C++20 Modules – Complete Guide

© 2021 RNDr. Šimon Tóth[1]

@HappyCerberus @SimonToth

@Happy.Cerberus @SimonToth

@SimonToth83

Compilers and build systems are slowly starting to support C++20 modules. Perfect time for you to read this guide and benefit from the massive compilation speedups. This article reflects the state as of September 2021.

This article was originally published at:
`https://itnext.io/c-20-modules-complete-guide-ae741ddbae3d`

Also available in video form: `https://www.youtube.com/watch?v=WRCwciJ5MTE`

For all the examples and bash scripts, visit:
`https://github.com/HappyCerberus/article-cpp20-modules`

Hello World

First, a bit of a reminder of how headers work in C++. When you write `#include "header.h"`, the preprocessor will essentially copy-paste the processed content of `header.h` in place of the include. This does involve expanding any recursive includes, so you can easily end up with megabytes of text from a simple include. For example, with a simple hello world, we end up with almost 2MB of text.

hello_world.cc

```cpp
#include <iostream>
int main() {
    std::cout << "Hello World!\n";
}
```

```
> clang++ -std=c++20 -stdlib=libc++ -E hello_world.cc | wc -c
1956614
```

This is already problematic, but consider that the compiler will do this for every implementation file in your project. Processing common includes multiple times. Various vendors have recognised this waste and provide a non-standard solution using pre-compiled headers, where a common set of headers is processed once and then included everywhere.

Modules aim to solve the same problem, but in a standard way. So let's have a look at hello world again, but now with modules:

hello_world_modular.cc

```
import <iostream>;

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

```
> clang++ -std=c++20 -stdlib=libc++ -fmodules -fbuiltin-module-map -E
↪  hello_modular_world.cc | wc -c
239
```

This also serves as an extreme example of the time savings you can get by switching to modules:

```
> time clang++ -std=c++20 -stdlib=libc++ hello_world.cc
real 0m0.639s
user 0m0.584s
sys 0m0.058s

> time clang++ -std=c++20 -stdlib=libc++ -fmodules -fbuiltin-module-map
↪  hello_modular_world.cc
real 0m0.087s
user 0m0.052s
sys 0m0.037s
```

While Clang comes with a mapping of standard headers to modules. Using GCC, we need to compile `iostream` manually.

```
> g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
> g++ -std=c++20 -fmodules-ts hello_modular_world.cc
```

Writing modules

So we can already use the standard headers as modules. Let's have a look at writing our own modules now. As usual with C++, the language features are quite broad and non-prescriptive.

Each module must have a single file exporting this module. This file is designated the module interface unit. Modules are completely orthogonal to namespaces, so nothing prevents you from exporting symbols into the global namespace (like the `plus` function here) or export as part of a namespace (as with the `minus` function here).

advanced_mathematics.cc

```
export module AdvancedMathematics;

export auto plus(auto x, auto y) -> decltype(x+y) {
    return x + y;
}

export namespace AdvancedMathematics {
    auto minus(auto x, auto y) -> decltype(x-y) {
        return x - y;
    }
}

void this_function_will_not_be_exported() {}
```

main.cc

```
import <iostream>;
import AdvancedMathematics;

int main() {
    std::cout << "1+2 = " <<  plus(1,2) << "\n";
    std::cout << "3-2 = " << AdvancedMathematics::minus(3,2)
              << "\n";
}
```

Unexported symbols have private visibility, which is the opposite of normal C++ behaviour, where only symbols within anonymous namespaces are unexported.

Compiling this example with GCC is fairly straightforward. GCC will automatically emit a pre-compiled module when building a module interface unit. Clangs implementation is still lacking, and the `clang++` interface doesn't expose flags for emitting the pre-compiled module, so we need to pass `-emit-module-interface` directly to the clang compiler module using `-Xclang`. For further examples, I will be sticking with GCC.

