

+ 23

C++ Modules:

Getting Started Today

ANDREAS WEIS



20
23



C++ Modules - Getting Started Today

Andreas Weis

Woven by Toyota

CppCon 2023



Introduction

About me - Andreas Weis (he/him)

-    ComicSansMS

-  Co-organizer of the Munich C++ User Group

- Currently working as a Vehicle Architect for Woven by Toyota



A familiar example...

— *File:* my_function.hpp

```
char const* my_function();
```

— *File:* my_function.cpp

```
char const* my_function() {  
    return "Hello from function!";  
}
```

— *File:* main.cpp

```
#include <my_function.hpp>  
#include <print>  
  
int main() {  
    std::println("{} ", my_function());  
}
```

#include can happen anywhere

— *File: a.hpp*

```
inline int my_function  
#include <a_impl.hpp>
```

— *File: a_impl.hpp*

```
{  
    return 42;  
}
```

Including files twice does not work

— *File: a.hpp*

```
class A {};
```

— *File: main.cpp*

```
#include <a.hpp>
```

```
#include <a.hpp>    // redefiniton error!
```

```
int main() {  
}
```

Included files are not isolated from the surrounding state

— *File: a.hpp*

```
class A {  
private:  
    char const* u_cant_touch_this() {  
        return "Preprocessor hits me so hard";  
    }  
};
```

— *File: main.cpp*

```
#define private public  
#include <a.hpp>  
#undef private  
int main() {  
    std::println("{} ", A{}.u_cant_touch_this());  
}
```


Included files do not need to be self-contained

— *File: a.hpp*

```
class A {  
    std::vector<int> numbers;  
};
```

— *File: main.cpp*

```
#include <vector>  
#include <a.hpp>  
  
int main() {  
}
```

Include files are compiled again for each translation unit

— *File: a.cpp*

```
#include <massive_header.hpp>
```

```
// [...]
```

— *File: b.cpp*

```
#include <massive_header.hpp>
```

```
// [...]
```

Hello Modules!

— *File:* module.cpp

```
export module my_module;
```

```
export char const* my_function() {  
    return "Hello Modules!";  
}
```

— *File:* main.cpp

```
#include <print>  
import my_module;
```

```
int main() {  
    std::println("{} ", my_function());  
}
```

Exporting things

```
// functions  
export int  getNumber();  
  
// types  
export class SomeType;  
  
// templates  
export template<typename T>  
T combine(T n1, T n2);  
export template<typename T>  
class MyTemplatedType;
```

Exporting things

```
export namespace a {  
    void is_exported();  
}  
  
namespace a {  
    void is_not_exported();  
}  
  
namespace a {  
    void will_be_exported_in_b();  
}  
  
export namespace b {  
    export using a::will_be_exported_in_b;  
}
```

Exporting things

```
export {  
    void will_be_exported();  
  
    void will_also_be_exported();  
  
    struct WillAlsoBeExported {  
        // [...]  
    };  
  
    // the following will not compile:  
    static void no_internal_linkage();  
}
```

Exporting things

— *File:* module1.cpp

```
export module A;
```

```
export int foo() { return 42; }
```

— *File:* module2.cpp

```
export module B;
```

```
export import A;
```

— *File:* main.cpp

```
import B;
```

```
int main() {  
    return foo();  
}
```

Not exported does not mean unreachable

— *File: module.cpp*

```
export module m;  
class NotExported { int i = 42; };  
export NotExported getNotExported()  
{ return {}; }
```

— *File: main.cpp*

```
import m;  
int main() {  
    int const ii = getNotExported().i;  
}
```

This is not new!

```
auto getS()  
{ struct S { int i = 42; }; return S{}; }
```


Daniela Engert - The three secret spices of C++ Modules - Visibility, Reachability, Linkage

Meeting C++ online



Daniela Engert

SELECTIVE VISIBILITY

```
1 // module interface TU
2 export module M;
3
4 struct S {           // PDL, introduces entity 'S', not exported
5     int s_ = 1;
6 };
7
8 export S foo(int); // PDL, introduces entity 'foo' and exports name 'foo'
9                  // PDL, names visible entity 'S'
10
11 // module implementation TU
12 module M;
13
14 S foo(int x) {       // PDL, names visible entities 'S' and 'foo'
15     return S{x};
16 }
17
18 // client TU
19 import M;            // introduces entity 'foo' by BMI, exported from module M
20
21 auto y = foo(1);     // PDL, names entity 'foo'
22                    // result type of 'foo' is totally invisible
```

- without modules, **total invisibility** of entities declared within a TU is **impossible**
- moving declarations from headers into modules makes them totally invisible from the outside
- exporting names from a module and importing them **controls** the extent to which names become **visible** in the translation unit importing the module's interface.



