C++20 Modules:
The Packaging and Binary Redistribution Story

LUIS CARO CAMPOS

Cppcon
The C++ Conference

2023
October 01 - 06

C++20 Modules: The Packaging and Binary Redistribution Story

Luis Caro Campos
SW Tech Lead, JFrog

Scope

- Brief introduction on C++ modules and their advantages
- Focus on **named modules**
- Using modules today
  - Can we package module-ready libraries and use them in our projects?

The include directive

```
#include <fmt/core.h>

int main() {

  fmt::print("Hello, world!\n");

}
```

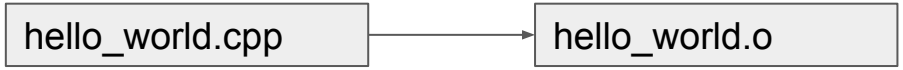
hello_world.cpp

- Include directive: “Tells the **preprocessor** to include the contents of a specified file at the point where the directive appears.”

Around ~350k lines after the include is replaced by the contents

```
hello_world.cpp:2:3: error: use of undeclared identifier 'fmt'
  fmt::print("Hello, world!\n");
  ^
1 error generated.
```

The include directive - cont'd



When invoking the compiler, it needs to be able to resolve the location of `#included` files

Typically:

- Some default locations:
  - Relative to the .cpp file (for “”)
  - System or compiler installation locations (e.g. `/usr/include`)
- **`-I` flags** pointing to other locations:
  - Typically handled by build system / package manager

The include directive - cont'd

```
clang++ -std=c++20  -o hello_world.cpp.o -c hello_world.cpp
hello_world.cpp:1:10: fatal error: 'fmt/core.h' file not found
#include <fmt/core.h>
         ^~~~~~~~~~~~
1 error generated.
```

```
clang++ -std=c++20  -o hello_world.cpp.o -c hello_world.cpp
-I/path/to/fmt/include
```

These days this is handled by build system abstractions and is hidden away from developers

```
target_link_libraries(hello_world PRIVATE fmt::fmt)
```

A typical library package

```
apt-get install libfmt-dev
```

```
conan install --require=fmt/10.1.0
```

```
vcpkg install fmt
```

```
├── include
│   └── fmt
│       ├── args.h
│       ├── chrono.h
│       ├── color.h
│       ├── compile.h
│       ├── core.h
│       ├── format-inl.h
│       ├── format.h
│       ├── os.h
│       ├── ostream.h
│       ├── printf.h
│       ├── ranges.h
│       ├── std.h
│       └── xchar.h
├── lib
    |-- cmake
    |   `-- fmt
    |       |-- fmt-config.cmake
    |       `-- fmt-targets.cmake
    └── libfmt.{a|so|dylib|lib}
```

“**fmt**” package

A typical library package

Headers

“Usage requirements”

Binary library

```
├── include
│   └── fmt
│       ├── args.h
│       ├── chrono.h
│       ├── color.h
│       ├── compile.h
│       ├── core.h
│       ├── format-inl.h
│       ├── format.h
│       ├── os.h
│       ├── ostream.h
│       ├── printf.h
│       ├── ranges.h
│       ├── std.h
│       └── xchar.h
├── lib
    |-- cmake
    |   `-- fmt
    |       |-- fmt-config.cmake
    |       `-- fmt-targets.cmake
    └── libfmt.{a|so|dylib|lib}
```

“**fmt**” package

C++ 20: The import keyword

```
#include <fmt/core.h>

int main() {
  fmt::print("Hello, world!\n");
}
```

```
import fmt;

int main() {
  fmt::print("Hello, world!\n");
}
```

```
clang++ -std=c++20 -o hello_world.cpp.o -c hello_world.cpp
hello_world.cpp:1:8: fatal error: module 'fmt' not found
import fmt;
~~~~~~~^~~
1 error generated.
```

Why use modules

In short: **better isolation** - some advantages to headers:

- Importers cannot affect the contents of the module being imported
- Imported modules cannot affect the state of the preprocessor in the importing code
- Ordering of imports does not matter
- Potential for improved build times (more on this later)

Why use modules (cont'd)

MSVC Compilation Funtimes

PUBLISHED BY WIREPAIR ON AUGUST 2, 2023

```
#define NOMINMAX 🔥
```

```
#include <windows.h>
#include <flatbuffers/flatbuffers.h>
```

wirepair.org

```
C:\path\to\include\fb/verifier.h(38,23): warning C4003: not enough arguments for function-like macro
invocation 'max'
C:\path\to\include\fb/verifier.h(38,23): error C2589: '(': illegal token on right side of '::'
C:\path\to\include\fb/verifier.h(38,12): error C2062: type 'unknown-type' unexpected
C:\path\to\include\fb/verifier.h(38,12): error C2059: syntax error: ')'
```

C++ 20: The import keyword

```
import fmt;

int main() {
  fmt::print("Hello, world!\n");
}
```

```
clang++ -std=c++20  -o hello_world.cpp.o -c hello_world.cpp
hello_world.cpp:1:8: fatal error: module 'fmt' not found
import fmt;
~~~~~~~^~~
1 error generated.
```

```
clang++ -std=c++20 -o hello_world.cpp.o -c hello_world.cpp -fmodule-file=fmt=/path/to/fmt.pcm
```

Resolving imports

```
#include <fmt/core.h>

int main() {
 fmt::print("Hello!\n");
}
```

```
clang++ -std=c++20  -o hello_world.cpp.o -c hello_world.cpp
-I/path/to/fmt/include
```

```
import fmt;

int main() {
 fmt::print("Hello!\n");
}
```

The compiler needs to locate and load the **binary module interface** (BMI) 👇:

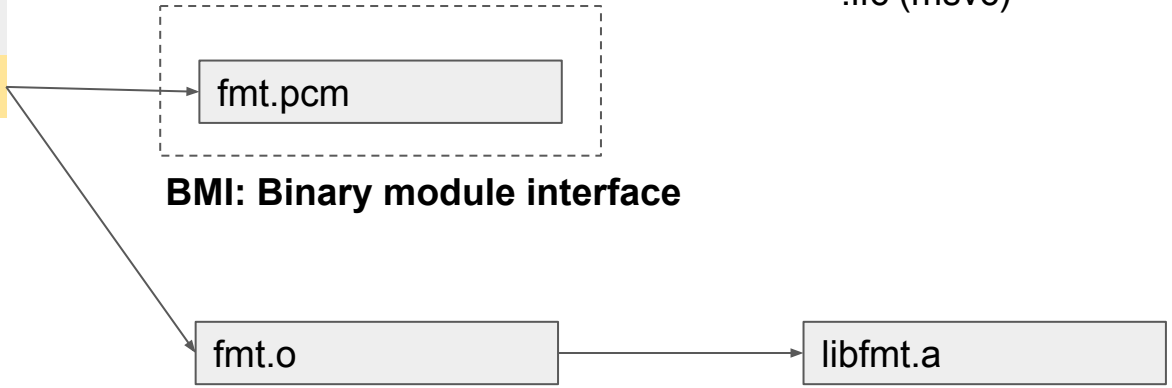
compiler	CLI
msvc	`/reference fmt=/path/to/fmt.ifc`
clang	`-fprebuilt-module-path=/path/to/folder` `-fmodule-file=fmt=/path/to/fmt.pcm`
gcc	`-fmodule-mapper=` `(server or file with module name <> file mappings)`

Binary module interfaces

```
export module fmt;

export namespace fmt {
 …
}
```

fmt.cc

BMI file extensions:
.pcm (clang)
.gcm (gcc)
.ifc (msvc)



Module interface unit

Other file extensions:
.ixx (msvc)
.cppm (clang)

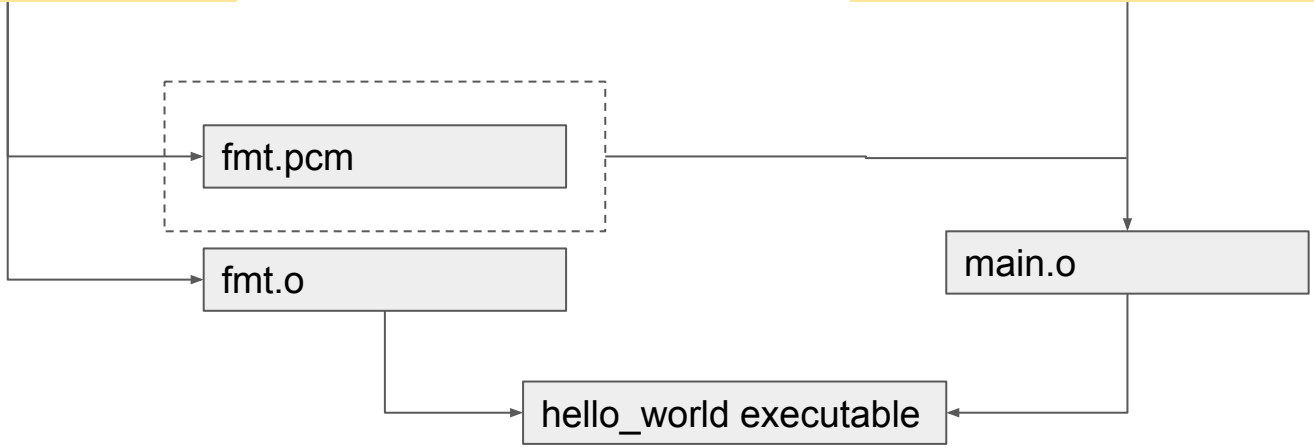
Module interface unit

```
export module fmt;

export namespace fmt {
  …
}
```

