Modern C++ Programming

13. Translation Units II

Include, Module, and Compilation

Federico Busato

2025-02-03

Table of Contents

Table of Contents

Table of Contents

`#include` Issues

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

header.hpp:

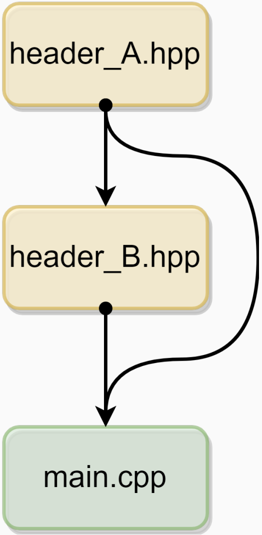
```
#ifndef HEADER_HPP  // include guard
#define HEADER_HPP

... many lines of code ...

#endif // HEADER_HPP
```

`#pragma once` preprocessor directive is an alternative to the **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

Common case:



header_A.hpp:

```cpp
#pragma once   // prevent "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"
}
```

Forward Declaration

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 defined

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

class A;   // class    forward declaration

int main() {
    f();   // ok, f() is defined in the translation unit
//  A a;   // compiler error no definition (incomplete type)
           // e.g. the compiler is not able to deduce the size of A
    A* a;  // ok
}

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

Forward Declaration vs. `#include`

Advantages:

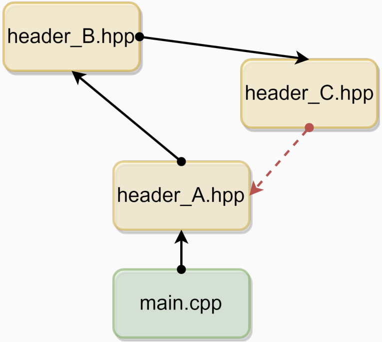
- Forward declarations can save compile time as `#include` forces the compiler to open more files and process more input
- Forward declarations can save on unnecessary recompilation. `#include` can force your code to be recompiled more often, due to unrelated changes in the header

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

google.github.io/styleguide/cppguide.html#Forward_Declarations

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:

```
#pragma once  // first include
#include "header_B.hpp"
class A {
    B* b;
};
```

header_B.hpp:

```
#pragma once  // second include
#include "header_C.hpp"
class B {
    C* c;
};
```

header_C.hpp:

```
#pragma once  // third include
#include "header_A.hpp"
class C {  // compile error "header_A.hpp": already included by "main.cpp"
    A* a;  // the compiler does not know the meaning of "A"
};
```

header_A.hpp:

```
#pragma once
class B;   // forward declaration
           // note: does not include "header_B.hpp"
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 and sources are included
  - Break circular dependencies (could be hard to find)

- `multiple definitions`

  Solutions:

  - `inline` function, variable definition or `extern` declaration
  - Add `include guard`/ `#pragma once` to header files
  - Place template definition in header file and full specialization in source files

C++20 Modules

The `#include` problem: *The duplication of work* - the same header files are possibly parsed/compiled multiple times and most of the compiled output is later-on thrown away again by the linker

C++20 introduces **modules** as a robust replacement for plain `#include`

Module (C++20)

A **module** is a set of source code files that are compiled independently of the translation units that import them

Modules allow defining clearer interfaces with a fine-grained control on what to *import* and *export* (similar to Java, Python, Rust, etc.)

- A Practical Introduction to C++20's Modules
- Modules the beginner's guide
- Understanding C++ Modules
- Overview of modules in C++

Less error-prone than `#include`:

- No effect on the compilation of the translation unit that *imports* the module
- Macros, preprocessor directives, and *non-exported* names declared in a module are not visible outside the module
- Declarations in the *importing* translation unit do not participate in overload resolution or name lookup in the *imported* module

Other benefits:

- **(Much) Faster compile time**. After a module is compiled once, the results are stored in a binary file that describes all the exported types, functions, and templates
- **Smaller binary size**. Allow to incorporate only the imported code and not the whole `#include`

Terminology

A **module** consists of one or more **module units**

A **module unit** is a *translation unit* that contains a `module` declaration

