C++20 Modules Cheatsheet

Authored by Lancern
Version 2023-10-22

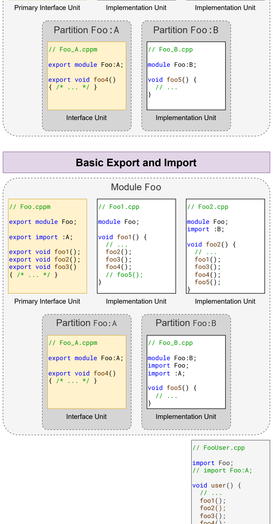
About File Extensions

Currently there are no consensus on the file extensions of module units. Different compilers uses different naming conventions:

- MSVC uses `.ixx` for module interface units and `.cpp` for module implementation units;
- GCC has no special extensions (yet) for module units;
- Clang uses `.cppm`, `.cxxm`, `.c++m`, and `.ccm` for module interface units, and `.cpp`, `.cxx`, `.c++`, and `.cc` for module implementation units.

This cheatsheet uses Clang's naming convention.

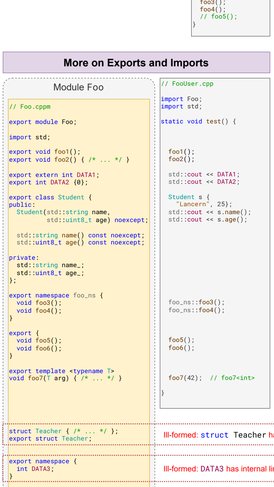
Basic Structure of a Module



- A *module unit* is a translation unit that contains a `module` declaration.
- If the `module` declaration is preceded by the `export` keyword, the module unit is an *interface unit*. Other module units are *implementation units*.
- Each module must have **exactly one** interface unit that is not a *module partition* (see below). This interface unit is called the module's *primary interface unit*.
- Each module may have any number of implementation units, including 0.
- Definitions can be put either in interface units or in implementation units. However, putting definitions in implementation units typically will better incremental build experiences.

- A module unit whose `module` declaration includes a *`module-partition`* (`:A`, `:B`, etc.) is called a *module partition*.
- Within a named module, no two module partitions can have the same *`module-partition`*. In other words, each partition can only have **exactly one** translation unit.
- All module partitions that are interface units must be exported directly or indirectly by the primary module unit.
- Module partitions that are implementation units cannot be exported in the primary module unit.

Basic Export and Import



- After importing a named module, all declarations and definitions exported by that module's primary interface unit will be available in the importing translation unit.
- Implementation units that are not module partitions automatically imports the module's primary interface unit.
- All module partitions that are interface units must be directly or indirectly exported in the primary interface unit.

- Module partitions do not automatically import the module's primary interface unit. So you have to manually import the module's primary interface unit if necessary.
- Module partitions can be imported by translation units in the same module, including other module partitions.

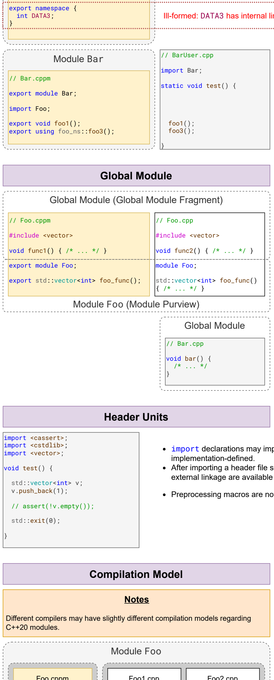
```cpp
// FooUser.cpp

import Foo;
// import Foo:A;

void user() {
  // ...
  foo1();
  foo2();
  foo3();
  foo4();
  // foo5();
}
```

- After importing a named module, all declarations and definitions exported by that module's primary interface unit will be available in the importing translation unit.
- You cannot import a module partition outside of its module.

