1-modules

June 3, 2024

1 C++20 improved compilation with Modules

1.1 Motivation

Despites many novelties, modern C++ still rely on the legacy preprocessor, which accumulates many defects.

Endless compilation time For each translation unit (body file to be compiled separately), the compilation must include recursively all included header files, then scan and compile the whole. There is a lot of redundant work, made worse with recent template-rich code. Pre-compilation techniques for header files have failed to address this problem.

Fragility The interpretation of an included file is very dependent on possible macros defined beforehand. This can lead to especially tricky bugs.

Code pollution To overcome certain problems related to the pre-processor, a certain number of practices have become mandatory, such as the *include guards*. This complicates the learning process for newcomers, and unnecessarily obscure the code.

Tools disturbance The pre-processing system singularly complicates the task of development tools, especially when they try to establish what constitutes the interface of a library .

1.2 General ideas

Already considered for C++11, postponed several times, the **C++ 20 modules** aim to suppress the use of the preprocessor, review the distinction between header and body files, better express the logical structure of the code, and of course **make compilation faster**.

1.2.1 Using a module

One can now **easily import modules**, as in any recent languages. If the standard library was written as a single module (this is not yet achieved), one may write:

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

1.2.2 Defining a module... the terminology

For a user-defined **named module**, the developer can simply put everything needed in a single file, said to be a **complete module**.

Yet, so to benefit from the efficiency of the external build systems, he will more likely split the logical **named module** into several physical files, called **module units**.

A typical setup, for each individual module, would be to have a *header-like* file, called **primary module interface unit**, and a *body-like* file, called a **module implementation unit**.

If one want to divide the module even more, he can subdivide the units into **module interface partitions** and **module implementation partitions**.

A **module unit** is made of one or several **module fragments** of different kinds: **global**, **purview**, **private**, **partitions**... Only some predefined mixtures of fragments are allowed, corresponding to different unit kinds.

To be noticed: some module units are kind of header/interface files, yet they now need to be compiled !

1.2.3 Diversity of strategies and implementations

> there is no one way of correctly using C++20 modules". *Niall Cooling*

The C++20 standard does not enforce a single way to split the modules into files, the file suffix, the use of auxiliary directories/files... the implementation of **C++ modules may differ significantly between compilers**.

In the following examples, we will investigate the GCC implementation, and we will stick to the usual *interface+implementation* pair of files for each module, similar to the old-school *header+body* files. This is somehow a modern implementation of the old "precompiled header files".

1.3 My first GCC module

GCC does not introduce some new files extensions, it is up to you to define some convention. Let's use a *pre-suffix* **`_mh`** for a header-like file (*Primary Module Interface Unit*), and **`_mb`** for a body-like file (*Module Implementation Unit*).

GCC requires the options **`-std=c++20 -fmodules-ts`**, and will produce an object file for each module file.

Warning: GCC requires the module interface to be compiled before the module implementation.

[6]:
```
%%file tmp.math_mh.cc

export module math ;

export int add( int first, int second ) ;
```

```
Overwriting tmp.math_mh.cc
```

The first statement `export module math` declare the module. The keyword `export` before `add()` makes it available to the module users.

[7]:
```
!rm -f tmp.math_mh.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mh.cc -o tmp.
 ↪math_mh.o
```

[8]:
```
%%file tmp.math_mb.cc

module math ;

int add( int first, int second )
 { return first + second ; }
```

```
Overwriting tmp.math_mb.cc
```

[9]:
```
!rm -f tmp.math_mb.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mb.cc -o tmp.
 ↪math_mb.o
```

[10]:
```
%%file tmp.main1.cpp

#include <iostream>

import math ;

int main()
 { std::cout<<add(40, 2)<<std::endl ; }
```

```
Overwriting tmp.main1.cpp
```

[11]:
```
!rm -f tmp.main1.exe && g++ -std=c++20 -fmodules-ts tmp.main1.cpp -o tmp.main1.
 ↪exe tmp.math_mh.o tmp.math_mb.o
```

[12]:
```
!./tmp.main1.exe
```

```
42
```

1.4 A module using another module

Withing a module file, any `import` or `include` statement must be made within a **global fragment**, introduced by the statement `module ;`.