Daniela Engert - The three secret spices of C++ Modules

Different shapes of modules

Primary Module Interface Unit

— *File:* my_module.cpp

```
export module my_module;  
  
export char const* my_function() {  
    return "Hello Modules!";  
}
```

Module Implementation Unit

— *File:* my_module.cpp

```
export module my_module;
```

```
export char const* my_function();
```

— *File:* my_module_impl.cpp

```
module my_module;
```

```
char const* my_function() {  
    return "Hello Modules!";  
}
```

Module Interface Partitions

— *File:* my_module.cpp

```
export module mice;  
  
export import :pinky;  
export import :the_brain;
```


— *File:* my_module_p1.cpp

```
export module m:pinky;  
export void narf() {}
```


— *File:* my_module_p2.cpp

```
export module m:the_brain;  
export void take_over_the_world() {}  
struct SecretMasterplan {};
```

We're not talking about Header Units



2023
MAY 8-12
Aspen, Colorado, USA




Daniel Ruoso


C++ Modules
*The Challenges of
Implementing Header Units*


Is that worth implementing?

- Requiring changes to existing libraries for them to deploy new metadata so they can be imported as header units seem to defeat the purpose; the library author can just offer a wrapper module instead
- The emulation of the import in the dependency scanning has unknown performance characteristics at scale
- A human trying to understand the preprocessor state will have to do the same emulation the compiler does (Raise your hand if you want to teach this to every current and future C++ engineer?)

TechAtBloomberg.com
© 2023 Bloomberg Finance L.P. All rights reserved.


74





CppNow.org

If you want to know more...

Daniela Engert - So you want to use C++ Modules... cross-platform? (NDC TechTown 2023)

MODULE TU TYPES & FEATURES

| | Defines interface | contributes to interface | implicitly imports interface | part of module purview | part of global module | exports MACROs | creates BMI | contributes to PMIF | fully isolated |
|--------------------------|-------------------|--------------------------|------------------------------|------------------------|-----------------------|----------------|-------------|---------------------|----------------|
| Primary Module Interface | ✓ | ✓ | | ✓ | ● | ✗ | ✓ | ✓ | ✓ |
| Mod. Implementation unit | ✗ | ✗ | ✓ | ✓ | ● | ✗ | ✗ | ✗ | ✓ |
| Interface partition | ✗ | ✓ | ✗ | ✓ | ● | ✗ | ✓ | ✓ | ✓ |
| Internal partition | ✗ | ✗ | ✗ | ✓ | ● | ✗ | ✓ | ◆ | ✓ |
| Private module fragment | | ✗ | | ✓ | | ✗ | | ✗ | ✓ |
| Header unit | ✓ | ✓ | | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |

✓ unconditionally ● if a GMF exists in the TU ◆ if TU's BMI is (transitively) imported into the PMIF

11

Building modules code with CMake

Building with CMake - Old School

```
cmake_minimum_required(VERSION 3.27)
project(my_project)

add_executable(my_executable)
target_sources(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/my_src.cpp
    ${PROJECT_SOURCE_DIR}/inc/my_header1.hpp
    ${PROJECT_SOURCE_DIR}/inc/my_header2.hpp
)
target_include_directories(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/inc)
```

Building with CMake - File Sets (since v3.23)

```
cmake_minimum_required(VERSION 3.27)
project(my_project)

add_executable(my_executable)
target_sources(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/my_src.cpp
    PUBLIC
    FILE_SET HEADERS
    BASE_DIRS ${PROJECT_SOURCE_DIR}/inc
    FILES
    ${PROJECT_SOURCE_DIR}/inc/my_header1.hpp
    ${PROJECT_SOURCE_DIR}/inc/my_header2.hpp
)
```

Building with CMake - Modules (experimental as of v3.37)

```
cmake_minimum_required(VERSION 3.27)
project(my_project)
set(CMAKE_CXX_STANDARD 20)

add_executable(my_executable)
target_sources(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/my_src.cpp
    PUBLIC
    FILE_SET MODULES
    BASE_DIRS ${PROJECT_SOURCE_DIR}/mod
    FILES
    ${PROJECT_SOURCE_DIR}/mod/my_module.cpp
)
```

Some boilerplate required...