More on Exports and Imports



- Any **declarations** that introduces a name with **external linkage** can be exported. These declarations will be available in the translation units that import the module's primary interface unit.
- For exported definitions, unlike conventional header files, they won't trigger "duplicate symbol" link error even if the interface unit is imported into multiple translation units.

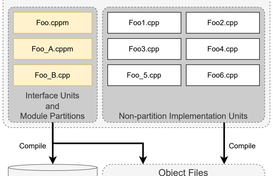
- Multiple declarations can be exported by a single `export` declaration with the special syntax as shown left.
- When exporting templates, you should put the definition of the template in the interface unit so that the compiler can instantiate the template definition when using it.

```cpp
struct Teacher { /* ... */ };
export struct Teacher;
```

Ill-formed: `struct Teacher` has *module linkage* and you cannot export a name with module linkage.

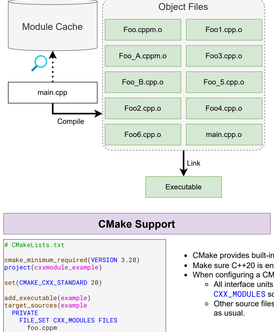
```cpp
export namespace {
  int DATA3;
}
```

Ill-formed: DATA3 has internal linkage and you cannot export a name with internal linkage.

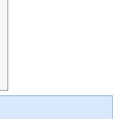


- You can re-export declarations imported from other modules.

Global Module



- The sequence of tokens before the first `module` declaration in a module unit are called *global module fragment*. Declarations in the global module fragment belong to the *global module*.
- The sequence of tokens since the first `module` declaration in a module unit are called *module purview*.
- If your module unit needs to `#include` some header files, you should put the `#include` directives in the global module fragment.



- All declarations in a translation unit that is not a module unit also belong to the global module.

Header Units

```cpp
import <cassert>;
import <cstdlib>;
import <vector>;

void test() {

  std::vector<int> v;
  v.push_back(1);

  // assert(!v.empty());

  std::exit(0);

}
```

- `import` declarations may import header files as *header units*. However, the set of header files that can be imported is implementation-defined.
- After importing a header file supported by the implementation, all declarations in the header file that introduces a name with external linkage are available in the importing translation unit.

- Preprocessing macros are not declarations. You cannot use preprocessing macros defined in the header file.

Compilation Model

Notes

Different compilers may have slightly different compilation models regarding C++20 modules.

- During compilation, interface units and partition implementation units will be tokenized, preprocessed, parsed and saved in the *module cache* in a format that can be read efficiently by the compiler.
- As usual, all translation units will be compiled into an object file.

- If some translation unit imports a module, the compiler will search the module cache and load the module's interface from the corresponding cache entry.
- If the module is not yet compiled and saved in the module cache, some compilers may be able to locate the module units of that module and recursively compile that module.

- The final output (executables, libraries, etc.) is produced by linking all the object files as usual.

CMake Support

```cmake
# CMakeLists.txt

cmake_minimum_required(VERSION 3.28)
project(exemodule_example)

set(CMAKE_CXX_STANDARD 20)

add_executable(exemple)
target_sources(exemple
  PRIVATE
    FILE_SET CXX_MODULES FILES
      foo.cppm
      foo_a.cppm
      bar.cppm
)
target_sources(exemple
  PRIVATE
    main.cpp
    foo.cpp
    bar.cpp
    non_module.cpp
)
```

- CMake provides built-in support for C++20 modules since CMake 3.28.
- Make sure C++20 is enabled via `CMAKE_CXX_STANDARD`.
- When configuring a CMake target that uses C++20 modules:
  - All interface units (including primary interface units and partition interface units) must be added to the target's `CXX_MODULES` source file set via the `target_sources` command.
  - Other source files (including implementation units and non-module translation units) can be added to the target as usual.

Tips

If your project does not use C++20 modules, you can set `CMAKE_CXX_SCAN_FOR_MODULES` to `OFF` to avoid unnecessary module dependency scans and gain potential build speed improvements.