Modern C++ Programming

9. Code Organization and Conventions

Federico Busato

University of Verona, Dept. of Computer Science
2018, v1.0

Agenda

- **Basic Concepts**
  - Translation Unit
  - Linkage
  - Global and local scope
- **Variables Storage**
  - Storage class specifiers
  - Storage duration
- **Dealing with Multiple Files**
  - One definition rule
  - Limit template instantiations
- **Namespace**
  - One definition rule
  - Namespace alias
  - Inline namespace
  - Anonymous namespace
- **C++ Project Organization**
  - Project Files
  - Include and library
- **Coding Style and Conventions**
  - File names and spacing
  - `#include`
  - Namespace
  - Variables
  - Functions
  - Structs and Classes
  - C++11/C++14 features
  - Control Flow
  - Entity names
  - Issues

Basic Concepts

Translation Unit

Header File and Source File

Header files allow to define interfaces (`.h`,`.hpp`, `..`), while keeping the implementation in separated **source files** (`.c`, `.cpp`, `...`).

Translation Unit

A **translation unit** (or compilation unit) is the basic unit of compilation in C++. It consists of the contents of a single source file, plus the contents of any header files directly or indirectly included by it. A single translation unit can be compiled into an object file, library, or executable program

Linkage

Linkage

Linkage refers to the visibility of symbols to the linker when processing files

Internal Linkage

Internal linkage refers to everything only in scope of a translation unit

External Linkage

External linkage refers to entities that exist beyond a single translation unit. They are accessible through the whole program, which is the combination of all translation units

Local and Global Scopes

Local Scope

Variables that are declared inside a function or a block are called local variables (**local scope**)

Global Scope

Variables that are defined outside of all the functions and hold their value throughout the life-time of the program are global variables (**global scope**)

Variables Storage

Storage Class

Storage Class

A **storage class** for a variable declarations is a type specifier that governs the lifetime, the linkage, and memory location of objects

- A given object can have only one storage class
- Variables defined within a block have automatic storage unless otherwise specified

Storage Class	Keyword	Lifetime	Visibility	Init value
Automatic	`auto`/no keyword	Code block	Local	Not defined
Register	`register`	Code block	Local	Not defined
Static	`static`	Whole program	Local	Zero-initialized
External	`extern`	Whole program	Global	Zero-initialized
Thread Local*	`thread_local`	Thread execution	Thread	Zero-initialized

*C++11

Storage Duration

The **storage duration** (or *storage class*) determines the *duration* of a variable, namely when it is created and destroyed

Storage Duration	Keyword	Allocation	Deallocation
Automatic	`auto`/no keyword	Code block start	Code end start
Static	`static`	Program start	Program end
Dynamic	`new`/`delete`	Memory allocation	Memory deallocation
Thread	`thread_local`	Thread start	Thread end

Storage Duration

Automatic storage duration. Scope variables (local variable). register or stack (depending on compiler, architecture, etc.). `register` hints to the compiler to place the object in the processor registers (deprecated in C++11)

Static storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends (`static` keyword at local or global scope)

Thread storage duration C++11. The object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. (`thread_local` can appear together with `static` or `extern`)

Dynamic storage duration. The object is allocated and deallocated per request by using dynamic memory allocation functions (`new/delete`)

Storage Class/Duration Examples

```
int        x;   // global scope*
static int y;   // static global scope*
extern int z;   // extern global scope*

thread_local int a;        // global* (each thread has its own value)
thread_local static int a; // static global* (each thread has
                           //                 its own value)

int main() {
    int           b;     // automatic
    auto          c = b; // automatic
    static int    d;     // local scope and whole program lifetime
    register int e;      // automatic (deprecated)
    thread_local int t; // automatic (each thread has its own value)

    auto array = new int[10]; // automatic and dynamic storage duration
}
```

static Variable Example

```cpp
#include <iostream>

void f() {
    static bool val = false;  // static
    if (val) {
        std::cout << "b";
        return;
    }
    val = true;
    std::cout << "a";
}

int main() {
    f(); // print "a"
    f(); // print "b"
    f(); // print "b"
}
```

Dealing with Multiple Translation Units

One Definition Rule (ODR):