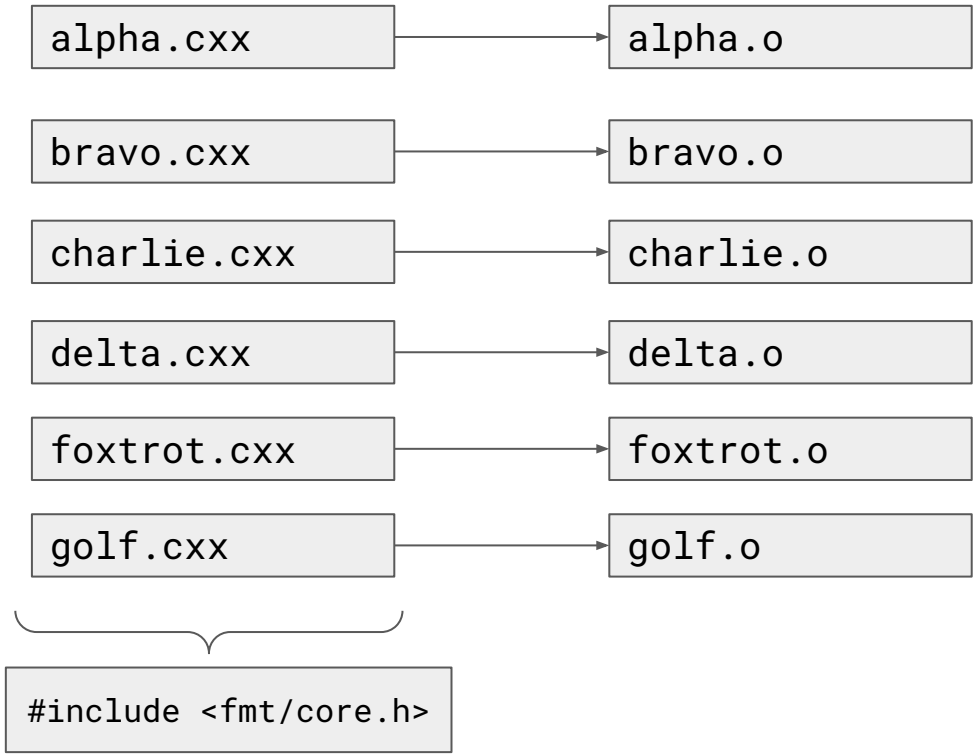
fmt.cc

```
import fmt;

int main() {
  fmt::print("Hello, world!\n");
}
```

main.cc



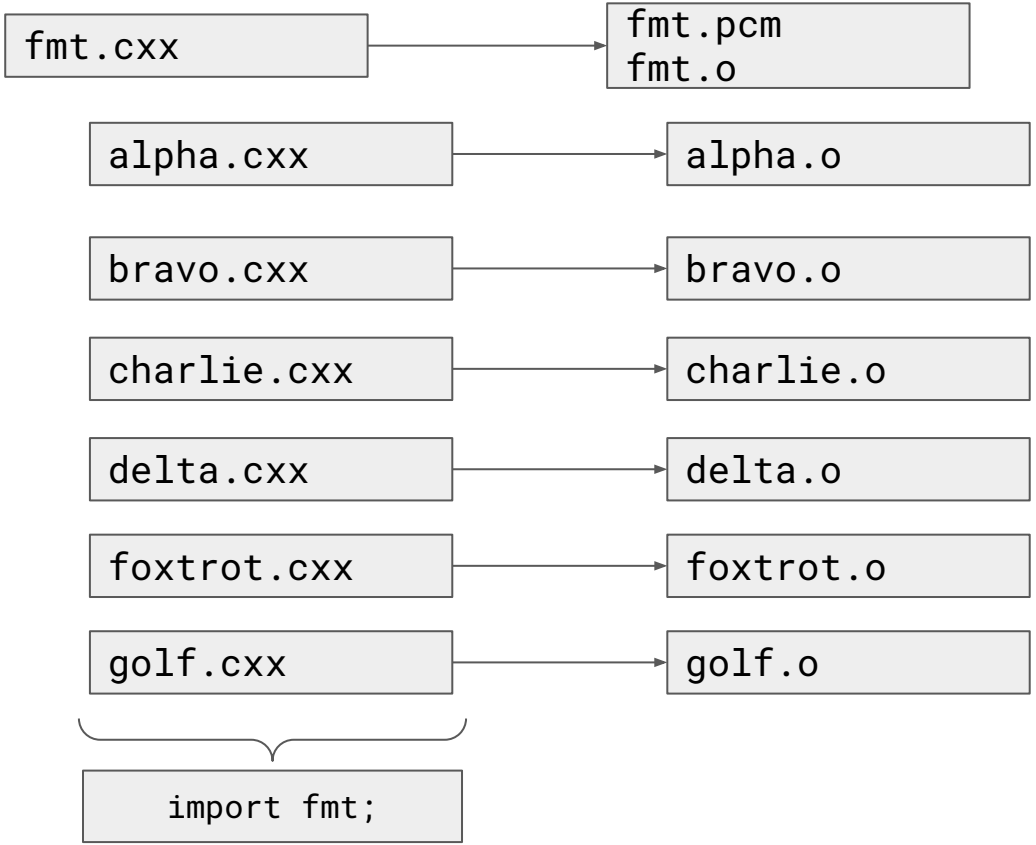
Build order - using #include



Compiler invocations:
Embarrassingly parallel

If many `.cxx` files in the project include the same headers -
Each compiler invocation is parsing them independently