CMake Modules support is currently experimental. The details are boring and bound to change quickly.

Refer to <https://t.ly/Dl8yE> for the current state.

For CMake 3.27 add:

```
set(CMAKE_EXPERIMENTAL_CXX_MODULE_CMAKE_API  
    aa1f7df0-828a-4fcd-9afc-2dc80491aca7)
```

For Clang you may also need to add:

```
set(CMAKE_CXX_EXTENSIONS OFF)
```

Use the latest tools!

- Absolute latest CMake (3.27)
- Latest Visual Studio 2022 (at least 19.34)
- Ninja 1.11
- Clang at least 17, prefer trunk
- gcc trunk

Even with the latest tools there are still plenty of bugs and inconsistent behavior between compilers!

A few things to keep in mind

Module source file or regular source file?

- If it has an `export module` somewhere → Module
- If it is a module partition → Module
- Otherwise → Regular.

Which file extension?

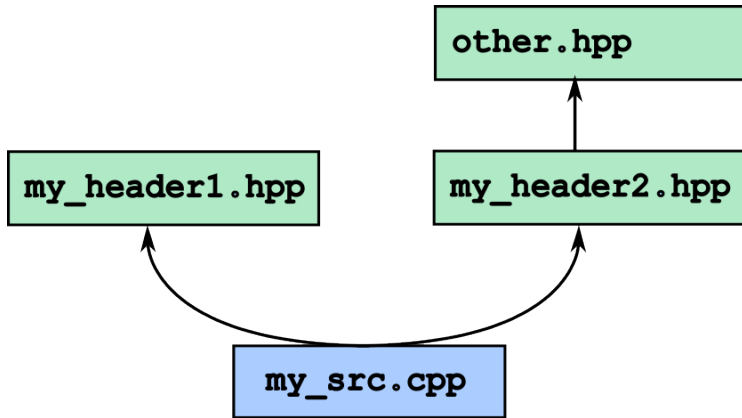
- Many different extensions started appearing in the compilers: `.ixx`, `.cppm`, `.cxxm`, `.c++m`, `.ccm`.
- With CMake you don't need to use any of them!
- If you decide to use them, be sure to *only* use them for module source files (as defined above)

What is a build system anyway?

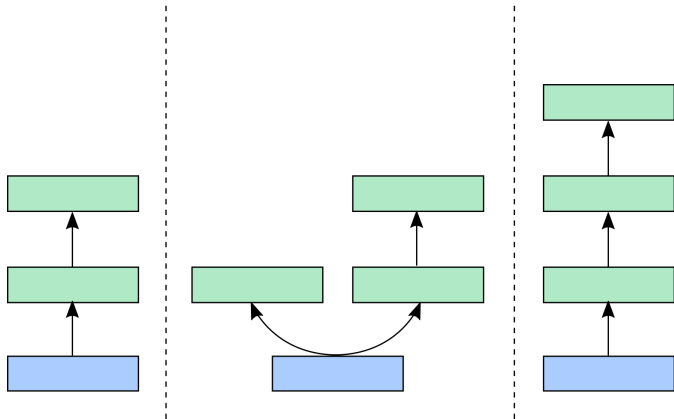
```
$ g++ -o my_src.o -I . -c my_src.cpp
```

```
$ g++ my_executable.o -o my_executable
```