[13]:
```
%%file tmp.answer_mh.cc

export module answer ;

export class Printer {
  public:
    void answer() ;
} ;
```

```
Writing tmp.answer_mh.cc
```

[14]:
```
!rm -f tmp.answer_mh.o && g++ -c -std=c++20 -fmodules-ts tmp.answer_mh.cc -o␣
 ↪tmp.answer_mh.o
```

[15]:
```
%%file tmp.answer_mb.cc

module ;

#include <iostream>

import math ;

module answer ;

void Printer::answer()
 { std::cout<<add(40, 2)<<std::endl ; }
```

```
Writing tmp.answer_mb.cc
```

[16]:
```
!rm -f tmp.answer_mb.o && g++ -c -std=c++20 -fmodules-ts tmp.answer_mb.cc -o␣
 ↪tmp.answer_mb.o
```

[17]:
```
%%file tmp.main2.cpp

import answer ;

int main()
 {
  Printer pr ;
  pr.answer() ;
 }
```

```
Writing tmp.main2.cpp
```

[18]:
```
!rm -f tmp.main2.exe && g++ -std=c++20 -fmodules-ts tmp.main2.cpp -o tmp.main2.
 ↪exe tmp.answer_mh.o tmp.answer_mb.o tmp.math_mh.o tmp.math_mb.o
```

[19]:
```
!./tmp.main2.exe
```

```
42
```

1.5 What about namespaces ?

The namespaces are orthogonal and independant of the modules. Yet, if one wants to mimic the Python approach, for example, he can associate each module with a namespace with same name.

[20]:
```
%%file tmp.math_mh.cc

export module math ;
```

```
namespace math {

  export int add( int first, int second ) ;

}
```

```
Overwriting tmp.math_mh.cc
```

[21]:
```
!rm -f tmp.math_mh.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mh.cc -o tmp.
 ↪math_mh.o
```

[22]:
```
%%file tmp.math_mb.cc

module math ;

namespace math {

  int add( int first, int second )
   { return first + second ; }

}
```

```
Overwriting tmp.math_mb.cc
```

[23]:
```
!rm -f tmp.math_mb.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mb.cc -o tmp.
 ↪math_mb.o
```

[24]:
```
%%file tmp.main3.cpp

#include <iostream>

import math ;

int main()
 { std::cout<<math::add(40, 2)<<std::endl ; }
```

```
Writing tmp.main3.cpp
```

[25]:
```
!rm -f tmp.main3.exe && g++ -std=c++20 -fmodules-ts tmp.main3.cpp -o tmp.main3.
 ↪exe tmp.math_mh.o tmp.math_mb.o
```

[26]:
```
!./tmp.main3.exe
```

```
42
```

[27]:
```
!rm -f tmp.math_mh.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mh.cc -o tmp.
 ↪math_mh.o
```

1.6 What about templates ?

It sounds like, as before, one must put the template body in the module interface.

[28]:
```
%%file tmp.math_mh.cc

export module math ;

export
template< typename T >
T add( T first, T second )
 { return first + second ; }
```

```
Overwriting tmp.math_mh.cc
```

[29]:
```
!rm -f tmp.math_mh.o && g++ -c -std=c++20 -fmodules-ts tmp.math_mh.cc -o tmp.
 ↪math_mh.o
```

[30]:
```
%%file tmp.main3.cpp

#include <iostream>

import math ;

int main()
 { std::cout<<add(40, 2)<<std::endl ; }
```

```
Overwriting tmp.main3.cpp
```

[31]:
```
!rm -f tmp.main3.exe && g++ -std=c++20 -fmodules-ts tmp.main3.cpp -o tmp.main3.
 ↪exe tmp.math_mh.o
```

[32]:
```
!./tmp.main3.exe
```

```
42
```

1.7 What about the standard library ?

GCC does not deliver yet the standard library as a set of modules.

1.8 Availability

Microsoft tools seem more advanced about this feature : * **Visual Studio 2017**, MSVC 19.25 & 19.28: partial support. * Clang 8, with `-fmodules-ts`: partial support. * GCC 11, with `-std=c++20 -fmodules-ts`: partial support.

1.9 Conclusion

Despites all C++ committee members have agreed on a common terminology and set of features, there are many ways to implement it, and it is largely embedded with the build tools. Thus, it is

unclear how a project developed for a given operating system, with a given set compilation and build tools, will be portable to another environment…

1.10 Questions ?

1.11 Sources

- Sticky Bits – Powered by Feabhas: C++20 modules with GCC11
- Cpp Reference, compiler support
- Cpp reference, modules
- Cpp Depend
- Modernes CPP, C++20: Modules
- Modernes CPP, C++20: More Details to Modules
- Minimal module support for the standard library