Build order - using import



Module interface units need to be compiled before any of the importers

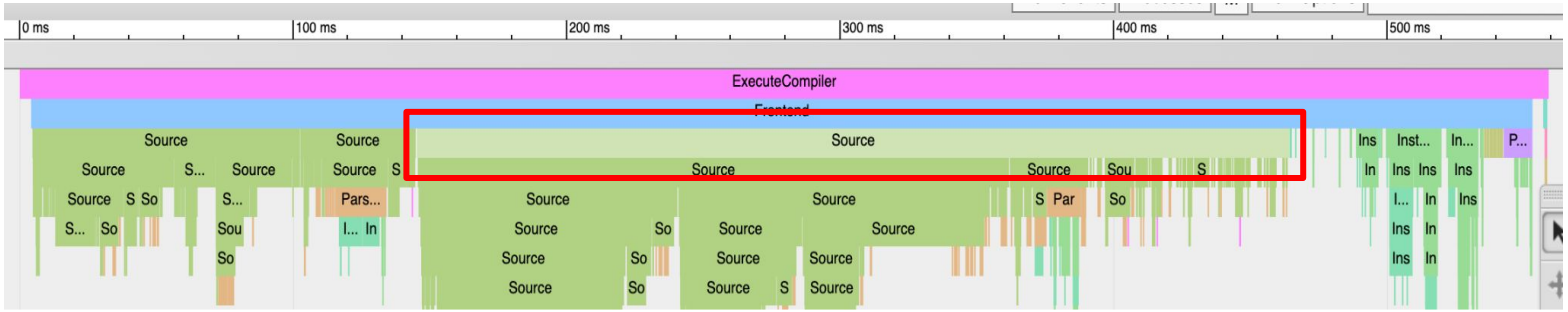
Compilation time

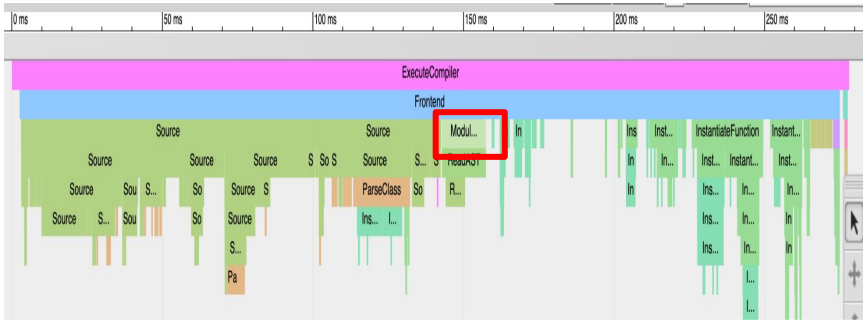
tristanbrindle.com

About Posts Projects

Experimenting with Modules in Flux

Aug 3, 2023 • Tristan Brindle

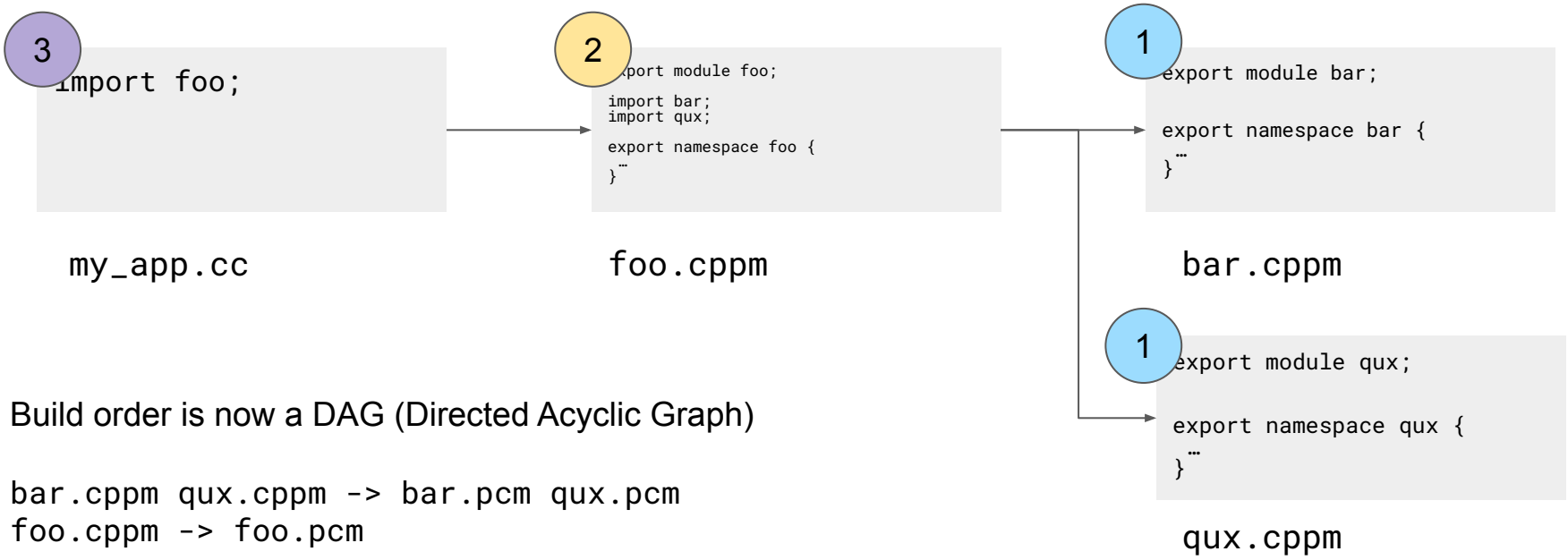





```
#include <flux.hpp>
```
took 319 ms

```
import flux;
```
took 14 ms

Your imports (can) have imports



Build order is now a DAG (Directed Acyclic Graph)

```
bar.cppm qux.cppm -> bar.pcm qux.pcm
foo.cppm -> foo.pcm
my_app.cc -> my_app
```

Include vs import summary

- Using **`#include`** means compiling several files in a project can be easily parallelized
  - But if the same header files are included in many translation units, the compiler does the same job repeatedly
  - Downsides w.r.t. the preprocessor
- Using **`import`** introduces a dependency order between translation units
  - The BMI for module interface units must be generated before any importer is translated - have to work out build order
  - But better isolation between modules and importers
  - Potential for improved compilation times

Deriving the correct build order

- In build scripts (CMake, MSbuild, etc) we are used to expressing dependencies between targets (libraries), but not between individual *files*
  - Exceptions: e.g. generated code like protobuf, Qt moc etc
- We could express the right order in Makefiles, Ninjafiles etc - but we don’t do those manually
- Imports *may* change over time, we would have to mirror the .cxx dependency graph in our build scripts, and it may become “out of sync” -> not ideal

Dependency scanning

Format for describing dependencies of source files

Ben Boeckel, Brad King
<ben.boeckel@kitware.com, brad.king@kitware.com>
version P1689R5, 2022-06-03

- Before compiling - roughly extract information:
  - Name of exported module (if any)
  - Which imports are required (if any)
- This can be used to calculate the right build order:
  - We know which sources produce which files
- New constraints:
  - No circular dependencies between imports (no cycles in the graph)
  - Build tool should be able to re-order dependencies based on the scanning results

Dependency scanning

kitware

about solutions news careers contact

news › blog › post

import CMake; C++20 Modules

January 19, 2023 Bill Hoffman, Ben Boeckel and Brad King

Work is underway to implement support for C++20 modules in CMake! Since the C++ standards committee started talking about adding modules to the C++ language, the CMake team at Kitware has been thinking about how they will be supported. Fortunately, CMake has supported Fortran modules since 2005. In 2015, support was added to the ninja build tool to support Fortran modules. In 2019 with news of modules being added to C++, the Kitware fork of ninja was rejoined with upstream ninja and dynamic dependencies were added to ninja. This blog describes the process that was taken and the current state of named C++ 20 modules in CMake. Header modules are not covered in this blog.

https://www.kitware.com/import-cmake-c20-modules/

Compilers (minimum versions):

- LLVM Clang 16
- Visual Studio 17.4 (msvc 19.34)
- gcc 14 (to be released)

CMake:

- Version 3.25
- Version 3.28 for gcc14 support (to be released)

Build tools:

- Ninja 1.10
- MSBuild

Dependency scanning - not the only approach

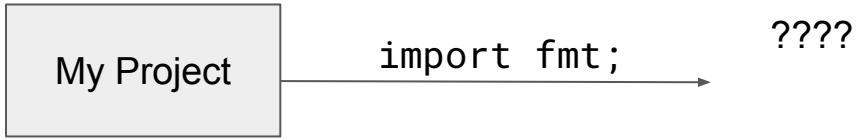
- Build2 module support (2 years ago) predates dependency scanning approach
- It follows a fuzzy search
- Described here: https://build2.org/build2/doc/build2-build-system-manual.xhtml#cxx-modules
- Dependency scanning is robust and technically hands-off:
  - If dependency scanning is enabled for everything (like MSBuild), then:
    - No need for any specific file extension for module interface units (any will do)
    - Compiler can work out the name of the exported module, is robust to macros that control the export
    - No need to match the exported module name with the filename at all - still recommended, but could be anything

Dependency scanning works really well!

Compiler, CMake and build tools all working together - this is a win for the C++ ecosystem!

Works well, **with an important constraint**:

- For everything that you **import**, the sources (module interface units) must be visible by the build system



Our simple project

```
cmake_minimum_required(VERSION 3.27)

project(hello-fmt LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

find_package(fmt REQUIRED)

add_executable(hello hello_world.cpp)
target_link_libraries(hello PRIVATE fmt::fmt)
```

CMakeLists.txt

```
import fmt;

int main() {
  fmt::print("Hello, world!\n");
}
```

hello_world.cpp

{fmt} module library

```
├── include
│   └── fmt
│       ├── args.h
│       ├── chrono.h
│       ├── color.h
│       ├── compile.h
│       ├── core.h
│       ├── format-inl.h
│       ├── format.h
│       ├── os.h
│       ├── ostream.h
│       ├── printf.h
│       ├── ranges.h
│       ├── std.h
│       └── xchar.h
├── lib
    |-- cmake
    |   `-- fmt
    |       |-- fmt-config.cmake
    |       `-- fmt-targets.cmake
    └── libfmt.{a|so|dylib|lib}
```

?

“**fmt**” package

“**fmt**” packaged module library

{fmt} library

For experimentation: **fork** of `{fmt}` using the new CMake module capabilities

```
add_library(fmt)
target_sources(fmt
  PUBLIC
    FILE_SET fmt_module TYPE CXX_MODULES FILES src/fmt.cc
)

target_compile_features(fmt PUBLIC cxx_std_20)

add_library(fmt::fmt ALIAS fmt)
```

github.com/jcar87/fmt

lcc/experimental/v10.1.1-cmake-modules

Packaging libraries today

Adopt sources in our project

No packaging

```
├── include/fmt
│       ├── core.h
├── lib
│   ├── libfmt.a
│   └── cmake
```

Static library

```
├── include/fmt
│       ├── core.h
├── lib
│   ├── libfmt.so
│   └── cmake
```

Shared library

```
├── include/fmt
│       ├── core.h
├── lib
│   └── cmake
```

Header only

Packaging module libraries - how?

```
Adopt sources in
our project
```

No packaging

```
├── lib
│   ├── libfmt.a
│   └── cmake
│   └── cxx/bmi
│       ├── fmt.bmi
```

BMI + binary library (shared or static)

```
├── lib
│   ├── libfmt.so
│   └── cxx/miu
│       ├── fmt.cxx
```

Module interfaces + binary library (shared or static)

```
├── lib
│   └── cxx/miu
│       ├── fmt.cxx
```

Module only

No “packaging” - build external sources in our project

```
include(FetchContent)
FetchContent_Declare(fmt
  GIT_REPOSITORY https://github.com/jcar87/fmt.git
  GIT_TAG 28fbcaa72b6224f7824672a39f80c6130e28f317 # 10.1.1 built as module
  OVERRIDE_FIND_PACKAGE
)

FetchContent_MakeAvailable(fmt)

find_package(fmt REQUIRED)
```

Packaging module libraries

```
No package
(include sources)
```

No packaging

```
├── lib
│   ├── libfmt.a
│   └── cmake
│   └── cxx/bmi
│       ├── fmt.bmi
```

BMI + binary library (shared or static)

```
├── lib
│   ├── libfmt.so
│   └── cxx/miu
│       ├── fmt.cxx
```

Module interfaces + binary library (shared or static)

```
├── lib
│   └── cxx/miu
│       ├── fmt.cxx
```