Tracking dependencies



Tracking dependencies



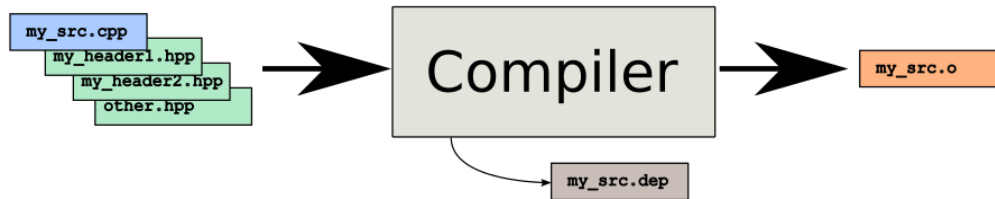
Tracking changes to header files

```
$ g++ -o my_src.dep -I . -M -c my_src.cpp
```

— *File:* my_src.dep

```
my_src.o: my_src.cpp \  
    my_header1.hpp  
    my_header2.hpp  
    other.hpp
```

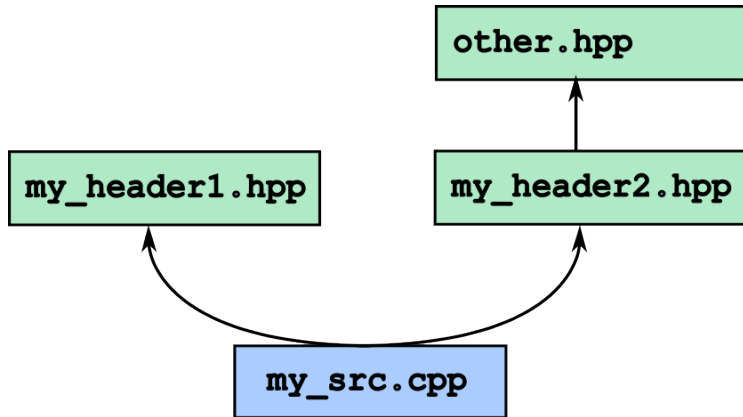
Tracking changes to header files



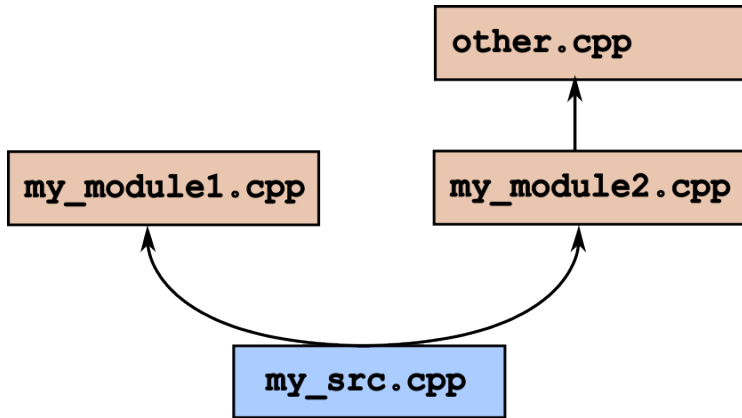
— *File:* my_src.dep

```
my_src.o: my_src.cpp \  
    my_header1.hpp \  
    my_header2.hpp \  
    other.hpp
```

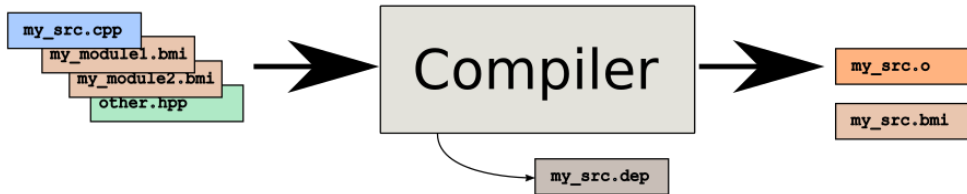
What changes for the build system?



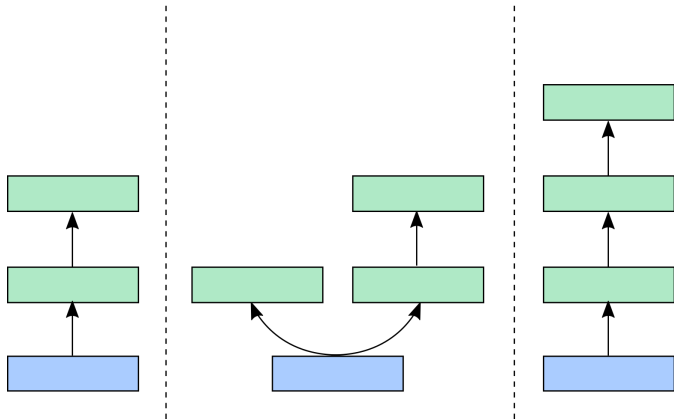
What changes for the build system?



