → easy to swap by mistake — CORECPP

※ **Pass-by-`const`-*pointer* or *reference*** for input parameters are not intended to be modified by the function GOOGLE, UNREAL

▪ Use `std::optional` to represent optional `by-value` input parameters GOOGLE

∗ **Pass-by-reference** for input/output parameters CORECPP

∗ **Pass-by-reference** for output parameters, except rare cases where it is optional in which case it should be `passed-by-pointer` GOOGLE

- Prefer **pass-by-value** for small and trivially copyable types CoreCpp, Hic

- Don't **pass-by-`const`-value**, especially in the declaration (same signature of `pass-by-value`) Google
  (opposite) Autosar

* **Don't use rvalue references `&&` except for move constructors and move assignment operators** Google

* Boolean parameters should be avoided — Unreal

- Prefer `enum` to `bool` on function parameters — WebKit, Chromium
- Parameter names should be the same for declaration and definition — Clang-Tidy, HIC
- All parameters should be aligned if they do not fit in a single line (especially in the declaration)

```
void f(int         a,
       const int* b);
```

Functions Arguments

- Consider introducing variables to describe the meaning of arguments — Google

```
f(true); // BAD
bool enable_checks = true; // GOOD
f(enable_checks);
```

- Use argument comment to describe "magic number" arguments — Clang-Tidy, Google

```
void f(bool enable_checks);
f(/*enable_checks=*/true);
```

- All arguments should be aligned to the first one if they do not fit in a single line — Google

```
my_function(my var1, my_var2,
            my_var3);
```

* **Prefer to return values** rather than output parameters — Google, CoreCpp

* **Prefer to return `by-value`** — Google

- Prefer to return a `struct` /structure binding to return multiple output values — CoreCpp

- Don't return `const` values — CoreCpp

- 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`

※ **Transfer ownership with smart pointers**. Never return pointers for new objects. Use `std::unique_ptr` instead — Google, Chromium, CoreCpp

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

void FooConsumer(std::unique_ptr<Foo> ptr);      // correct
```

※ **Never return reference/pointer for local objects**. Return a pointer only to indicate a position — CoreCpp1, CoreCpp2, Google, SEI Cert

Function Specifiers

- If a function might have to be evaluated at compile time, declare it `constexpr`
  CoreCpp1, CoreCpp2
- Do not separate declaration and definition for `template` and `inline` functions
  Google
- Use `inline` only for small functions (e.g. ≤ 10 lines, no loops or switch statements)
  Google, Hic, CoreCpp
- Do not use `inline` when declaring a function (only in the definition)
- Do not use `inline` when defining a function in a class definition
  LLVM
- Use `noexcept` when it is useful and correct
  Google

Lambda Expressions

* **Prefer explicit captures** if the lambda may escape the current scope — Google

- Use default capture by reference (`[&]`) only when the lifetime of the lambda is obviously shorter than any potential captures — Google, CoreCpp

- Do not capture variables implicitly in a lambda, e.g. `[&]{body}` — Hic

- Omit parentheses for a C++ lambda whenever possible
  `[this] { return m_member; }` — Webkit
  (opposite) — Hic

```
int a[] { ++i  }; // Not a lambda
[]      { ++i; }; // A lambda
```

Structs and Classes

`struct` vs. `class`

* Use `struct` only for passive objects that carry data; everything else is `class` — Google, CoreCpp
* Use `class` rather than `struct` if any member is non-`public` — CoreCpp
* Prefer `struct` instead of `pair` or `tuple` — Google

Initialization

※ **Objects are fully initialized by constructor calls and all resources acquired must be released by the class's destructor**

Google, CoreCpp$_1$ CoreCpp$_2$, Hic, Clang-Tidy

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

Chromium, CoreCpp$_1$, CoreCpp$_2$ Clang-Tidy

∗ **Initialize member variables in the order of member declaration**

CoreCpp, Hic

∗ **Prefer initialization to assignment in constructors** CoreCpp

```
struct A {
    int _x;
    A(int x) { x = _x; } // bad
```

Braced Initializer Lists

- Initialize variables with `=`, constructors with `{}` — Mozilla
- Prefer braced initializer lists `{}` for constructors to clearly distinguish from function calls, avoid implicit narrowing conversion, and avoid the *most vexing parse* problem — CoreCpp1, CoreCpp2, CoreCpp3