Module only

Packaging the BMIs

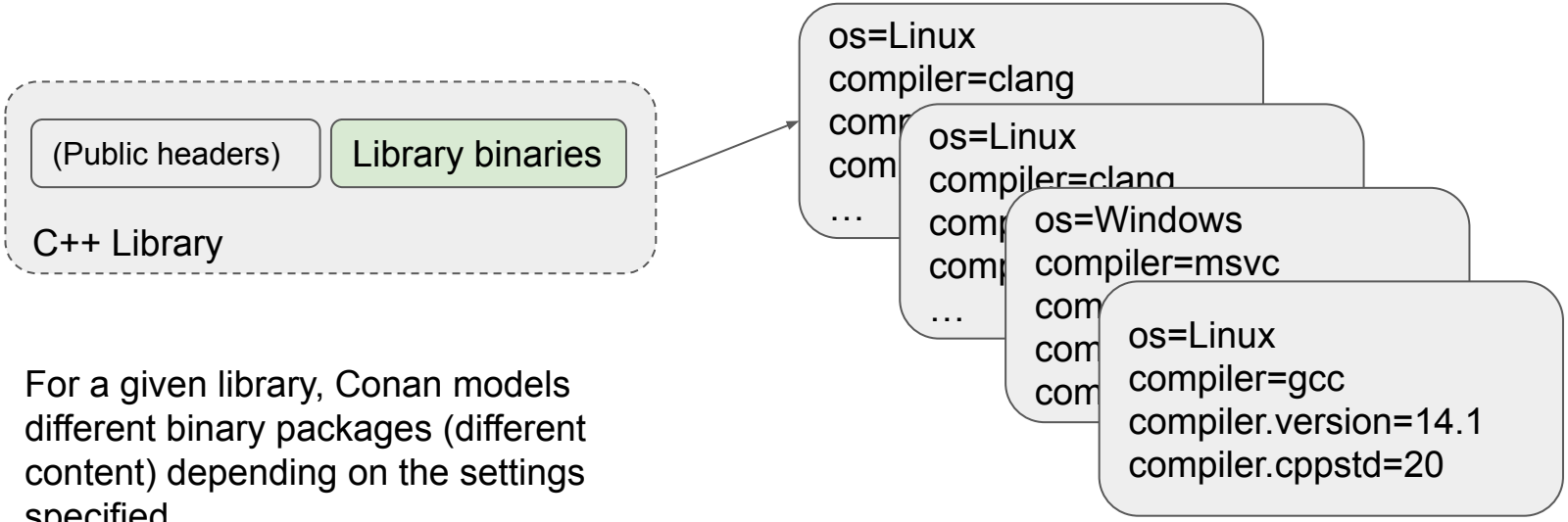
- BMIs are not compatible across compilers (different file formats altogether)
  - Or even across different versions of the same compiler

3.23.3 Compiled Module Interface

CMIs are an additional artifact when compiling named module interfaces, partitions or header units. These are read when importing. CMI contents are implementation-specific, and in GCC's case tied to the compiler version. Consider them a rebuildable cache artifact, not a distributable object.

☝️ GCC documentation

Packaging the BMIs - cont'd



For a given library, Conan models different binary packages (different content) depending on the settings specified.

If strict compiler version is required and developers can enforce it, this could be suitable.

Packaging the BMIs - cont'd

```
|-- include
|   `-- fmt
|       |-- core.h
|       `-- ...
`-- lib
    |-- cmake
    |   `-- fmt
    |       |-- fmt-config.cmake
    |       `-- fmt-targets.cmake
    |-- cxx
    |   |-- bmi
    |       `-- fmt.pcm
    `-- libfmt.a
```

- Package the BMI (for msvc)
- For msvc, consumers needs to pass:
  - `/reference fmt=/path/to/fmt.ifc`
- We already have abstractions for this:
  - INTERFACE_COMPILE_FLAGS (CMake)
  - CFlags (pkg-config)
  - Typically for include directories
  - This is … similar, right?

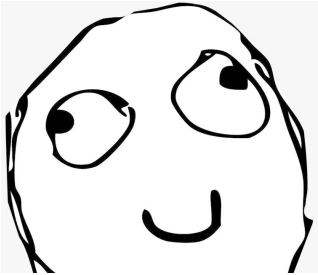
```
set_target_properties(fmt::fmt PROPERTIES
  INTERFACE_COMPILE_FEATURES "cxx_std_20;cxx_std_11"
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
  INTERFACE_COMPILE_OPTIONS "$<$<CXX_COMPILER_ID:MSVC>:/reference;fmt=${_IMPORT_PREFIX}/lib/cxx/miu/bmi/fmt.ifc>"
)
```

Packaging the BMIs - cont'd

```
cmake .. -GNinja -DCMAKE_BUILD_TYPE=Release --fresh
-- The CXX compiler identification is MSVC 19.37.32822.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files/Microsoft Visual
Studio/2022/Community/VC/Tools/MSVC/14.37.32822/bin/Hostx64/arm64/cl.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
…

-- Configuring done (1.6s)
-- Generating done (0.0s)
-- Build files have been written to: C:/Users/luisc/dev/conan-io/cxx-modules-experimental/fmt-tests/consumer-test/build
(venv) PS C:\Users\luisc\dev\conan-io\cxx-modules-experimental\fmt-tests\consumer-test\build> ninja
[4/4] Linking CXX executable hello.exe

(venv) PS C:\Users\luisc\dev\conan-io\cxx-modules-experimental\fmt-tests\consumer-test\build> ./hello.exe
Hello, World! From fmt module
```

Packaging the BMIs - cont’d

```
def package_info(self):
      self.cpp_info.libs = ["fmt"]
      self.cpp_info.includedirs = []

      if self.settings.compiler == “msvc”:
        bmi_dir = os.path.join(self.package_folder, "bmi").replace('\\','/')
        self.cpp_info.cxxflags = ["/reference fmt=fmt.cc.ifc", f"/ifcSearchDir{bmi_dir}"]
      elif self.settings.compiler == "clang":
        self.cpp_info.cxxflags = [f"-fmodule-file=fmt={self.package_folder}/bmi/fmt.pcm"]
```



This could work!

*except for gcc

Packaging the BMIs - cont’d

```
FAILED: CMakeFiles/example.dir/example.cpp.o

/lib/llvm-16/bin/clang++   -stdlib=libstdc++ -O3 -DNDEBUG -std=c++20
-fmodule-file=fmt=/root/.conan2/p/b/fmtdba5a5f9dc355/p/bmi/fmt.pcm -MD -MT CMakeFiles/example.dir/example.cpp.o -MF
CMakeFiles/example.dir/example.cpp.o.d -o CMakeFiles/example.dir/example.cpp.o -c
/cxx-modules/conan-recipes/fmt/test_package/example.cpp

error: default visibility for functions and variables [-fvisibility] differs in PCH file vs. current file

error: module file /root/.conan2/p/b/fmtdba5a5f9dc355/p/bmi/fmt.pcm cannot be loaded due to a configuration mismatch with the
current compilation [-Wmodule-file-config-mismatch]

/cxx-modules/conan-recipes/fmt/test_package/example.cpp:4:3: error: use of undeclared identifier 'fmt'

  fmt::print("Hello, world!\n");

  ^

3 errors generated.

ninja: build stopped: subcommand failed.
```

Packaging BMIs - cont'd

Consistency Requirement

If we envision modules as a cache to speed up compilation, then - as with other caching techniques - it is important to keep cache consistency. So **currently** Clang will do very strict check for consistency.

New dimension of complexity: **BMI compatibility**

We may be binary/ABI compatible with the symbols in the library (.a/.so)

But our sources are not necessarily BMI compatible …

Different compilers may have different rules - e.g. `-fvisibility=hidden' was not an issue with gcc

Packaging BMIs - cont'd

```
FAILED: CMakeFiles/example.dir/example.cpp.o

/lib/llvm-16/bin/clang++   -stdlib=libstdc++ -O3 -DNDEBUG -std=c++20
-fmodule-file=fmt=/root/.conan2/p/fmt7d4cc28edd661/p/bmi/fmt.pcm -MD -MT CMakeFiles/example.dir/example.cpp.o
-MF CMakeFiles/example.dir/example.cpp.o.d @CMakeFiles/example.dir/example.cpp.o.modmap -o
CMakeFiles/example.dir/example.cpp.o -c /cxx-modules/conan-recipes/fmt/test_package/example.cpp

/cxx-modules/conan-recipes/fmt/test_package/example.cpp:1:1: fatal error: malformed or corrupted AST file:
'could not find file '/root/foobarfoobar/p/b/fmt4ff89414aff58/b/src/fmt.cc' referenced by AST file
'/root/.conan2/p/fmt7d4cc28edd661/p/bmi/fmt.pcm''

import fmt;

^

1 error generated.

ninja: build stopped: subcommand failed.
```

Packaging BMIs - across computers (cont'd)

With **msvc** this actually works

But if there are **compiler errors**, non existent files might be referenced. This could make it harder for developers to troubleshoot errors, and for tooling if the actual files cannot be located

```
[3/4] Building CXX object CMakeFiles\example.dir\example.cpp.obj

FAILED: CMakeFiles/example.dir/example.cpp.obj

C:\Users\luisc\dev\conan-io\cxx-modules-experimental\conan-recipes\fmt\test_package\example.cpp(4): error C2665: 'fmt::v10::print': no overloaded
function could convert all the argument types

C:\Users\luisc\dev\conan-io\cxx-modules-experimental\conan-home\p\b\fmtd161dd09a5aa7\b\include\fmt\xchar.h(236): note: could be 'void
fmt::v10::print<>(fmt::v10::basic_format_string<wchar_t>)'
```

Packaging BMIs - conclusions

- Not advised by compiler maintainers
  - BMIs should all be generated by the project we are building - even for externally provided dependencies
- We would need very strict control:
  - Compiler, compiler version, compiler flags, same paths everywhere
    - Guaranteed across all dependencies
    - Varies per compiler

