C++ Modules - Getting Started Today

Andreas Weis

Meeting C++ 2024

Introduction

About me - Andreas Weis (he/him)

- ComicSansMS
- cpp@andreas-weis.net
- Co-organizer of the Munich C++ User Group

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;
struct NotExported { int i = 42; };
export NotExported getNotExported()
{ return {}; }
```

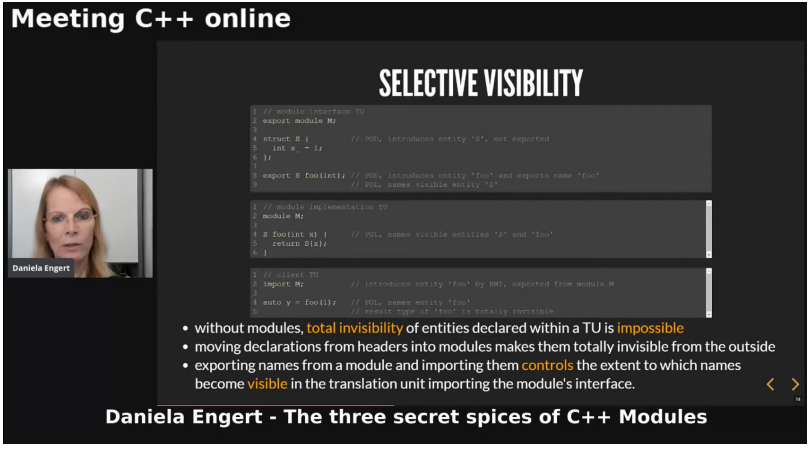
— *File:* `main.cpp`

```
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



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;
```

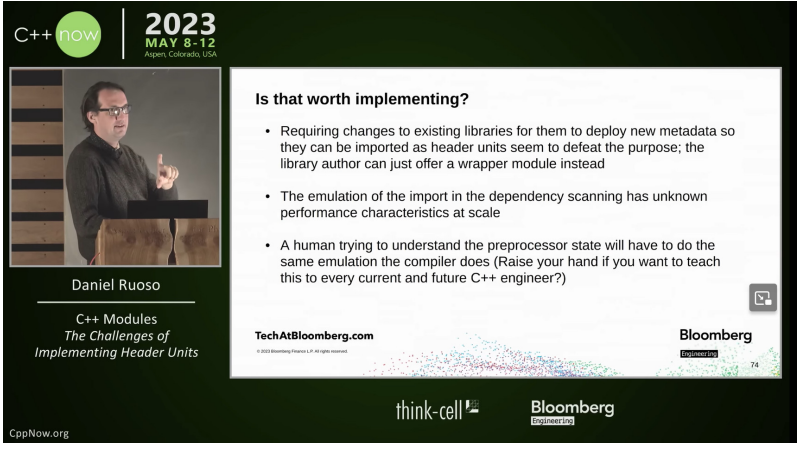
— *File:* my_module_p1.cpp

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

— *File:* my_module_p2.cpp

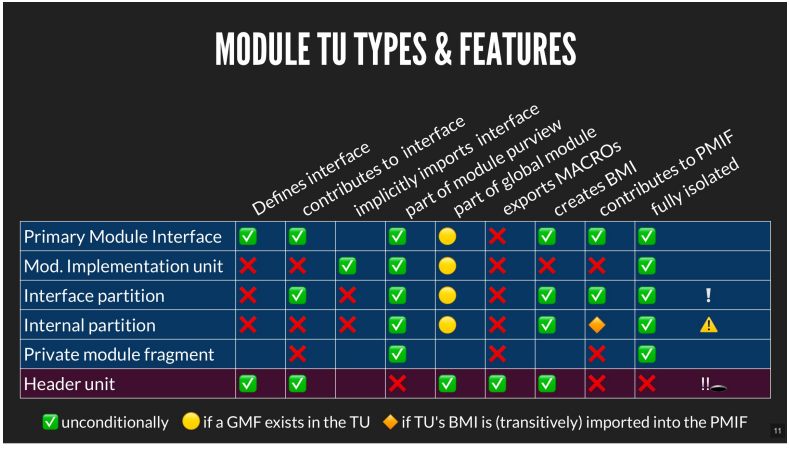
```
export module mice:the_brain;
export void take_over_the_world() {}
struct SecretMasterplan {};
```

We're not talking about Header Units



If you want to know more...

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



Building modules code with CMake

Building with CMake - Old School

```
cmake_minimum_required(VERSION 3.28)
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 3.23)
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 (since v3.28)

```
cmake_minimum_required(VERSION 3.28)
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 CXX_MODULES
  FILES
    ${PROJECT_SOURCE_DIR}/mod/my_module.cpp
)
```

Use the latest tools!