```
module my.module.example;
```

A **module name** is a concatenation of *identifiers* joined by dots (the dot carries no meaning) `my.module.example`

A **module unit purview** is the content of the translation unit

A **module purview** is the set of **purviews** of a given *module name*

Visibility and Reachability

Visibility of **names** instructs the linker if a symbol can be used by another translation unit. *Visible* also means *a candidate for name lookup*

Reachable of **declarations** means that the semantic properties of an entity are available

- Each *visible* declaration is also *reachable*
- Not all *reachable* declarations are also *visible*

Reachability Example

Common example: the members of a class are reachable (i.e. can be used) or the class size is known, but not the class type itself

```
auto g() {
    struct A {
        void f() {}
    };
    return A{};
}
//------------------------------------------------------------------------------

auto x = g();           // ok
// A y = g();           // compile error, "A" is unknown at this point
x.f();                  // ok
sizeof(x);              // ok
using T = decltype(x); // ok
```

Module Unit Types

- A **module interface unit** is a *module unit* that exports a symbol and/or *module name* or *module partition name*
- A **primary module interface unit** is a *module interface unit* that exports the *module name*. There must be one and only one *primary module interface unit* in a module
- A **module implementation unit** is a *module unit* that does not export a *module name* or *module partition name*

A **module interface unit** should contain only declarations if one or more *module implementation units* are present. A **module implementation unit** implements/defines the declarations of *module interface units*

Keywords

`module` specifies that the file is a *named module*

```
module my.module; // first code line
```

`import` makes a module and its symbols visible in the current file

```
import my.module; // after module declaration and #include
```

`export` makes symbols visible to the files that **`import`** the current module

- `export module <module_name>` makes visible all the exported symbols of a module. It must appear once per module in the *primary module interface unit*
- `export namespace <namespace>` makes visible all symbols in a namespace
- `export <entity>` makes visible a specific function, class, or variable
- `export {<code>}` makes visible all symbols in a block

`import` Example

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World";
}
```

Preprocessing size `-E`: ∼1MB

```cpp
import <iostream>;

int main() {
    std::cout << "Hello World";
}
```

Preprocessing size: 236B (x500)

Compile time: 2x (up to 10x) less

```
g++-12 -std=c++20 -fmodules-ts main.cpp -x c++-system-header iostream
```

export Example - Single Primary Module Interface Unit

my_module.cpp

```cpp
export module my.example;      // make visible all module symbols

export int f1() { return 3; } // export function

export namespace my_ns {      // export namespace and its content
int f2() { return 5; }
}

export {                      // export code block
int f3() { return 2; }
int f4() { return 8; }
}

void internal() {}            // NOT exported. It can be used only internally
```

`export` Example - Two Module Interface Units

`my_module1.cpp` *Primary Module Interface Unit*

```cpp
export module my.example; // This is the only file that exports all module symbols

export int f1() { return 3; } // export function
```

`my_module2.cpp` *Module Interface Unit*

```cpp
module my.example; // Module declaration but symbols are not exported

export namespace my_ns {      // export namespace
int f2() { return 5; }
}

export {                      // export code block7
int f3() { return 2; }
int f4() { return 8; }
}
```

`export` Example - Module Interface and Implementation Units

`my_module1.cpp` *Primary Module Interface Unit*

```cpp
export module my.example; // This is the only file that exports all module symbols

export int f1();          // export function

export {                  // export code block
int f3();
int f4();
}
```

`my_module2.cpp` *Module Implementation Unit*

```cpp
module my.example; // Module declaration but symbols are not exported

int f1() { return 3; }
int f3() { return 2; }
int f4() { return 8; }
```

Keyword Notes

`import`

- A **module implementation unit** can `import` another module, but cannot `export` any names. Symbols of the *module interface unit* are imported implicitly
- All `import` must appear before any declarations in that module unit and after `module;` a `export module` (if present)

`export`

- Symbols with *internal linkage* or *no linkage* cannot be exported, i.e. anonymous namespaces and `static` entities
- The `export` keyword is used in **module interface units** only
- The semantic properties associated to **exported** symbols become *reachable*

`export import` Declaration

Imported modules can be directly **re-exported**

```
export module main_module; // Top-level primary module interface unit

export import sub_module;  // import and re-export "sub_module"
```

```
export module sub_module; // Primary module interface unit

export void f() {}
```

```
import main_module;

int main() {
    f(); // ok, f() is visible
}
```

Global Module Fragment

A **global module fragment** (*unnamed module*) can be used to *include header files* in a *module interface* when importing them is not possible or preprocessing directives are needed

```
module;                    // start Global Module Fragment

#define ENABLE_FAST_MATH
#include "my_math.h"

export module my.module; // end Global Module Fragment
```