Packaging module libraries - how?

```
No package
(include sources)
```

No packaging

```
├── lib
│   ├── libfmt.a
│   └── cmake
│   └── cxx/bmi
│       ├── fmt.bmi
```

BMI + binary library (shared or static)

```
├── lib
│   ├── libfmt.so
│   └── cxx/miu
│       ├── fmt.cc
```

Module interfaces + binary library (shared or static)

```
├── lib
│   └── cxx/miu
│       ├── fmt.cc
```

Module only

Packaging the module interfaces

```
|-- include
|   `-- fmt
|       |-- <otherfiles>.h
|       `-- core.h
`-- lib
    |-- cmake
    |   `-- fmt
    |       `-- fmt-config.cmake
    |-- cxx
    |   `-- miu
    |       `-- fmt.cc
    `-- libfmt.a
```

- We still package a binary library
- But we also package the module interface units:
  - The file that does `export module`, plus all re-exported module interface partitions
- This time around we can't rely on existing abstractions
  - target_sources(xx PUBLIC) in CMake would be the closest
- The build system doing the build has to translate all the modules unit used across the project, whether the libraries are provided externally or not
  - **<u>This is a new capability</u>**

Packaging the module interfaces - cont'd

Build system support

- CMake 3.28 (yet to be released)

cxxmodules: support modules on IMPORTED targets

Merged **Ben Boeckel** requested to merge `ben.boeckel/cmake:import…` into `master` 3 months ago

Merged a month ago!

- Build2 - special field in .pkgconfig files

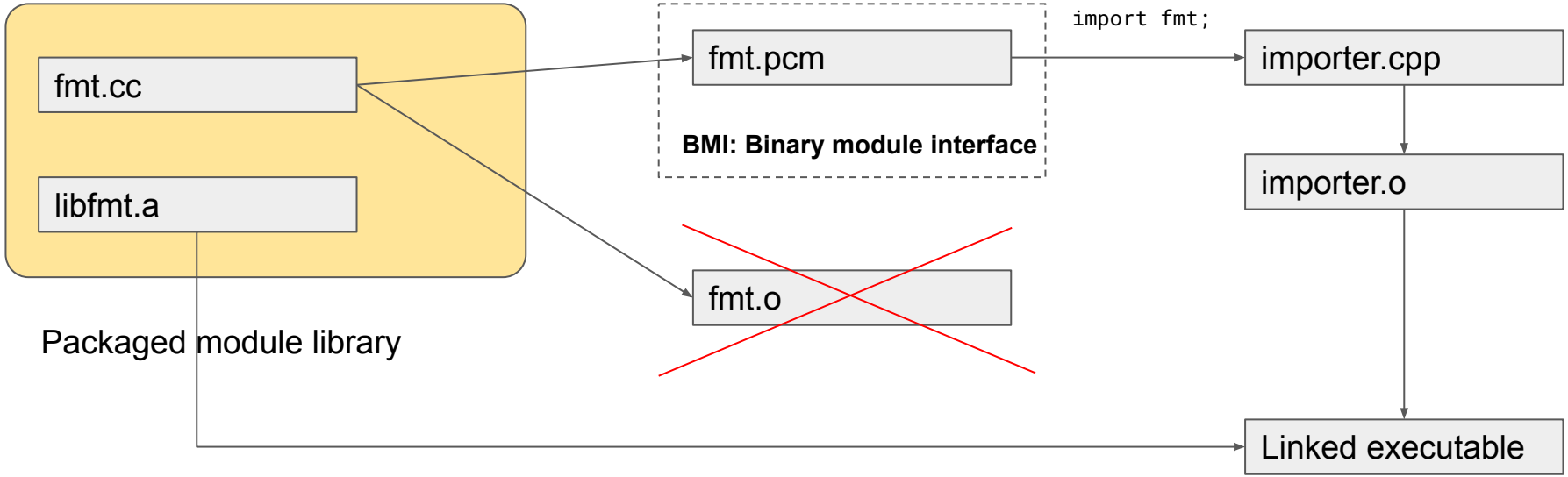
Packaging the module interfaces - cont'd

CMake 3.28 (to be released)

```
# Create imported target fmt::fmt
add_library(fmt::fmt SHARED IMPORTED)

set_target_properties(fmt::fmt PROPERTIES
  CXX_EXTENSIONS "OFF"
  IMPORTED_CXX_MODULES_COMPILE_DEFINITIONS "FMT_LIB_EXPORT"
  IMPORTED_CXX_MODULES_COMPILE_FEATURES "cxx_std_20;cxx_std_11"
  IMPORTED_CXX_MODULES_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
  INTERFACE_COMPILE_DEFINITIONS "FMT_SHARED"
  INTERFACE_COMPILE_FEATURES "cxx_std_20;cxx_std_11"
  INTERFACE_INCLUDE_DIRECTORIES "${_IMPORT_PREFIX}/include"
)
target_sources(fmt::fmt
  INTERFACE
  FILE_SET "fmt_module"
  TYPE "CXX_MODULES"
  BASE_DIRS "${_IMPORT_PREFIX}/lib/cxx/miu"
  FILES "${_IMPORT_PREFIX}/lib/cxx/miu/src/fmt.cc"
)
```

- The external module interface file (fmt.cc) is taken into account for dependency scanning
- From this, the build system can derive:
  - Names of exported modules
  - Correct build order - generate the BMI before the files that import it

Packaging the module interfaces (cont'd)



Tell compiler to generate BMI only:

- gcc: `-fmodule-only`
- Clang: `--precompile`
- msvc: `/ifcOnly`

Packaging the module interface (cont'd)

- All BMIs generated locally, <u>increasing</u> compatibility
  - Same compiler and compiler version as the translation units doing the importing
  - Module interface units are visible and exist on the local filesystem
- "Usage requirements" need to be expanded:
  - The list of module interface units would suffice, but …
  - We also need compiler flags that consumers need to pass to produce a compatible BMI
    - Include directories (if needed)
    - Macro definitions, compiler options
  - There is no standard way of describing a package … or a library!
    - P2577R2: C++ Module Discovery in Prebuilt Library Releases
    - P2701R0: Translating Linker Input Files to Module Metadata Files

Packaging the module interface (cont'd)

```
|-- include
|   `-- fmt
|       |-- <otherfiles>.h
|       `-- core.h
`-- lib
    |-- cmake
    |   `-- fmt
    |       `-- fmt-config.cmake
    |-- cxx
    |   `-- miu
    |       `-- fmt.cc
    `-- libfmt.a
```

Headers or no headers?

Two possible reasons:

- They are needed by the module interface units
- If we want dual support:
  - `#include <fmt/core.h>`
  - `import fmt;`

Packaging module libraries - how?

```
No package
(include sources)
```

No packaging

```
├── lib
│   ├── libfmt.a
│   └── cmake
│   └── cxx/bmi
│       ├── fmt.bmi
```

BMI + binary library (shared or static)

```
├── lib
│   ├── libfmt.so
│   └── cxx/miu
│       ├── fmt.cc
```

Module interfaces + binary library (shared or static)

```
├── lib
│   └── cxx/miu
│       ├── fmt.cc
```

Module only

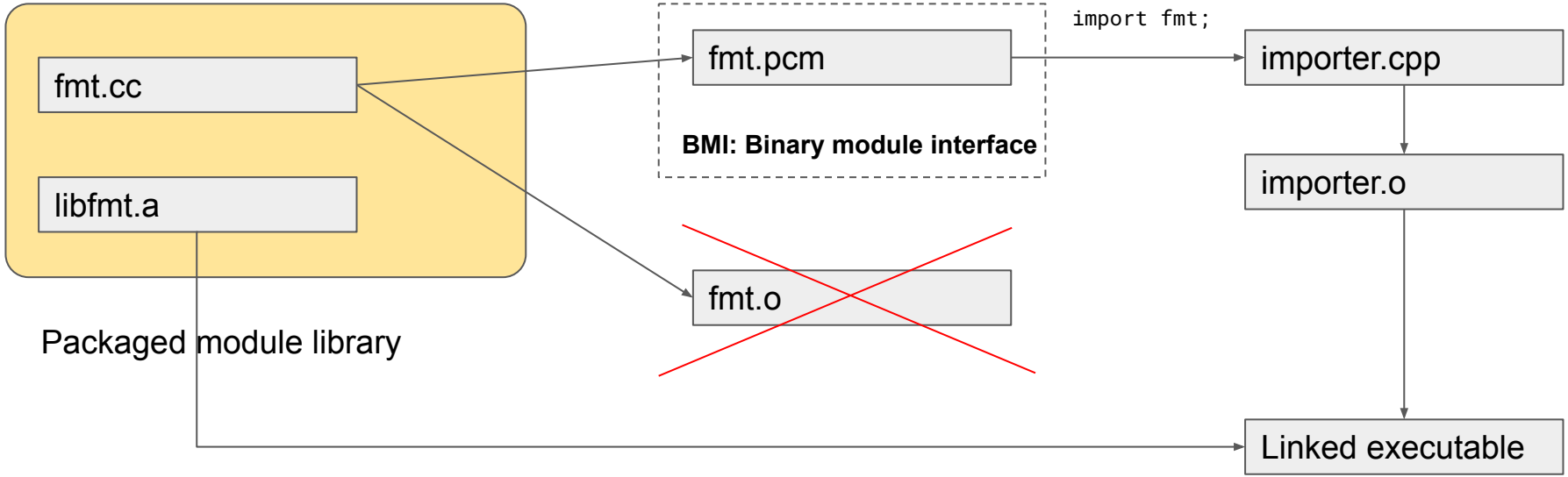
Module units only

```
|--  lib
    |-- cmake
    |   `-- fmt
    |       `-- fmt-config.cmake
    |-- cxx
    |   `-- miu
    |       `-- fmt.cxx
    `   libfmt.a
```