- Absolute latest CMake (at least 3.28, at least 3.30 for `import std`)
- Latest Visual Studio 2022 (at least toolset version 19.34)
- Ninja 1.11
- Clang at least 17, prefer 19
- gcc at least 14, consider 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?

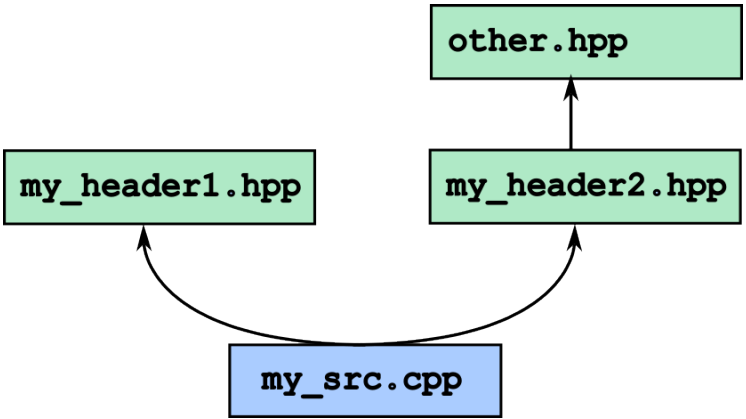
- Many different extensions started appearing in the compilers: `.ixx`, `.cppm`, `.cxxm`, `.c++m`, `.ccm`.
- With CMake you don't need to use any of them!
- If you decide to use them, be sure to *only* use them for module source files (as defined above)

What is a build system anyway?

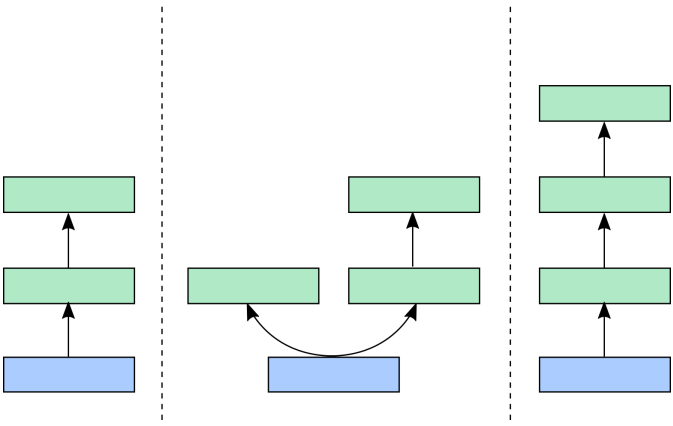
```
$ g++ -o my_src.o -I . -c my_src.cpp

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

Tracking dependencies



Tracking dependencies



Old-school: Manual tracking

Makefile

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

my_executable: main.o my_src.o
  $(CXX) -o my_executable main.o my_src.o
```

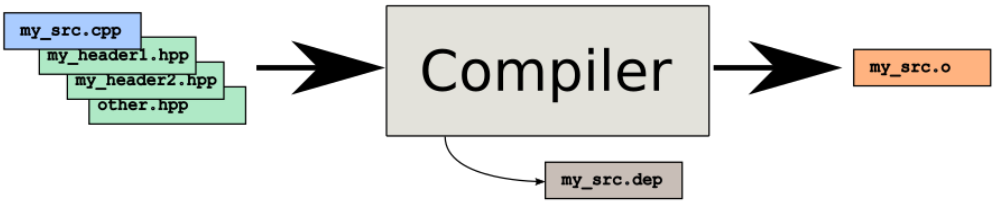
Tracking changes to header files

```
$ 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
```