(1) In any (single) **translation unit**, a `template`, `type`, `function`, or `object`, *cannot* have more than one definition

- Any number of declarations are allowed

(2) In the **entire program**, an `object` or `non-inline function` *cannot* have more than one definition

(3) A `template`, `type`, or (global) `inline functions`, can be defined in more than one translation unit. For a given entity, each definition must be the same

- `Non-extern objects` and `functions` in different translation units are different entities, even if their names and types are the same

`static` and `extern` keywords

`static` *global variable* or *functions* are visible only within the file (internal linkage)

- **`Non-static`** global variables or functions with the same name in different translation units produce name collision (or name conflict)

`extern` keyword is used to declare the existence of *global variables* or *functions* in another translation unit (external linkage)

- the variable or function must be defined in a one and only one translation unit

If, within a translation unit, the same identifier appears with both *internal* and *external* linkage, the behavior is undefined

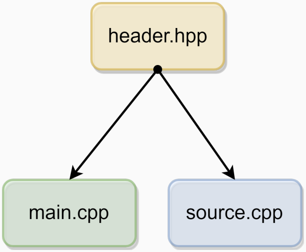
One Definition Rule (Example, points (1), (2))

header.hpp:

```
void f();
```

main.cpp:

```
#include "header.hpp"
#include <iostream>
int        a = 1;
static int b = 2; // internal linkage
extern int c;     // external linkage

int main() {
    std::cout << b;  // print 2
    std::cout << c;  // print 4
    f();             // print 5
}
```



source.cpp:

```
#include "header.hpp"
// int        a = 2;  // linking error !! (multiple definitions)
static int b = 5;  // ok
int        c = 4;  // ok

void f() {  std::cout << b; } // print 5
```

One Definition Rule (Example, point (3))

header.hpp:

```
inline void f() {} // the function is inline (no linking error)

template<typename T = void>
void g() {}        // the function is a template (no linking error)

using var_t = int; // types can be defined multiple times (no linking error)
```

main.cpp:

```
#include "header.hpp"  // included two times in the program!!
                       // (main.cpp and source.cpp)
int main() {
   f();
   g();
}
```

source.cpp:

```
#include "header.hpp"

void h() {
    f();
    g();
}
```

One Definition Rule (Classes)

header.hpp:

```
struct A {
    void f() {};  // declaration/definition inside struct (correct)
    void g();
};
void A::g() {}    // definition (wrong)!! multiple definitions

struct B {
    void f();     // declaration (correct) : definition is in source.cpp
};
```

main.cpp:

```
#include "header.hpp" // linking error !!
                      // multiple definitions of A::g()
int main() {
}
```

source.cpp:

```
#include "header.hpp" // linking error !!
                      // multiple definitions of A::g()
void B::f() {} // definition, ok
```

header.hpp:

```
template<typename T>
void f();

// template<>        // linking error (multiple definitions) included twice
// void f<int>() {}  // full specializations are standard functions
```

main.cpp:

```
#include "header.hpp"

int main() {
//  f<int>();     // linking error, f<int>() is not defined here
}
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void f() {}     // valid only in this translation unit

void g() {
    f<int>();   // ok
}
```

Good practice: declare full specializations in header

header.hpp:

```cpp
template<typename T>
void f();

void f<int>();  // informs the user that f<int>() has been specialized
//--------------------------------------------------------------------
template<typename T>
struct A {
    void g();
};
template<>
void A<int>::g(); // informs the user that A<int>::g() has been specialized
```

source.cpp:

```cpp
#include "header.hpp"
template<>
void f<int>() {}     // f() specialization

template<>
void A<int>::g() {} // A::g() specialization
```

Forward declaration is a declaration of an identifier for which a complete definition has not yet given

"*forward*" means that an entity is declared before it is used

Functions and **Classes** have external linkage by default

`main.cpp`:

```
void f();  // function forward declaration
class A;   // class    forward declaration

class B {
    friend A;  // ok, A is declared
//  A a;       // compiler error!! no definition (incomplete type)
};             // e.g. the compiler is not able to deduce the size of A
int main() {
    f();   // ok, f() is a function and not a variable
//  A a;   // compiler error!! no definition (incomplete type)
}
```