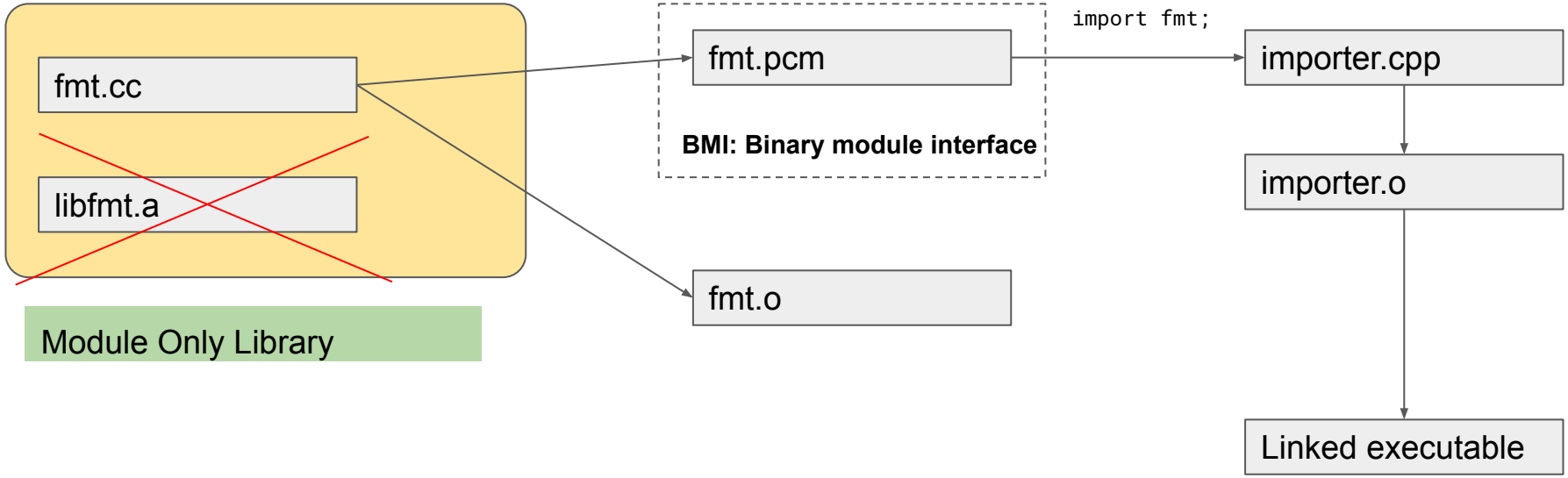
No binary files distributed with the package!

- **Header only libraries** are very popular
  - Roughly 1 in 4 recipes in Conan Center are header only
  - Perceived as “easy” to integrate
- **Module only libraries**
  - `export module foobar;` (named modules)
  - What to do with these?
  - At the very least, invoke compiler to generate the BMI

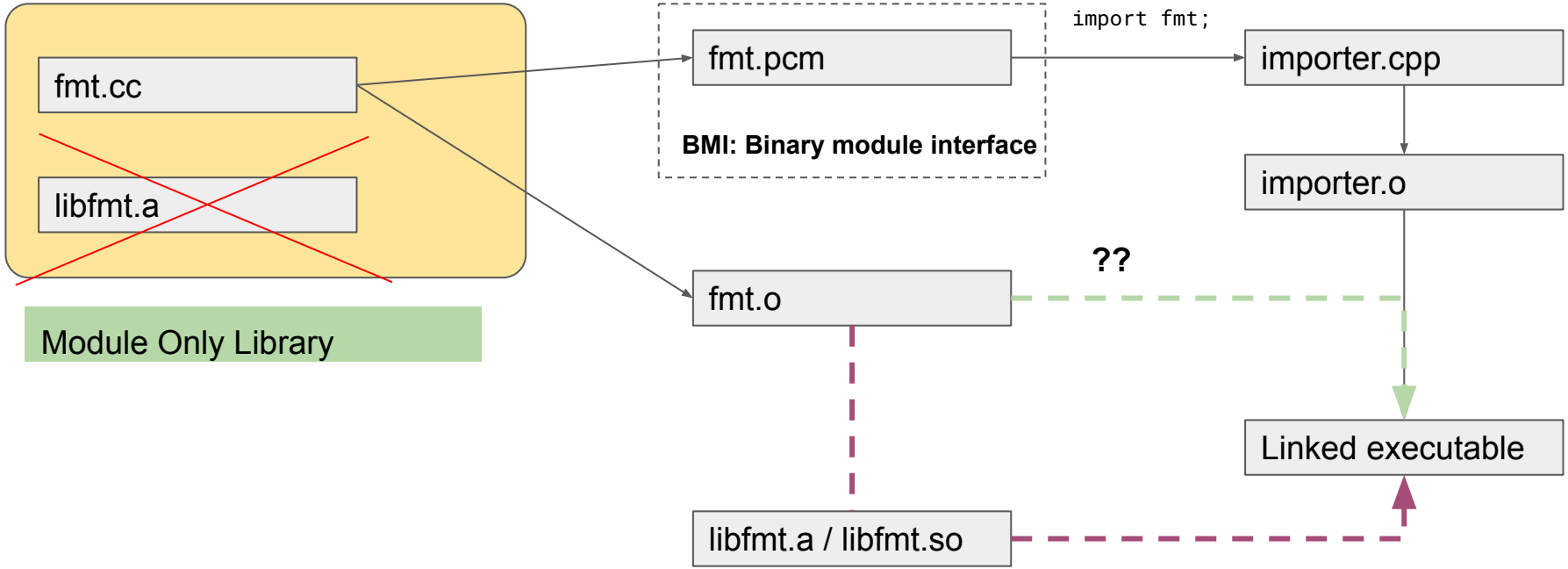
Packaging the module interfaces (cont'd)



Module units only - cont'd



Module units only - cont'd



Model units only - cont'd

COMPILER EXPLORER Add... More Templates Share Policies Other

C++ source #1 C++

```cpp
export module foobar;

export namespace foobar {

    constexpr int byfive(int foo) {
        return foo * 5;
    }

    template <typename T1, typename T2>
    auto sum(T1 first, T2 second) {
        return first + second;
    }
}
```

x86-64 gcc 13.2 (Editor #1) x86-64 gcc 13.2 -std=c++20 -fmodules-ts -c

```asm
initializer for module foobar:
        push    rbp
        mov     rbp, rsp
        nop
        pop     rbp
        ret
```

Module units only - cont’d

- We only have source files
  - Maybe some compiler flags
- How to proceed …
  - Build into library?
  - Shared or static?
  - Who makes that decision?
  - Propagate and link object files directly?
  - Discard object files and propagate BMIs if the module supports it?

Packaging module libraries

```
No package
(include sources)
```

No packaging

```
├── lib
│   ├── libfmt.a
│   └── cmake
│   └── cxx/bmi
│       ├── fmt.bmi
```

BMI + binary library (shared or static)

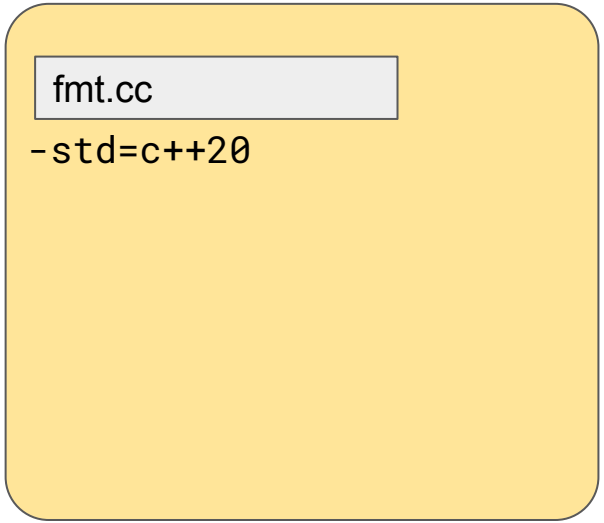
```
├── lib
│   ├── libfmt.so
│   └── cxx/miu
│       ├── fmt.cxx
```

Module interfaces + binary library (shared or static)

```
├── lib
│   └── cxx/miu
│       ├── fmt.cxx
```