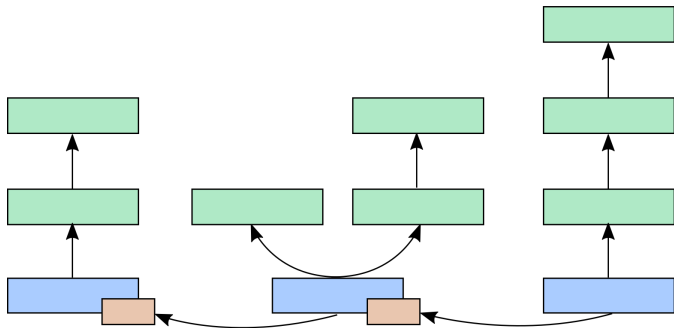
What changes for the build system?



What changes for the build system?



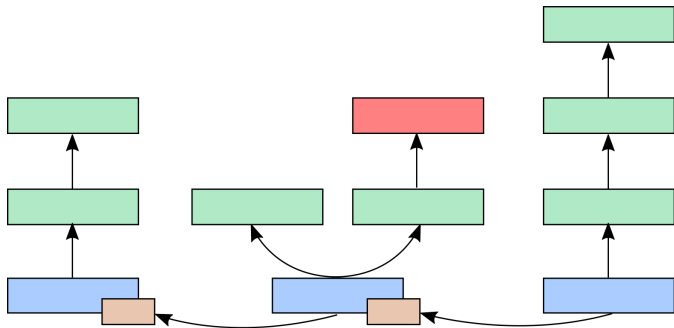
What changes for the build system?



What changes for the build system?

```
add_executable(my_executable)
target_sources(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/my_src.cpp
    PUBLIC
    FILE_SET MODULES
    BASE_DIRS ${PROJECT_SOURCE_DIR}
    FILES
        ${PROJECT_SOURCE_DIR}/my_module1.cpp
        ${PROJECT_SOURCE_DIR}/my_module2.cpp
        ${PROJECT_SOURCE_DIR}/other.cpp
)
```

What changes for the build system?



Some useful advice for starting out...



Interacting with headers - the global module fragment

```
export module A;  
  
import std;  
  
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```

Interacting with headers - the global module fragment

```
export module A;
```

```
import std;
```

```
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```

Interacting with headers - the global module fragment

```
module;  
  
#include <vector>  
  
export module A;  
  
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```

Interacting with headers - the global module fragment

```
module;
```

```
#include <vector>
```

```
export module A;
```

```
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```

Interacting with headers - the global module fragment

```
module;
```

```
#include <vector>
```

```
export module A;
```

```
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```


Interacting with headers - the global module fragment

```
module;
```

```
#include <vector>
```

```
export module A;
```

```
export std::vector<int> getVector() {  
    return std::vector<int>{ 1, 2, 3, 4 };  
}
```

Beware of mixing headers and modules

— *File:* my_module.cpp

```
export module m;
```

```
import std;
```

```
export std::string f() { return "!"; }
```

— *File:* main.cpp

```
import m;
```

```
int main() {  
    auto const s = f();  
}
```

Beware of mixing headers and modules

— *File:* my_module.cpp

```
export module m;
```

```
import std;
```

```
export std::string f() { return "!"; }
```

— *File:* main.cpp

```
import m;
```

```
int main() {  
    auto const s = f();  
}
```

Beware of mixing headers and modules

— *File:* my_module.cpp

```
export module m;
```

```
import std;
```

```
export std::string f() { return "!"; }
```

— *File:* main.cpp

```
import m;
```

```
int main() {  
    std::string const s = f();  
}
```

Beware of mixing headers and modules

— *File:* my_module.cpp

```
export module m;
```

```
import std;
```

```
export std::string f() { return "!"; }
```

— *File:* main.cpp

```
import m;
```

```
#include <string>
```

```
int main() {  
    std::string const s = f();  
}
```

When mixing, always put #includes first!

