Modern C++ Programming

9. Code Organization

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 Translation Units**
  - One definition rule
  - Class
  - Function Template
  - Class Template
- **`#include` issues**
  - Forward declaration
  - Include guard
  - Circular dependency
  - Common linking errors
- **Namespace**
  - Overview
  - Namespace alias
  - Anonymous namespace
  - Inline namespace
- **C++ Project Organization**
  - Project Files
  - Include and library

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

Local and Global Scopes

Scope

The **scope** of a variable/function/object is the region of the code within the entity can be accessed

Local Scope

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

Local and Global Scopes

```
int var1;       // global scope

int f() {
    int var2;   // local scope
}

struct A {
    int var3;   // local scope
}

int main() {
    int var4;   // local scope
}
```

Linkage

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 *single* 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

`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

`static` Variable Example

```cpp
#include <iostream>

void f() {
    static int val = 1;  // static
    val++;
}

int main() {
    std::cout << f(); // print 1
    std::cout << f(); // print 2
    std::cout << f(); // print 3
}
```

Internal/External Linkage Example

```
int         var1 = 3;  // external linkage
                       // (in conflict with variable in other
                       //  translation units with the same name)

static int var2 = 4;   // internal linkage (visible only in the
                       //                  current translation unit)

extern      var3;      // external linkage
                       // (implemented in another translation unit)

void   f() {}          // external linkage (may conflict)
static f() {}          // internal linkage

extern void g();       // external linkage
                       // (implemented in another translation unit)
int main() {
}
```

`const` and `constexpr` notes

`const` at global scope implies `static`
→ internal linkage

`constexpr` implies `const`, which implies `static`
→ internal linkage

```
const     int var1 = 3;  // internal linkage
constexpr int var2 = 2;  // internal linkage

static const     int var3 = 3;  // internal linkage (redundant)
static constexpr int var4 = 2;  // internal linkage (redundant)

int main() {
}
```

Variables Storage

Storage Class

Storage Class Specifier

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 Class Examples

```
int                       v1;     // automatic
static                int v1 = 2; // static (global)
extern                int v3;     // external
thread_local          int v4;     // each thread has its own value
thread_local static int v5;       // each thread has its own value

int main() {
    int                 v6;     // automatic
    auto                v7 = 3; // automatic
    register int        v8;     // automatic (deprecated!)
    static int          v9;     // static (local)
    thread_local int v10;       // automatic (each thread has its own value)

    auto array = new int[10]; // automatic
}
```

Storage Duration

Storage Duration

The **storage duration** (or *duration 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`, global scope variable, `extern`	Program start	Program end
Dynamic	`new`/`delete`	Memory allocation	Memory deallocation
Thread	`thread_local`	Thread start	Thread end

Full Story:
`http://en.cppreference.com/w/cpp/language/storage_duration`

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 Duration Examples

```
int        v1;     // static duration
static int v2 = 4; // static duration
extern int v3;     // static duration

void f() {
    int              v4;      // automatic duration
    auto             v5 = 3;  // automatic duration
    static int       v6;      // static duration
    auto array = new int[10]; // dynamic duration (allocation)
} // array, v1, v2, v3, v6 variables deallocation (from stack)
  // the memory associeted with "array" is not deallocated!!

int main() {
    auto array = new int[10]; // dynamic duration (allocation)
    delete[] array;           // dynamic duration (deallocation)
}
// main end: v1, v2, v3, v6 deallocation
```

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 `inline functions`, can be defined in more than one translation unit. For a given entity, each definition must be the same

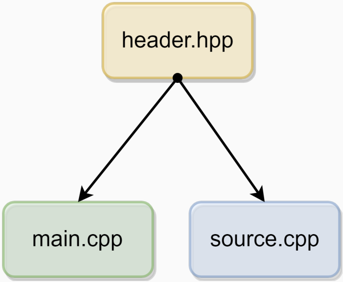
- Common case: same header included in multiple translation units
- `Non-extern objects` and `functions` in different translation units are different entities, even if their names and types are the same

One Definition Rule - Code Structure 1

First code structure:

one header, two source files $\rightarrow$ two translation units

the header is included in both translation units



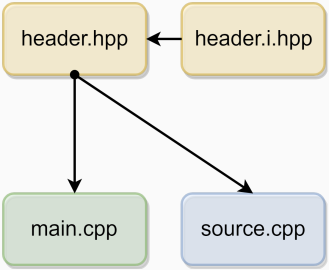
One Definition Rule - Code Structure 2

Second code structure:

two header, two source files $\rightarrow$ two translation units

one header for declarations (`.hpp`), and the other one for implementations (`.i.hpp`)

the header and the header implementation are included in both translation units*



* separate header declaration and implementation is not mandatory but, it allows to better organize the code