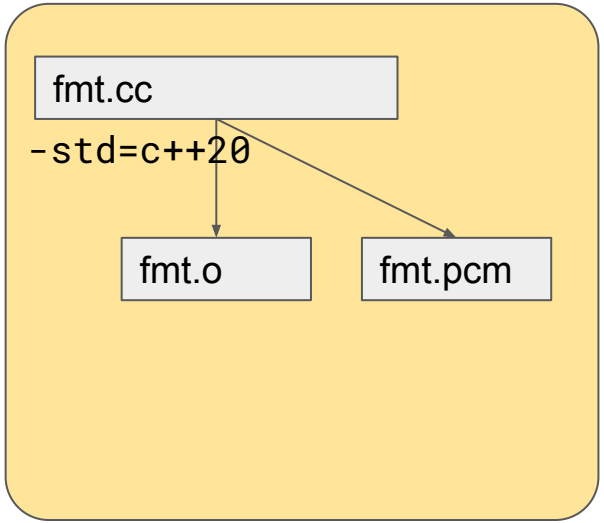
Module only

BMI compatibility



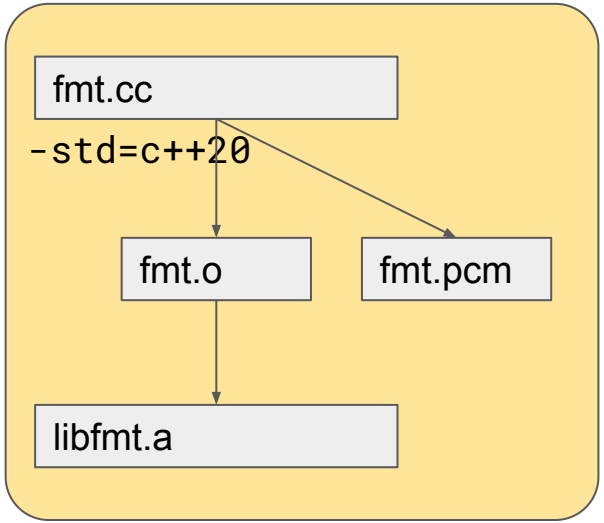
Packaged module library

BMI compatibility



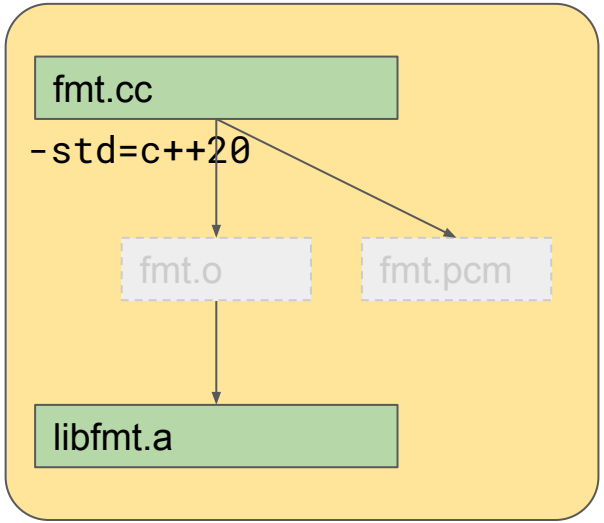
Packaged module library

BMI compatibility



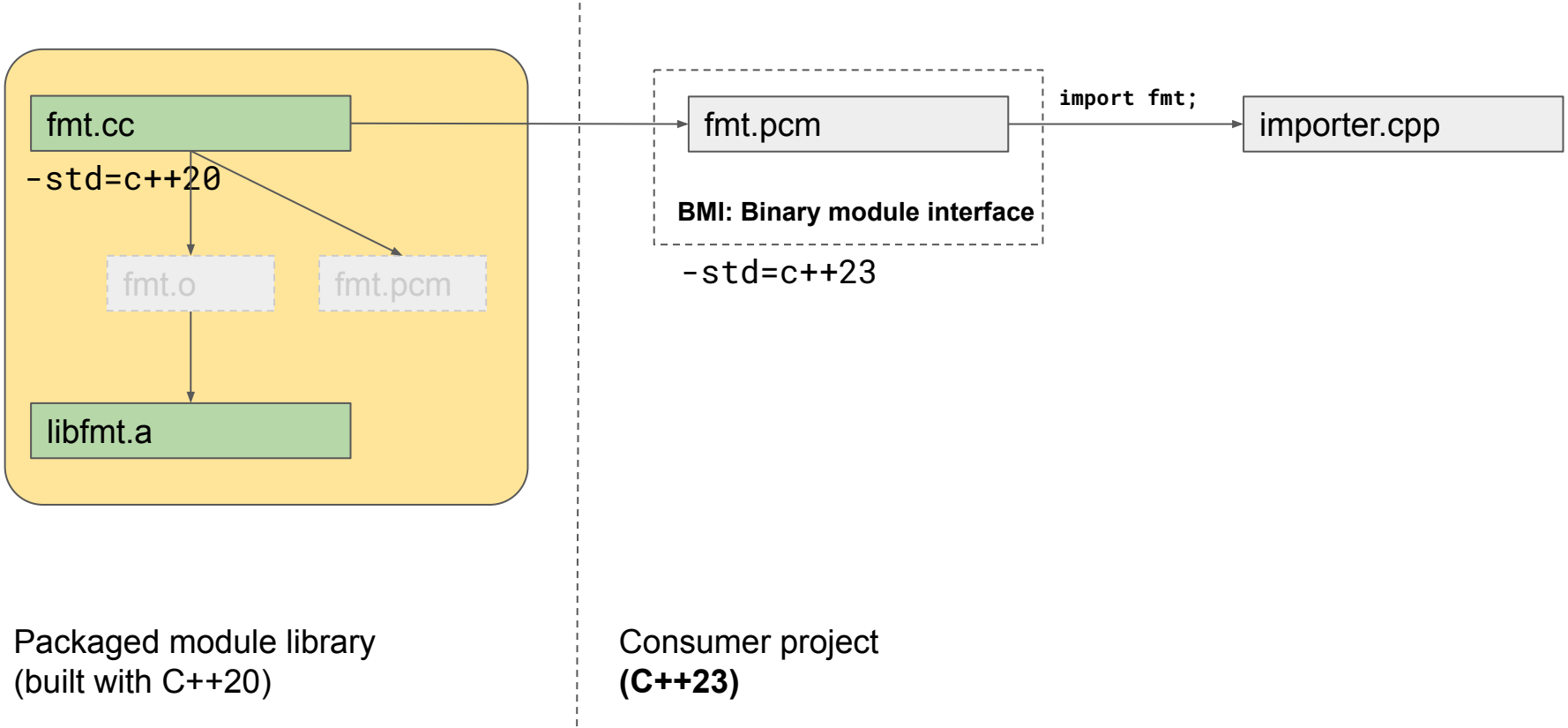
Packaged module library

BMI compatibility

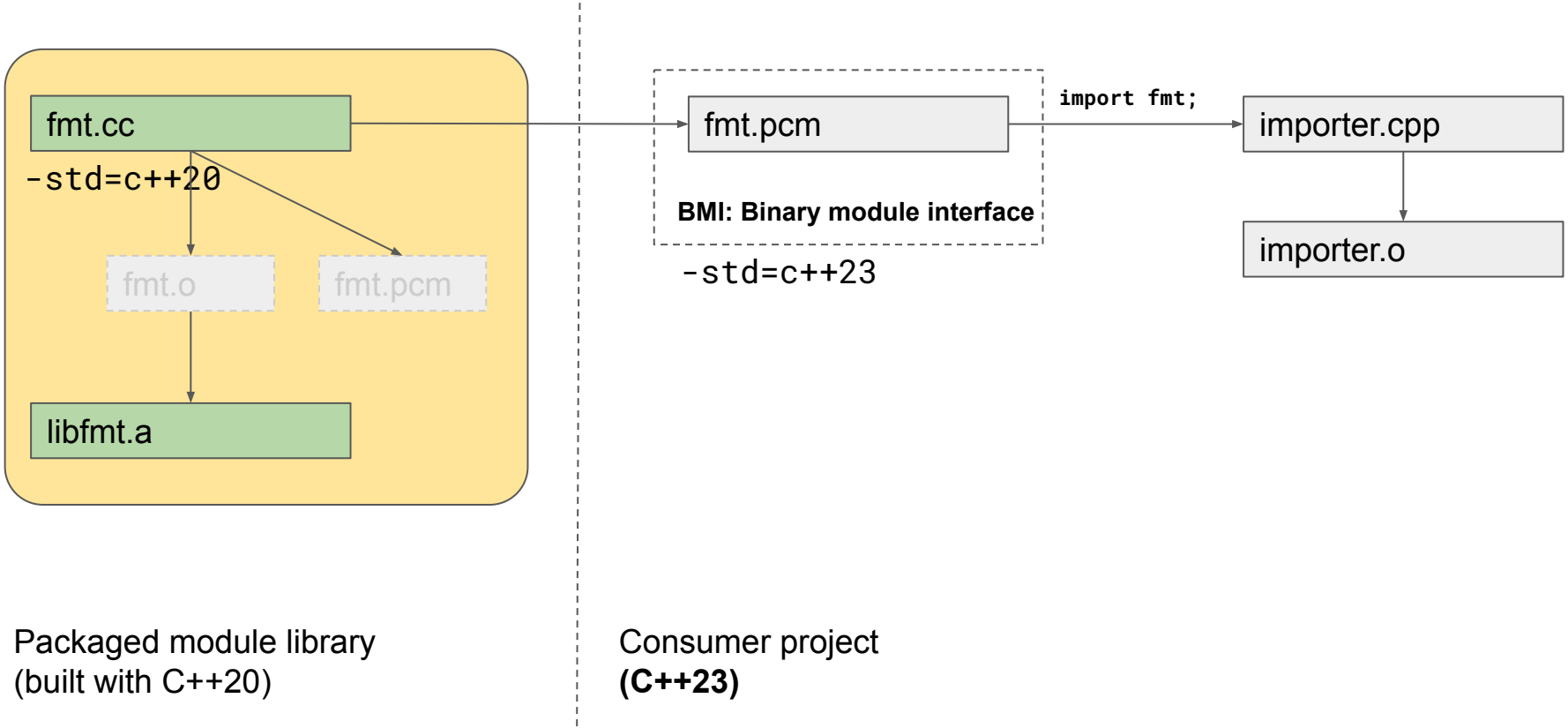


Packaged module library

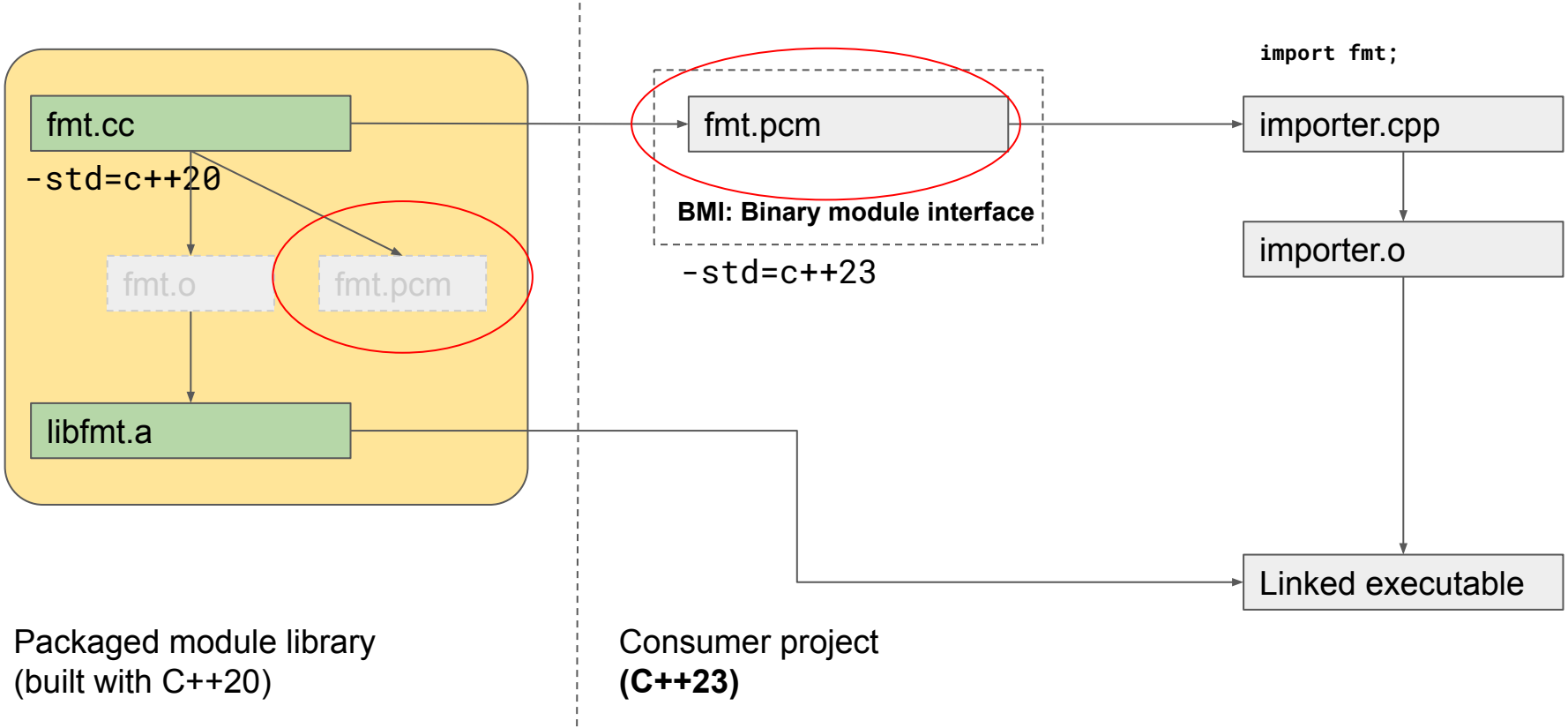
BMI compatibility



BMI compatibility



BMI compatibility



BMI compatibility

```
include(FetchContent)
FetchContent_Declare(fmt
GIT_REPOSITORY https://github.com/jcar87/fmt.git
# 10.1.1 built as module
GIT_TAG 28fbcaa72b6224f7824672a
OVERRIDE_FIND_PACKAGE
)

FetchContent_MakeAvailable(fmt)

find_package(fmt REQUIRED)

add_executable(hello hello_wor
target_link_libraries(hello PRI
target_compile_options(hello PR
```

```
[7/8] Building CXX object CMakeFiles/hello.dir/hello_world.cpp.o
FAILED: CMakeFiles/hello.dir/hello_world.cpp.o
/opt/gcc13/bin/g++  -I/cxx-modules/basic-fmt-consumer/build.gcc/_deps/fmt-src/include -O3
-DNDEBUG -std=gnu++20 -std=c++23 -MD -MT CMakeFiles/hello.dir/hello_world.cpp.o -MF
CMakeFiles/hello.dir/hello_world.cpp.o.d -fmodules-ts
-fmodule-mapper=CMakeFiles/hello.dir/hello_world.cpp.o.modmap -MD -fdeps-format=p1689r5 -x
c++ -o CMakeFiles/hello.dir/hello_world.cpp.o -c
/cxx-modules/basic-fmt-consumer/hello_world.cpp
In module imported at /cxx-modules/basic-fmt-consumer/hello_world.cpp:2:1:
fmt: error: language dialect differs 'C++20/coroutines', expected 'C++23/coroutines'
fmt: error: failed to read compiled module: Bad file data
fmt: note: compiled module file is '_deps/fmt-build/CMakeFiles/fmt.dir/fmt.gcm'
fmt: fatal error: returning to the gate for a mechanical issue
compilation terminated.
```

BMI compatibility (cont'd)


```
add_subdirectory(flux)
add_subdirectory(fmt)
add_subdirectory(argparse)

add_library(foo)
target_sources(foo
  PUBLIC
    FILE_SET cxx_modules TYP
     foo.cxx
)

add_executable(hello main.cx
target_link_libraries(hello
      PRIVATE
        foo fmt::fmt flux::f
```

```
/lib/llvm-16/bin/clang++  -I/usr/lib/gcc/aarch64-linux-gnu/12/include
-I/cxx-modules/cmake-example/fmt/include -I/cxx-modules/cmake-example/flux/include -O3
-DNDEBUG -std=c++20 -fsized-deallocation -faligned-allocation -fchar8_t -MD -MT
CMakeFiles/hello.dir/main.cxx.o -MF CMakeFiles/hello.dir/main.cxx.o.d