```
# Compiling with GCC
> g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
> g++ -std=c++20 -fmodules-ts -c advanced_mathematics.cc
> g++ -std=c++20 -fmodules-ts -c main.cc
> g++ -std=c++20 main.o advanced_mathematics.o -o main

# Compiling with Clang
> FLAGS="-std=c++20 -stdlib=libc++ -fmodules -fbuiltin-module-map"
> clang++ $FLAGS -fmodules-ts -Xclang -emit-module-interface -c
↪  advanced_mathematics.cc -o AdvancedMathematics.pcm
> clang++ $FLAGS -c advanced_mathematics.cc
> clang++ $FLAGS -fprebuilt-module-path=. -c main.cc
> clang++ $FLAGS main.o advanced_mathematics.o -o main

> ./main
1+2 = 3
3-2 = 1
```

If desired, a module can be spread across multiple files — multiple *module units*. However,

only one of these files can export the module (and be the *module interface unit*).

spreadables.cc

```
export module Spreadables;

import <string>;

export {
    std::string butter();
    std::string jam();
}
```

butter.cc

```
module Spreadables;

import <string>;

std::string butter() {
    return "Spreading some butter.";
}
```

jam.cc

```
module Spreadables;

import <string>;

std::string jam() {
    return "Spreading some jam.";
}
```

bread.cc

```
import <iostream>;

import Spreadables;

int main() {
    std::cout << "Taking some bread.\n";
    std::cout << butter() << "\n";
    std::cout << jam() << "\n";
}
```

Using modules this way mirrors how one would use header and implementation files, except the module interface unit needs to be compiled to generate the pre-compiled module.

```
> g++ -std=c++20 -fmodules-ts -xc++-system-header iostream
> g++ -std=c++20 -fmodules-ts -xc++-system-header string
> g++ -std=c++20 -fmodules-ts -c spreadables.cc
> g++ -std=c++20 -fmodules-ts -c jam.cc
> g++ -std=c++20 -fmodules-ts -c butter.cc
> g++ -std=c++20 -fmodules-ts -c bread.cc
> g++ -std=c++20 bread.o spreadables.o jam.o butter.o -o bread

> ./bread
Taking some bread.
Spreading some butter.
Spreading some jam.
```

Module partitions

Another approach to spreading a module across multiple files are module partitions (currently unsupported by Clang). Partitions are only accessible by the main module, must be a single unit (single file), but can be an interface unit (export symbols) or just a unit. However, if the partition is a module interface unit, it must be re-exported inside the main module `export import :partition;`. The main module will have access to all partition symbols, even if they are not marked as exported.

main_internal.cc

```cpp
// This file is module unit for a partition internal of module main
module main:internal;

void internal_function() {} // will be visible in main module
```

main_base.cc

```cpp
// This file is module interface unit for a partition base
// of module main
export module main:base; // this must be re-exported

export void visible() {} // will be visible to users of main
void internal_base() {} // will be visible to main module only
```

main.cc

```cpp
// This file is module interface unit for module main
export module main;
// Import partition internal, cannot be re-exported
import :internal;
// Import partition base and re-export its exported symbols
export import :base;
```

Compile-time constructs vs Modules

You can use compile-time constructs (templates, auto, decltype, constexpr, consteval) anywhere inside module interfaces or module partitions.

main_internal.cc

```
module main:internal;

template <typename T>
T identity(T x) { return x; }
```

main.cc

```
export module main;
import :internal;

void function() {
    if (identity(3) != 3) abort();
}
```

However, when using multi-file modules, only the module interface unit contributes to the pre-compiled module. All compile-time construct therefore need to be placed in the interface file.

Dealing with legacy headers

If you cannot simply import your legacy header (notably when using feature macros), you can still use them in the global module fragment section.

legacy.cc

```
module; // start of global module fragment

#define _POSIX_C_SOURCE 200809L
#include <stdlib.h>

export module MyModule;
// etc...
```

Stylistic recommendations

Now for a bit more subjective part of this article. Here are my two recommendations that I think would make sense to follow as a style.

Firstly, regarding namespaces. Module names can contain the. symbol that does not have any semantic meaning. This allows us to structure the naming to match nested namespaces. My recommendation, therefore, is to structure the naming and exports of a module like the following:

```
export module my.nested.named.module;

export namespace my::nested::named::module {
    // exported content of the module
}
```

Secondly, regarding multi-file modules. Multi-file modules are supported in two ways. First, simply by having multiple module units and secondly by having module partitions. My recommendation here is the following:

1. Single file per module if possible.
2. If multiple files are beneficial for code readability, sensibly utilize module partitions (splitting off logically related functionality).

Thank you for reading

Thank you for reading this article. Did you enjoy it? If yes, pass it along to coleagues or classmates that might benefit from reading it.