`source.cpp`:

```
void f() {}  // definition of f()
class A  {}; // definition of A()
```

Forward Declaration 2/2

Advantages:

- Forward declarations can save compile time, as `#include` force the compiler to open more files and process more input
- Forward declarations can save on unnecessary recompilation. `#include` can force your code to be recompiled more often, due to unrelated changes in the header

Disadvantages:

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change
- A forward declaration may be broken by subsequent changes to the library
- Forward declaring multiple symbols from a header can be more verbose than simply `#including` the header

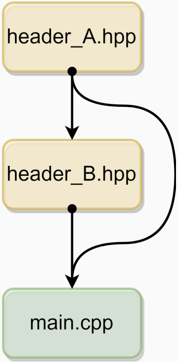
Full Story:
`google.github.io/styleguide/cppguide.html#Forward_Declarations`

The **include guard** avoids the problem of multiple inclusions of a header file in a translation unit

`#pragma once` (C++11,C++14) preprocessor directive force the current file to be included only once in a translation unit

`#pragma once` should be used in every header files

Common case:



header_A.hpp:

```
#pragma once   // it prevents "multiple definitions" linking error

struct A {
};
```

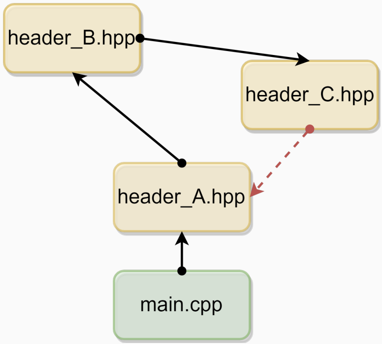
header_B.hpp:

```
#include "header_A.hpp"  // included here

struct B {
    A a;
};
```

main.cpp:

```
#include "header_A.hpp"  // .. and included here
#include "header_B.hpp"

int main() {
    A a;  // ok, here we need "header_A.hpp"
    B b;  // ok, here we need "header_B.hpp"
}
```

A **circular dependency** is a relation between two or more modules which either directly or indirectly depend on each other to function properly



Circular dependencies can be solved by using forward declaration, or better, by rethinking the project organization

header_A.hpp:

```cpp
#pragma once
#include "header_B.hpp"

class A {
    B* b;
};
```

header_B.hpp:

```cpp
#pragma once
#include "header_C.hpp"

class B {
    C* c;
};
```

header_C.hpp:

```cpp
#pragma once
#include "header_A.hpp"

class C {  // compile error!! "header_A" already included by "main.cpp"
    A* a;  // the compiler cannot view the "class C"
};
```

header_A.hpp:

```
#pragma once
class B;   // forward declaration

class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once
class C;   // forward declaration

class B {
   C* c;
};
```

header_C.hpp:

```
#pragma once
class A;   // forward declaration

class C {
    A* a;
};
```

Limit Template Instantiations

header.hpp:

```cpp
template<typename T>
void f();

//-------------------------------

template<typename T>
class A {
    void g();
    void h();
};

//-------------------------------

template<typename T>
class B {
    void g();
    void h();
};
```

source.cpp: (templates in a source file)

```cpp
#include "header.hpp"
template<typename T>
void f() {}

template void f<int>();
template void f<char>();
//-------------------------------
template<typename T>
void A<T>::g() {}

template<typename T>
void A<T>::h() {}

template class A<int>;
//-------------------------------
template<typename T>
void B<T>::g() {}
template<typename T>
void B<T>::h() {}

template void B<int>::f();
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
    f<int>();    // ok
    f<char>();   // ok
//  f<short>();  // compile error!! not specialized

    A<int>   a1; // ok
    a1.f();      // ok
    a1.g();      // ok
//  A<short> a2; // compile error!! not specialized

    B<int> b1;   // ok
    b1.f();      // ok
//  b1.g();      // compile error!! not specialized
}
```

Summary

- **header:** declaration of
  - structs/classes
  - functions, inline functions
  - template function/classes
  - extern global variables/functions
- **header implementation:** definition of (see next slides)
  - inline functions
  - template functions/classes
- **source file:** definition of
  - functions
  - templates full specialization
  - limited template instantiations
  - static global variables
  - extern variables/functions definition