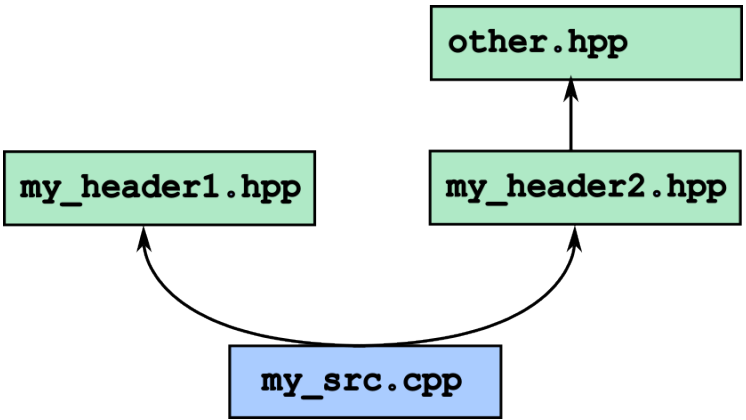
Tracking changes to header files



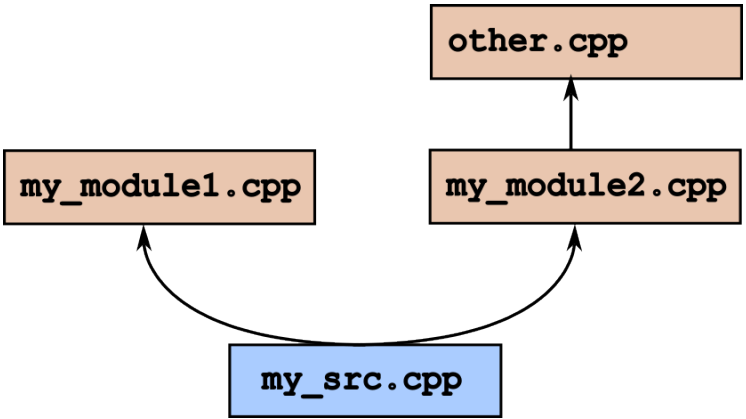
— *File:* `my_src.dep`

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

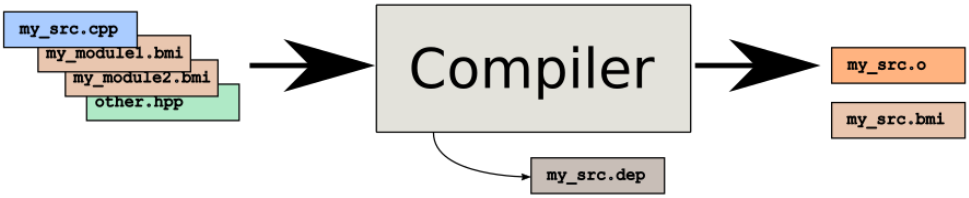
What changes for the build system?



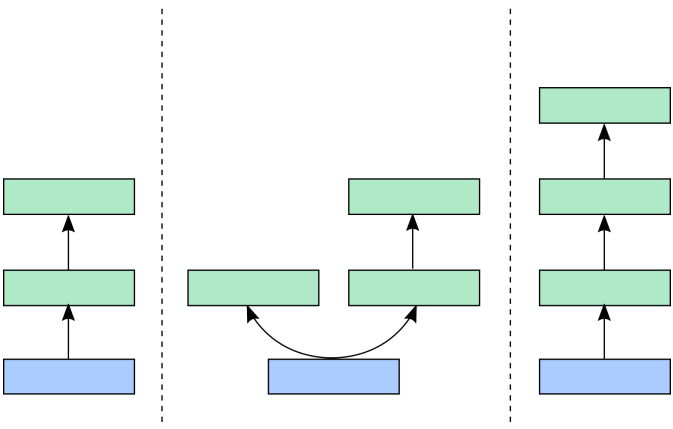
What changes for the build system?



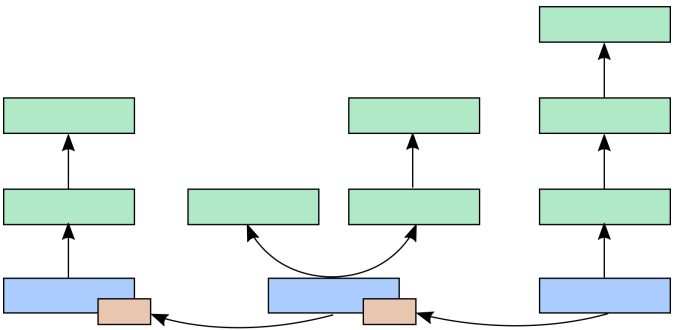
What changes for the build system?



What changes for the build system?