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

header.hpp:

```cpp
void f();
```

main.cpp:

```cpp
#include "header.hpp"
#include <iostream>

// internal linkage
int        a = 1;
static int b = 2;

// external linkage
extern int c;

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

source.cpp:

```cpp
#include "header.hpp"

// linking error !!
// (multiple definitions)
// int      a = 2;
static int b = 5;  // ok


int c = 4;  // ok

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

One Definition Rule (Example, point (3))

header.hpp:

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

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

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

main.cpp:

```cpp
#include "header.hpp"

int main() {
    f();
    g(3);
}
```

source.cpp:

```cpp
#include "header.hpp"

void h() {
    f();
    g(3);
}
```

→ no compile errors

Correct organization:

header.hpp:

```cpp
inline void f();    // declaration

template<typename T>
void g(T x);        // declaration

using var_t = int; // type
#include "header.i.hpp"
```

header.i.hpp:

```cpp
void f() {}         // definition

template<typename T>
void g(T x) {}      // definition
```

main.cpp:

```cpp
#include "header.hpp"

int main() {
    f();
    g(3);
}
```

source.cpp:

```cpp
#include "header.hpp"

void h() {
    f();
    g(3);
}
```

header.hpp:

```
class A {
public:
    void        f();
    static void g();
private:
    int        x;
    static int y;
};
```

main.cpp:

```
#include "header.hpp"
#include <iostream>

int main() {
    A a;
    std::cout << A.x;   // print 1
    std::cout << A::y; // print 2
}
```

source.cpp:

```
#include "header.hpp"

void A::f() {}
void A::g() {}

int A::x = 1;
int A::y = 2;
```

header.hpp:

```
struct A {
    int x1;
    int x2 = 3;
    int x3 { 4 };

    static int y;
//  static int y = 3; // compile error!!
//          must be initialized out-of-class

    const int z = 3; // only in C++11
//  const int z;     // compile error!!
//                      must be initilized

    static const int w1;
    static const int w2 = 4; // inline
//                              definition
};
```

source.cpp:

```
#include "header.hpp"

int         A::x1 = 1;
int         A::y  = 2;
const int   A::w1 = 3;
```

ODR Common Errors (Classes)

header.hpp:

```cpp
struct A {
    void f() {};  // declaration/definition inside struct (correct)
    void g();     // declaration
    void h();     // declaration
};

void A::g() {}    // definition (wrong)!! multiple definitions
```

main.cpp:

```cpp
#include "header.hpp"
// linking error !!
// multiple definitions of A::g()

int main() {
}
```

source.cpp:

```cpp
#include "header.hpp"
// linking error !!
// multiple definitions of A::g()

void A::h() {  // definition, ok
}
```

Function Template

Function Template

header.hpp:

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

#include "header.i.hpp"
```

header.i.hpp:

```cpp
template<typename T>
void f(T x) {} // definition
```

main.hpp:

```cpp
#include "header.hpp"

int main() {
    f(3);    // call f<int>()
    f(3.3f); // call f<float>()
    f('a');  // call f<char>()
}
```

source.cpp:

```cpp
#include "header.hpp"

void h() {
    f(3);    // call f<int>()
    f(3.3f); // call f<float>()
    f('a');  // call f<char>()
}
```

Function Template Specialization

header.hpp:

```
template<typename T>
void f(T x);    // declaration
```

main.hpp:

```
#include "header.hpp"

int main() {
   f(3);     // call f<int>()
   f(3.3f); // call f<float>()
// f('a');  // compile error!!
}  // specialization not exist
```

source.cpp:

```
#include "header.hpp"

template<typename T>
void f(T x) {}  // definition

// template specialization
template f<int>(int y);
template f<float>(float y);
```

Function Template Specialization Syntax

Alternative forms:

header.hpp:

```
template<typename T>
void f(T x);    // declaration

void f<int>(); // inform the specialization exists in
               // another translation unit (not mandatory)

// extern void f<int>(); // alternative form
```

ODR Common Errors (Function Templates)

header.hpp:

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

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

main.cpp:

```cpp
#include "header.hpp"

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

source.cpp:

```cpp
#include "header.hpp"

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

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

Class Template

Class Template

header.hpp:

```
template<typename T>
strut A {
    T    x;     // declaration
    void f();   // declaration
}
#include "header.i.hpp"
```

header.i.hpp:

```
template<typename T>
T A<T>::x = 3; // definition

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

main.hpp:

```
#include "header.hpp"

int main() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

source.cpp:

```
#include "header.hpp"

int g() {
   A<int>   a1; // ok
   A<float> a2; // ok
   A<char>  a3; // ok
}
```

Class Template Specialization

header.hpp:

```
template<typename T>
strut A {
    T    x;     // declaration
    void f();   // declaration
}
```

main.hpp:

```
#include "header.hpp"

int main() {
    A<int>  a1; // ok
//  A<char> a2; // compile error!!
}
```

source.cpp:

```
#include "header.hpp"

template<typename T>
int A<T>::x = 3;  // definition

template<typename T>
void A<T>::f() {} // definition

// template specialization
template class A<int>;
```

Summary

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

`#include` Issues

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 <u>external</u> 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()
```

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

header.hpp:

```
#ifndef HEADER_HPP  // include guard
#define HEADER_HPP

... many lines of code ...

#endif // HEADER_HPP
```

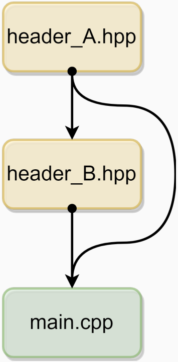
`#pragma once` preprocessor directive is an alternative to the the **include guard** to force current file to be included only once in a translation unit

- **#pragma once** is less portable but less verbose and compile faster than the **include guard**

The **Include guard**/**#pragma once** should be used in every header file

Include Guard 2/3

Common case:



Include Guard

3/3

header_A.hpp:

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

struct A {
};
```

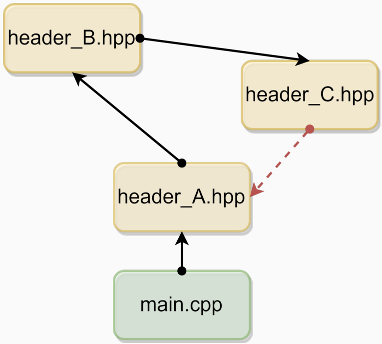
header_B.hpp:

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

struct B {
    A a;
};
```

main.cpp:

```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

Circular Dependencies

header_A.hpp:

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

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

header_B.hpp:

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

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

header_C.hpp:

```
#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"
};
```

Circular Dependencies (fix) 3/3

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;
};
```

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 or use `extern` declaration
  - Add `include guard`/`#pragma once` to header files
  - 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 very_very_long_namespace {
    void g() {}
}

namespace ns = very_very_long_namespace; // namespace alias

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

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"; }
}           // exteranl linkage, but
            // visible only internally
int main() {
    f();   // ok, print "main"
}
```

source.cpp

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

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

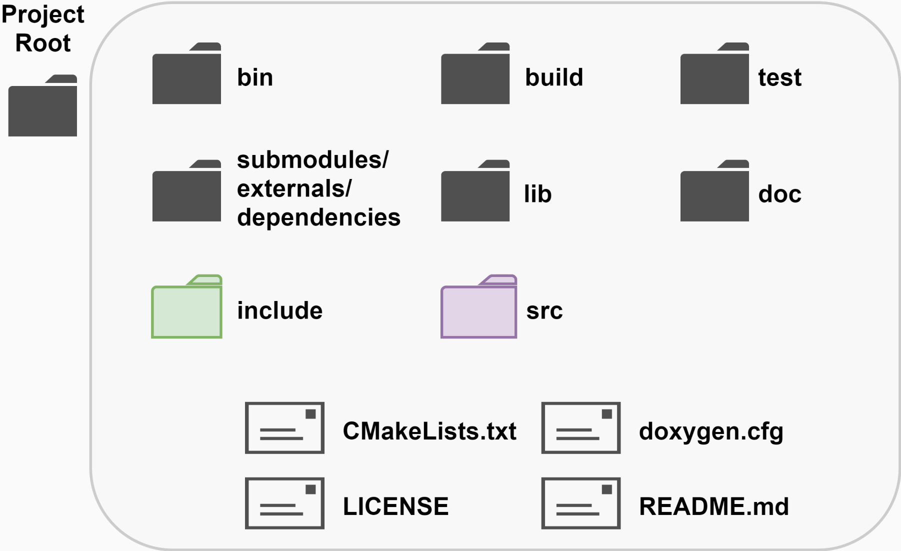
`inline` Namespace

`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)
}
```

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 <u>same</u> directory structure

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

<u>Organization</u>:

- **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

Common Rules

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

- `MyClass.hpp/MyClass.i.hpp/MyClass.cpp` with `class MyClass`
- `MyNP.hpp/MyNP.i.hpp/MyNP.cpp` with `namespace MyNP`

All code should be included in a **`namespace`**
→ avoid global namespace pollution

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`

How to Compile

Method 1

Compile all files together (naive):

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

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

`-I` can be used multiple times

Method 2

Compile each *translation unit* in a file object:

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

Link all file objects:

```
g++ -std=c++14 main.o source.o -o main.x
```

Compile with libraries

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