```
#include <string>

// Guideline: All #includes should
// come before the first import!
import std;

int main() {
    std::string const s = f();
}
```

Modularizing legacy libraries - Include standard library early

— *File: lib.hpp*

```
#include <string>
std::string f();
```

— *File: libm.cpp*

```
export module lib;
export {
#include <lib.hpp>
}
```

Modularizing legacy libraries - Include standard library early

— *File: lib.hpp*

```
#include <string>
std::string f();
```

— *File: libm.cpp*

```
module;
#include <string>

export module lib;
export {
#include <lib.hpp>
}
```


Modularizing legacy libraries - Export names with using

— *File: lib.hpp*

```
class AwesomeType;
```

— *File: libm.cpp*

```
module;  
#include <lib.hpp>  
export module lib;  
using ::AwesomeType;
```

Modularizing legacy libraries - Preprocessor macros

— *File: libm.cpp*

```
export module lib;  
#define AWESOME_MACRO 42
```

— *File: main.cpp*

```
import lib;  
  
int main() {  
    return AWESOME_MACRO;    // error! macros  
                             // cannot be exported  
}
```

Modularizing legacy libraries - Preprocessor macros

— *File: libm.cpp*

```
export module lib;  
// ...
```

— *File: libm.hpp*

```
#define AWESOME_MACRO 42
```

— *File: main.cpp*

```
#include <libm.hpp>  
import lib;  
  
int main() {  
    return AWESOME_MACRO;  
}
```

Modularizing legacy libraries - Preprocessor macros


— *File:* libm.cpp

```
export module lib;  
#define AWESOME_MACRO 42  
export constexpr int AwesomeConstant =  
    AWESOME_MACRO;
```

— *File:* main.cpp

```
import lib;  
  
int main() {  
    return AwesomeConstant;  
}
```

Daniela Engert - Contemporary C++ in Action



Daniela Engert

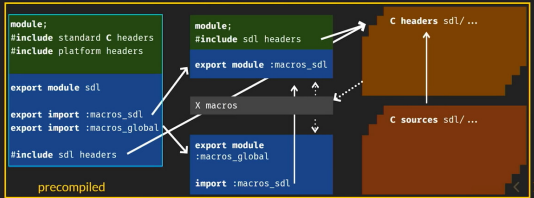
Contemporary C++ in Action

Cppcon 2022
The C++ Conference
September 12th-16th

MODULE SDL

A multi-part module compiled as a static library that contains

- a global module fragment with all headers that **must not be attached** to the module
- the module purview with all **exported interfaces**
- the **implementation** is compiled separately **with C semantics**



```
module;
#include standard C headers
#include platform headers

export module sdl

export import :macros_sdl
export import :macros_global
#include sdl headers
```

precompiled

module;
#include sdl headers
export module :macros_sdl

X macros

export module :macros_global
import :macros_sdl

C headers sdl/ ...

C sources sdl/ ...

Video Sponsorship Provided By:

ansatz think-cell

Conclusion

- Modules are slowly maturing. Try them out today!
- There is still a lot of dark corners in the implementations, but the more people use them, the quicker those get fixed.
- Integrating header-based legacy code is challenging and requires some practice.
- There is a lot of low-hanging fruit there for people interested in contributing to compilers and tooling

Conclusion

- Modules are slowly maturing. Try them out today!
- There is still a lot of dark corners in the implementations, but the more people use them, the quicker those get fixed.
- Integrating header-based legacy code is challenging and requires some practice.
- There is a lot of low-hanging fruit there for people interested in contributing to compilers and tooling

Where to go from here...

- Luis Caro Campos - C++20 Modules: The Packaging and Binary Redistribution Story (CppCon 2023, right afterwards in Cottonwood)
- Daniela Engert - Modules: The Beginner's Guide (Meeting C++ 2019)
- Daniela Engert - A Short Tour of C++ Modules (CppCon 2021)
- Bill Hoffman - import CMake: 2023 State of C++20 modules in CMake (CppNow 2023)
- Bret Brown - Modern CMake Modules (CppCon 2021)
- vector-of-bool - Understanding C++ Modules (Blog post series)
- Boeckel, King, Maynard, Hoffman - How CMake supports Fortran modules and its applicability to C++ (WG21 D1483)

Thanks for your attention.



Andreas Weis