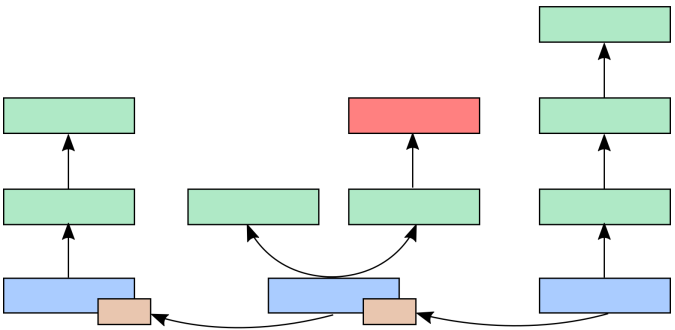
What changes for the build system?



What changes for the build system?

```
add_executable(my_executable)
target_sources(my_executable PUBLIC
    ${PROJECT_SOURCE_DIR}/my_src.cpp
  PUBLIC
  FILE_SET CXX_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();
}
```

Names exported from a module become attached to the module

```
int the_answer() {
  return 42;
}
```

```
$ nm my_source.o
0000000000000000 T _Z10the_answerv
$ c++filt _Z10the_answerv
the_answer()
```

Names exported from a module become attached to the module

```
namespace my_namespace {
  int the_answer() {
    return 42;
  }
}
```

```
$ nm my_source.o
0000000000000000 T _ZN12my_namespace10the_answerEv
$ c++filt _ZN12my_namespace10the_answerEv
my_namespace::the_answer()
```

Names exported from a module become attached to the module

```
export module my_module;
int the_answer() {
  return 42;
}
```

```
$ nm my_source.o
0000000000000000 T _ZGIW9my_module
0000000000000000 T _ZW9my_module10the_answerv
$ c++filt _ZGIW9my_module
initializer nonefor module my_module
$ c++filt _ZW9my_module10the_answerv
the_answer@my_module()
```

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;
export using ::AwesomeType;
```

Modularizing legacy libraries - Detach from module

```
module;
// Global module fragment - can't export from here
void f1() {}

export module lib;
// Model purview - declaring here attaches to module
export void f2() {}


// Use extern to detach declaration
export extern "C++" void f3() {}
```

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`

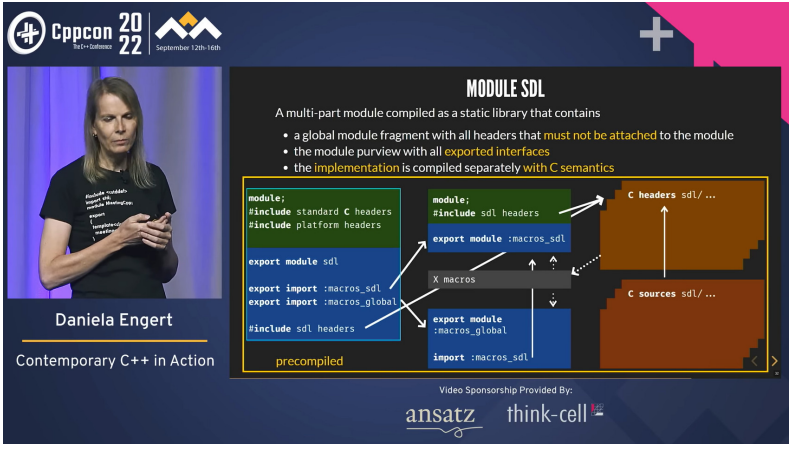
```
export module lib;
#define AWESOME_MACRO 42
export constexpr int AwesomeConstant =
    AWESOME_MACRO;
```

— *File:* `main.cpp`

```
import lib;

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

Daniela Engert - Contemporary C++ in Action



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...

- Daniela Engert - Modules: The Beginner's Guide (Meeting C++ 2019)
- Daniela Engert - A Short Tour of C++ Modules (CppCon 2021)
- Bill Hoffman - import CMake: Mastering C++ Modules (CppCon 2024)
- Bret Brown - Modern CMake Modules (CppCon 2021)
- vector-of-bool - Understanding C++ Modules (Blog post series)
- Modules on cppreference
- Boeckel, King, Maynard, Hoffman - How CMake supports Fortran modules and its applicability to C++ (WG21 D1483)

Thanks for your attention.

Modules *Hello World* example on Compiler Explorer:
https://godbolt.org/z/Ea54d1Ta3



ComicSansMS /

cpp@andreas-weis.net