Common Linking Errors

Very common *linking* errors:

- `undefined reference`

  Solutions:

  - Check if the right headers are included
  - Break circular dependencies with forward declarations

- `multiple definitions`

  Solutions:

  - `inline` function definition
  - Add `#pragma once` to header files
  - Use `extern` declaration for global variables
  - Place template definition in header file and full specialization in source files

Namespace

Overview

The problem: Named entities, such as variables, functions, and compound types declared outside any block has *global scope*, meaning that its name is valid anywhere in the code

Namespaces *allow to group named entities that otherwise would have global scope into narrower scopes, giving them* ***namespace scope*** *(where `std` stands for "standard")*

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Defining a Namespace

```
#include <iostream>
using namespace std;

namespace ns1 {
    void f() {
        cout << "ns1" << endl;
    }
}

namespace ns2 {
    void f() {
        cout << "ns2" << endl;
    }
}

int main () {
    ns1::f(); // print "ns1"
    ns2::f(); // print "ns1"
//  f();      // compile error!! f() is not visible
}
```

Namespace Conflits

```
#include <iostream>
using namespace std;

void f() {
     cout << "global" << endl;
}

namespace ns1 {
    void f() { cout << "ns1::f()" << endl; }
    void g() { cout << "ns1::g()" << endl; }
}

int main () {
    f();      // ok, print "global"
//  g();      // compile error!! g() is not visible
    using namespace ns1;
//  f();      // compile error!! ambiguous function name
    ::f();    // ok, print "global"
    ns1::f(); // ok, print "ns1::f()"
    g();      // ok, print "ns1::g()", only one choice
}
```

Nested Namespaces and Multiple files

header.hpp:

```cpp
#include <iostream>
namespace ns1 {
    void f() { cout << "ns1::f()" << endl; }
    namespace ns2 {
        void f() { cout << "ns1::ns2::f()" << endl; }
        void g() { cout << "ns1::ns2::g()" << endl; }
    }
}
```

main.cpp:

```cpp
#include "header.hpp"
namespace ns1 {     // the same namespace can be declared multiple times
    void g() {}     // ok
//  void f() {}     // compile error!! function name conflict with
}                   //                 header.hpp: "ns1::f()"

int main() {
   ns1::f();        // ok, print "ns1::f()"
   ns1::ns2::f();   // ok, print "ns1::ns2::f()"
   using namespace ns1::ns2;
   g();             // ok, print "ns1::ns2::g()"
}
```

Namespace Alias

Namespace alias allows declaring an alternate name for an existing namespace

```cpp
namespace ns1 {
    void g() {}
}

namespace ns_alias = ns1; // namespace alias

int main() {
    using namespace ns_alias;
    g();
}
```

`inline` Namespace

`inline` namespaces is a concept similar to library versioning. It is a mechanism that makes a nested namespace look and act as if all its declarations were in the surrounding namespace

```cpp
namespace ns1 {
    inline namespace V99 {
        void f(int) {}   // most recent version
    }
    namespace V98 {
        void f(int) {}
    }
}

using namespace ns1;

int main() {
    V98::f(1);  // call V98
    V99::f(1);  // call V99
    f(1);       // call default version (V99)
}
```

Anonymous Namespace

A namespace with no identifier before an opening brace produces an **unnamed/anonymous namespace**

Entities inside an anonymous namespace are used for declaring unique identifiers, visible in the same source file