Macro definitions or other preprocessing directives are not visible outside the file itself

Private Module Fragment

A **private module fragment** allows a module to be represented as a single translation unit without making all the contents of the module reachable to importers

→ A modification of the *private module fragment* does not cause recompilation

If a module unit contains a *private module fragment*, it will be the only module unit of its module

```
export module my.example;
export int f();

module :private; // start private module fragment

int f() {          // definition not reachable from importers of f()
    return 42;
}
```

Header Module Unit

Legacy headers can be directly imported with `import` instead of `#include`

All declarations are implicitly *exported* and attached to the **global module (fragment)**

- Macros from the header are available for the *importer*, but macros defined in the *importer* have no effect on the *imported header*
- Importing compiled declarations is faster than `#include`

C++23 will introduce modules for the standard library

Module Partitions 1/2

A *module* can be organized in *isolated* **module partitions**

Syntax:

```
export module module_name : partition_name;
```

- *Declarations* in any of the **partitions** are visible within the entire module
- Like common modules, a *module partition* consists in one **module partition interface unit** and zero or more **module partition implementation units**
- *Module partitions* are not *visible* outside the module
- *Module partitions* do not *implicitly import* the module interface
- All names `exported` by *partition interface* files must be `imported` and `re-exported` by the *primary module interface file*

main_module.ixx

```
export module main_module;

export import :partition1; // re-export f() to importers of "main_module"
export import :partition2; // re-export g() to importers of "main_module"

export void h() { internal(); } // internal() can be directly used
```

partition1.ixx

```
export module module_name:partition1;

export void f() {}
```

partition2.ixx

```
export module module_name:partition2;

export void g() {}
void internal() {} // not exported
```

Compiling Multiple Translation Units

Fundamental Compiler Flags

Include flag: `g++ -I include/ main.cpp -o main.x`

- `-I` : Specify the **`include` path** for the project headers
- `-isystem` : Specify the **`include` path** for system (external) headers (warnings are not emitted)

They can be used multiple times

Important: *include* and *library* compiler flags, as well as multiple values in an environment variable, are evaluated in order from left to right. The first match suppress the other ones

Compile to a file object: `g++ -c source.cpp -o source.o`

Compile Methods

Method 1

Compile all files together (naive):

```
g++ main.cpp source.cpp -o main.out
```

Method 2

Compile each *translation unit* in a file object:

```
g++ -c source.cpp -o source.o
```

```
g++ -c main.cpp -o main.o
```

Multiple objects can be compiled in parallel

Link all file objects:

```
g++ main.o source.o -o main.out
```

Libraries in C++

Static Library

A **static library** is a set of object files (just the concatenation) that are directly <u>linked</u> into the final executable. If a program is compiled with a static library, all the functionality of the static library becomes part of final executable

- A static library cannot be modified without re-link the final executable
- Increase the size of the final executable
+ The linker can optimize the final executable (*link time optimization*)

Given the static library `my_lib`, the corresponding file is:

Linux	`libmy_lib.a`
Windows	`my_lib.lib`

Building Static Libraries

Steps to build a static library

- Compile object files for each translation unit (.cpp)
- Create the static library by using the **archiver** (**ar**) Linux utility

```
g++ source1.c -c source1.o
g++ source2.c -c source2.o
ar rvs libmystaticlib.a source1.o source2.o
```

Using Static Libraries

A *static library* has to be **linked** to the final executable:

- **Linux** `g++ -llibrary main.cpp -o main`
- **Windows** `msvc <path_to_library>/library.lib main.cpp /OUT:main.exe`

The directories where to search for *static* libraries at *compile-time* are specified with environment variables:

- **Linux** `LIBRARY_PATH` Search for `.a` files
- **Windows** `LIBPATH` Search for `.lib` files

It is also possible to specify additional *library paths* with compiler flags:

- **Linux** `g++ -L<library_path> main.cpp -o main`
- **Windows** `msvc /LIBPATH<library_path> main.cpp /OUT:main.exe`

Dynamic Library

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

+ A dynamic library can be modified without re-link: bug fixing, new functionalities
- Dynamic library functions are called outside the executable. Neither the linker nor the compiler can optimize the code between shared libraries and the final executable
- The environment variables must be set to the right shared library path, otherwise the application crashes at the beginning

Given the shared library `my_lib`, the corresponding file is:

Linux `libmy_lib.so`

Windows `my_lib.dll` + `my_lib.lib`

Building Dynamic Libraries

Steps to build a dynamic library

- Compile object files for each translation unit (`.cpp`). Since library cannot store code at fixed addresses, the compiler must generate *position independent code* (`-fPIC`)
- Create the dynamic library

```
g++ source1.c -c source1.o -fPIC
g++ source2.c -c source2.o -fPIC
g++ source1.o source2.o -shared -o libmydynamiclib.so
```

Using Dynamic Libraries 1/2

Dynamic libraries need to be available when the program executes (*run-time*). The program searches for dynamic libraries in the same directory and the paths specified in the following environment variables:

Linux Search for `.so` files

- `LD_LIBRARY_PATH` environment variable
- `/lib64` and `/usr/lib64`
- `RPATH` and `RUNPATH` fields with custom values embedded in the executable
- `/etc/ld.so.cache` cache of library locations created by the `ldconfig` command. Can be inspected by `ldconfig -p`

Using Dynamic Libraries 2/2

`Windows` Search for `.dll` files

- `PATH` environment variable
- Executable directory and current working directory
- `%SystemRoot%\System32`, `%SystemRoot%` system directories
- `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs` list of known DLLs

Application Binary Interface (ABI)

An **Application Binary Interface (ABI)** defines the low-level details of how programs composed of separately compiled modules work together. An ABI specifies how functions are called and how data is exchanged.

A **stable ABI** is essential to update the program's shared libraries without recompiling all the code

Some examples of ABI-breaking changes are changing the type or order of members within a `struct`, modifying the return type or parameters of a function, or adding a `virtual` function to a class that previously did not have one

An ABI can be also checked across different shared library/header versions with specific tools, such as `ABI Compliance Checker`

Demangling

Name mangling is a technique used to solve various problems caused by the need to resolve unique names

Transforming C++ ABI (Application binary interface) identifiers into the original source identifiers is called **demangling**

Example (linking error):

```
_ZNSt13basic_filebufIcSt11char_traitsIcEED1Ev
```

After demangling:

```
std::basic_filebuf<char, std::char_traits<char> >::~basic_filebuf()
```

How to demangle: `echo <name> | c++filt`

Online Demangler: `https://demangler.com`

Find Dynamic Library Dependencies

The **ldd** utility shows the shared objects (shared libraries) required by a program or other shared objects

```
$ ldd /bin/ls
      linux-vdso.so.1 (0x00007ffcc3563000)
      libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
      libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
      libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
      libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
      libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
      /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
      libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
      libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

Alternatively, `LD_DEBUG=libs` can be used to print search and load paths of shared libraries at runtime

Find Object/Executable Symbols ★

The `nm` utility provides information on the symbols being used in an object file or executable file

```
$ nm -D -C something.so
    w __gmon_start__
    D __libc_start_main
    D free
    D malloc
    D printf

# -C:  Decode low-level symbol names
# -D:  accepts a dynamic library
```

Find Object/Executable Symbols ★ 2/3

readelf displays information about ELF format object files

```
$ readelf --symbols something.so | c++filt
    ... OBJECT  LOCAL  DEFAULT   17 __frame_dummy_init_array_
    ... FILE    LOCAL  DEFAULT  ABS prog.cpp
    ... OBJECT  LOCAL  DEFAULT   14 CC1
    ... OBJECT  LOCAL  DEFAULT   14 CC2
    ... FUNC    LOCAL  DEFAULT   12 g()

# --symbols: display symbol table
```

Find Object/Executable Symbols ★ 3/3

`objdump` displays information about object files

```
$ objdump -t -C something.so | c++filt
... df *ABS*   ...  prog.cpp
...  O .rodata ...  CC1
...  O .rodata ...  CC2
...  F .text   ...  g()
...  O .rodata ...  (anonymous namespace)::CC3
...  O .rodata ...  (anonymous namespace)::CC4
...  F .text   ...  (anonymous namespace)::h()
...  F .text   ...  (anonymous namespace)::B::j1()
...  F .text   ...  (anonymous namespace)::B::j2()

# --t: display symbols
# -C:  Decode low-level symbol names
```

References and Additional Material

- 20 ABI (Application Binary Interface) breaking changes every C++ developer should know
- Policies/Binary Compatibility Issues With C++
- 10 differences between static and dynamic libraries every C++ developer should know