```
void f(float x) {
   int v(int(x)); // function declaration
   int v{int(x)}; // variable
}
```

- Do not use braced initializer lists `{}` for constructors (at least for containers, e.g. `std::vector`). It can be confused with `std::initializer_list` — LLVM

Special Member Functions

* **Use delegating constructors to represent common actions for all constructors of a class** CoreCpp, Hic

* **Mark *destructor* and *move constructor/assignment*** `noexcept` CoreCpp$_1$, CoreCpp$_2$, Hic$_1$, Hic$_2$, SEI Cert, Clang-Tidy

* **Avoid implicit conversions**. Use the `explicit` keyword for conversion operators and constructors, especially single argument constructors Google, CoreCpp$_1$, CoreCpp$_2$, Hic, μOS, Clang-Tidy

=default, =delete

* **Indicate if a non-trivial class is copyable, move-only, or neither copyable nor movable** by using `= default` / `= delete` for constructors and assignment operators if not directly implemented

  Google, Mozilla, Chromium, CoreCpp

* **Prefer `= default` constructors** over user-defined / implicit default constructors

  Mozilla, Chromium, CoreCpp, Hic

* **Use `= delete` for mark deleted functions**

  CoreCpp, Hic

* **Don't return pointers or references to non-`const` objects from `const` methods** CHROMIUM

* **Use `const` functions wherever possible** GOOGLE, CHROMIUM, μOS, CLANG-TIDY

* Make a function a member only if it needs direct access to the representation of a class. Use a `static` function or a free-function otherwise CORECPP

- Don't define a `class` or `enum` and declare a variable of its type in the same statement, e.g. `struct Data /*...*/ data;` CORECPP

* **Do not overload operators with special semantics** `&&`, `^` `&&`, `||`, `,`, `&`, `operator""` (user-defined literals) GOOGLE, HIC, μOS

* **Prefer** to define non-modifying binary operators as **non-member functions** e.g. `operator==` GOOGLE, HIC

* Place **free-functions** that interact with a class in the **same namespace**, e.g. `operator==` CORECPP

* **Declare data members** `private`**, unless they are constants**. This simplifies reasoning about invariants GOOGLE, HIC

※ **Avoid virtual method calls in constructors** Google, CoreCpp, SEI Cert

※ **Default arguments are allowed only on *non-virtual* functions**
Google, CoreCpp, Hic, Clang-Tidy

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

* **Always use `override/final` function member keywords**
Google, Webkit, Mozilla, Unreal, Hic, Clang-Tidy, CoreCpp

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

* **Provide a virtual method anchor (`.cpp` implementation) for classes in headers** LLVM

* ***Multiple implementation inheritance* is discouraged** Google, Chromium, Hic, Clang-Tidy

* **Prefer *composition* to *inheritance*** Google

* **Inheritance should be `public`** Google

* **A polymorphic class should suppress public copy/move semantics** CoreCpp

- **Declare class data members in special way**
  - 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

  Examples:
  - Trailing underscore (e.g. `member_var_`) — Google, μOS, Chromium
  - Leading underscore (e.g. `_member_var`) — .NET
  - Public members (e.g. `m_member_var`, `mVar`) — Webkit, Mozilla
  - Static members (e.g. `s_static_var`, `sVar`) — Webkit, Mozilla

  Personal Comment: Prefer `_member_var` as I read left-to-right and is less invasive

- Class members are indented — Google

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

* **Declarations order** — GOOGLE
  (a) Types and type aliases
  (b) (Optionally, for structs only) non-static data members
  (c) Static constants
  (d) Factory functions
  (e) Constructors and assignment operators
  (f) Destructor
  (g) All other functions
  (h) All other data members

```
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 of the initializer list should be indented on a separate line, e.g. GOOGLE, WEBKIT

```
A::A(int x1, int y1) :
        x{x1},  // double indentation
        y{y1} {
   body
}
A::A(int x1, int y1) :
    : x{x1},
      y{y1} {
   body
}
```

- If possible, **avoid** `this->` keyword
- Prefer `empty()` method over `size()` to check if a container has no items

  Mozilla
- Do not use `get` for observer methods (`const`) without parameters, e.g. `get_size()` → `size()`

  Webkit
- Precede getters that return values via out-arguments with the word `get`

  Chromium
- Precede setters with the word `set`. Use bare words for getters

  Webkit, Chromium