Anonymous namespaces vs. static global entities

Anonymous namespaces allow type declarations and they are less verbose

main.cpp

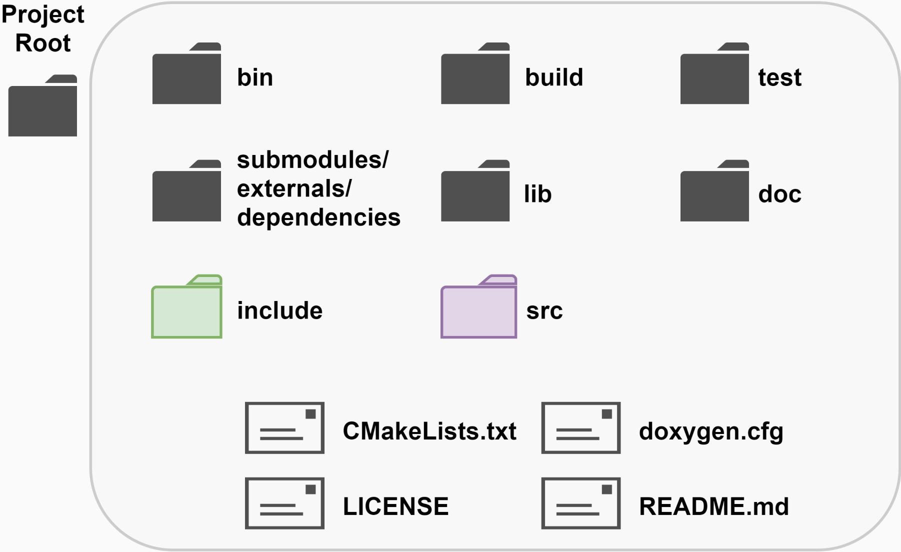
```cpp
#include <iostream>
namespace { // anonymous
    void f() { std::cout << "main"; }
}

int main() {
    f();   // ok, print "main"
}
```

source.cpp

```cpp
#include <iostream>
namespace { // anonymous
    void f() { std::cout << "source"; }
}

int g() {
    f();   // ok, print "source"
}
```

C++ Project Organization

Project Organization



Project Directories

bin Output executables

build All object (intermediate) file

data Files used by the executables

doc Project documentation

includes Project header files

src Project source files

test Source files for tests

lib External libraries or third party

submodules (also “externals” or “dependencies”)
External dependencies or submodules

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 (see next lecture)

`doxygen.cfg` Configuration file used by `doxygen` to generate the documentation (see next lecture)

*: `Markdown` is a language to generate text file with a syntax corresponding to a very small subset of HTML tags
`github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`

File extensions

Common C++ file extensions:

- **header** `.h .hh .hpp .hxx`
- **header implementation** `.i.h .i.hpp` EDALab
- **src** `.c .cc .cpp .cxx`
- **textually included at specific points** `.inc` Google

Common conventions:

- `.h .c .cc` Google
- `.hh .cc`
- `.hpp .cpp`
- `.hxx .cxx`

`src/include` directories

`src/include` directories should present exactly the same directory structure

Every directory included in `src` should be also present in `include`

Organization:

- **headers** and **header implementations** in `include`
- **source files** in `src`
- The **main** file (if present) can be placed in `src` and called `main.*` or placed in the project root directory with a generic name

Code Organization Example

- **include**
  - `MyClass1.hpp`
  - `MyTemplClass.hpp`
  - `MyTemplClass.i.hpp`
  - **subdir1**
    - `MyLib.hpp`
    - `MyLib.i.hpp` (template/inline functions)
- **src**
  - `MyClass1.cpp`
  - `MyTemplClass.cpp` (specialization)
  - **subdir1**
    - `MyLib.cpp`
- `main.cpp` (if necessary)
- `README.md`
- `CMakeLists.txt`
- `doxygen.cfg`
- `LICENSE`
- **build** (empty)
- **bin** (empty)
- **doc** (empty)
- **test**
  - `test1.cpp`
  - `test2.cpp`

Compiler includes and libraries flags

Specify the **`include` path** to the compiler:

```
g++ -std=c++14 -Iinclude main.cpp -o main
```

`-I` can be used multiple times

Specify the **library path** (path where search for static/dynamic libraries) to the compiler:

```
g++ -std=c++14 -L<library_path> main.cpp -o main
```

`-L` can be used multiple times

Specify the **library name** (e.g. `liblibrary.a`) to the compiler:

```
g++ -std=c++14 -llibrary main.cpp -o main
```

The predefined environmental variable in Linux/Unix for linking dynamic libraries/shared libraries is `LD_LIBRARY_PATH`

C++ Library

A **library** is a package of code that is meant to be reused by many programs

A **static library (`.a`)** consists of routines that are compiled and linked directly into your program. If a program is compiled with a static library, all the functionality of the static library becomes part of your executable

A **dynamic library**, also called a **shared library (`.so`)**, consists of routines that are loaded into your application at run-time. If a program is compiled with a dynamic library, the library does not become part of your executable. It remains as a separate unit

Coding Styles and Conventions

Most important rule:
BE CONSISTENT!!

“The best code explains itself”

GOOGLE

Coding Styles

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

Most popular coding styles:

- *LLVM Coding Standards*
  `llvm.org/docs/CodingStandards.html`
- *Google C++ Style Guide*
  `google.github.io/styleguide/cppguide.html`

File names and Spacing

File names:

- Lowercase Underscore (`my_file`) GOOGLE
- Camel UpperCase (`MyFile`) LLVM

Spacing:

※ Never use tab LLVM, GOOGLE,

- tab → 2 spaces GOOGLE
- tab → 4 spaces LLVM

※ Separate commands, operators, etc., by a space (`Google`, `LLVM`)

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

※ Line length (width) should be at most **80 characters** long (help code view on a terminal) LLVM, GOOGLE

Order of #include — LLVM, GOOGLE

(1) Class header (it is only one)

(2) Local project includes (in alphetical order)

(3) System includes (in alphetical order)

System includes are self-contained, local includes might not

Project includes — LLVM, GOOGLE

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

System includes — LLVM, GOOGLE

- should be indicated with <> syntax
  e.g. `#include <iostream>`

- **Use only necessary includes**
- **Include as less as possible, especially in header files**
- **Every includes must be self-contained** (the project must compile with every include order)
- **Report at least one function used for each include**
  `<iostream>   // std::cout, std::cin`
- **Use C++ headers instead of C headers:**
  `<cassert>` instead of `<assert.h>`
  `<cmath>` instead of `<math.h>`, etc.

Example:

```
#include "MyClass.hpp"             // MyClass
#include "my_dir/my_headerA.hpp"   // npA::ClassA, npB::f2()
#include "my_dir/my_headerB.hpp"   // np::g()
#include <iostream>                // std::cout
#include <cmath>                   // std::fabs()
#include <vector>                  // std::vector
```

Namespaces

Namespace guidelines:

- Avoid `using`-directives at global scope — LLVM, GOOGLE
- Limit `using`-directives at local scope and prefer explicit namespace specification — GOOGLE
- Always place code in a namespace — GOOGLE
- Avoid *anonymous* namespaces in headers

Style guidelines:

- The contents of namespaces are not indented — GOOGLE
- Close namespace declarations with `} // namespace <namespace_identifier>` — LLVM
- Close anonymous namespace declarations with `} // namespace anonymous`

Variables

- Avoid static and global variables LLVM, GOOGLE
- Prefer variable/iterator preincrement LLVM, GOOGLE
- Place a variables in the narrowest scope possible, and initialize variables in the declaration GOOGLE, ISOCPP
- Declaration of pointer variables or arguments may be placed with the asterisk *adjacent* to either the *type* or to the variable *name* for all in the same way
  `char* c;` `char *c;` GOOGLE
- Use fixed-width integer type (e.g. `int64_t`) GOOGLE
- Use brace initialization to convert arithmetic types (narrowing) e.g. `int64_t{x}` GOOGLE

Functions

Code guidelines:

- *Do not return pointers to local initialized heap memory!*
- Prefer return values rather than output parameters GOOGLE
- Limit overloaded functions GOOGLE
- Default arguments are allowed only on *non*-virtual functions GOOGLE

Style guidelines:

- All parameters should be aligned if possible (especially in the declaration) GOOGLE

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

- Parameter names should be the same for declaration and definition
- Do not use `inline` when declaring a function (only in the definition → `.i.hpp` files) LLVM

Code guidelines:

- Use a `struct` only for passive objects that carry data; everything else is a `class` LLVM, GOOGLE
- Objects that are fully initialized by constructor call GOOGLE
- Avoid multiple inheritance GOOGLE
- *Do not return pointers to local initialized heap memory!*

Minors:

- Use braced initializer lists for aggregate types `A{1, 2}`; LLVM, GOOGLE
- Do not use braced initializer lists for constructors LLVM
- Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors GOOGLE

Style guidelines:

- Class inheritance declarations order:
  `public`, `protected`, `private` GOOGLE
- First data members, then function members
- Declare class data members in special way*. Examples:
  - Trailing underscore (e.g. `member_var_`) GOOGLE
  - Leading underscore (e.g. `_member_var`) EDALAB, .NET
  - Public members (e.g. `m_member_var`)
- **Do not use '`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 visible 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();
};
```

Use C++11/C++14 features where possible

- Use *constant expressions* instead *macros* — Google
- `static_cast` `reiterpreter_cast` instead *old style cast* `(type)` — Google
- Use *range-for* loops whatever possible — LLVM
- Use `auto` type deduction to make the code more readable
  ```
  auto array = new int[10];
  auto var = static_cast<int>(var);
  ```
  LLVM, Google
- `nullptr` instead `0` or `NULL` — LLVM
- Use `[[deprecated]]` to indicate deprecated functions
- Use `using` instead `typedef`

Use C++11/C++14 features for classes:

- Use `explicit` constructors
- Use *defaulted* default constructor
- Use `override` function keyword
- Use `final` function keyword

- Multi-lines statements and complex conditions require curly braces GOOGLE
- Boolean expression longer than the standard line length requires to be consistent in how you break up the lines GOOGLE
- Curly braces are not required for single-line statements (but allowed) GOOGLE
- The `if` and `else` keywords belong on separate lines GOOGLE

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

if (c2) {  // required
    <statement1>
    <statement2>
}
```

```
if (complex_condition1 &&
    complex_condition2) { // required
    <statement1>
}

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

- Do not use `else` after a `return` LLVM
- Use *early exits* (`continue` , `break` , `return`) to simplify the code LLVM
- Turn predicate loops into predicate functions LLVM
- Merge multiple conditional statements

```
void f() {
    if (c1) {
        <statement1>
        return/break/continue;
    }   // error!!
    else
        <statement2>
}
void f() {
    if (c1) {
        <statement1>
        return/break/continue;
    }   // correct
    <statement2>
}
```

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

if (<condition1>) { // error!!
    if (<condition2>)
        <statement>
}
                // correct
if (<condition1> && <condition2>)
    <statement>
```

General rule: *avoid abbreviation and very long names*

variable Variable names should be nouns

- Uppercase Camel style e.g. `MyVar` LLVM
- Lowercase separated by underscore e.g. `my_var` GOOGLE

constant

- `k` prefix, e.g. `kConstantVar` GOOGLE
- Upper case separated by underscore `CONSTANT_VAR`

function Should be verb phrases (as they represent actions)

- Lowercase camel style, e.g. `myFunc()` LLVM
- Uppercase camel style for standard functions
  e.g. `MyFunc()` GOOGLE
  Lowercase separated by underscore for cheap functions
  e.g. `my_func()` GOOGLE, STD

namespace Lowercase separated by underscore
e.g. `my_namespace` GOOGLE, LLVM

typename Uppercase camel style (including `classes`, `structs`, `enums`, `typedefs`, etc.)
e.g. `HelloWorldClass` LLVM, GOOGLE

enum name
- `k` prefix
  e.g. `enum MyEnum { kEnumVar1, kEnumVar2 }` GOOGLE
- Uppercase camel style
  e.g. `enum MyEnum { EnumVar1, EnumVar2 }` LLVM
  - prefer `enum class`

macro Uppercase separated by underscore
e.g. `MY_MACRO` GOOGLE
- do not use macro for enumerator, constant, and functions

Other Issues

Do not use *RTTI* (`dynamic_cast`) or *exceptions* LLVM, GOOGLE

Code style

- Use common loop variable names
  - `i,j,k,l` used in order
  - `it` for iterators
- Use `true`, `false` for boolean variables instead numeric value `0`, `1`
- Prefer consecutive alignment

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

- Use the same line ending (e.g. `'\n'`) for all files
- Use `UTF-8` file encoding for portability
- Close files with a blank line

Code Documentation

- Each file should start with a license

LLVM

- Each file should include
  `@author` (name, surname, affiliation, email),
  `@version`, `@date`
- Use always the same style
- Comment style
  - Multiple lines

    ```
    /**
     * comment1
     * comment2
     */
    ```

  - single line

    ```
    /// comment
    ```