@CMakeFiles/hello.dir/main.cxx.o.modmap -o CMakeFiles/hello.dir/main.cxx.o -c
/cxx-modules/cmake-example/main.cxx
error: sized deallocation was disabled in PCH file but is currently enabled
error: module file CMakeFiles/foo.dir/foo.pcm cannot be loaded due to a configuration
mismatch with the current compilation [-Wmodule-file-config-mismatch]
2 errors generated.
ninja: build stopped: subcommand failed.
```

BMI compatibility (cont'd)

```
import foo;
import fmt;
import flux;
import argparse;
```

my_app.cxx

```
add_library(argparse STATIC)
add_library(argparse::argparse ALIAS argparse)
target_sources(argparse
  PUBLIC
    FILE_SET modules TYPE CXX_MODULES
    BASE_DIRS module
    FILES argparse.ixx
)
target_compile_features(argparse PUBLIC cxx_std_20)
if(CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  target_compile_options(argparse PUBLIC -fsized-deallocation)
endif()
```

BMI compatibility (cont'd)

```
import foo;
import fmt;
import flux;
import argparse;
```

my_app.cxx

```
add_executable(my_app my_app.cxx)
target_link_libraries(my_app PRIVATE
                      foo
                      fmt::fmt
                      flux::flux
                      argparse::argparse
```

BMIs built with different flags

-fsized-deallocation

BMI compatibility - cont’d

- Dependency scanning give us the correct build order
- But it may be difficult for the build system to reason that we will have incompatible BMIs
- We may need to be build multiple BMI variants for the same module

Error reporting

Errors related to dependencies are frustrating.
Modules add a new dimension and new “classes” of errors

- Generating a BMI
- Locating a BMI (mostly solved my dependency scanning)
- Loading a BMI
  - Mostly due to compiler option mismatches
- Compiler errors calling the functions declared in a module
- Linker errors

Error reporting (cont'd)

Translating a module interface unit

```
FAILED: CMakeFiles/fmt.dir/src/fmt.cc.o CMakeFiles/fmt.dir/fmt.gcm
/opt/gcc13/bin/g++  -I/cxx-modules/fmt-tests/fmt/include -std=c++20 -fvisibility=hidden
-fvisibility-inlines-hidden -MD -MT CMakeFiles/fmt.dir/src/fmt.cc.o -MF
CMakeFiles/fmt.dir/src/fmt.cc.o.d -fmodules-ts
-fmodule-mapper=CMakeFiles/fmt.dir/src/fmt.cc.o.modmap -MD -fdeps-format=p1689r5 -x c++ -o
CMakeFiles/fmt.dir/src/fmt.cc.o -c /cxx-modules/fmt-tests/fmt/src/fmt.cc
/cxx-modules/fmt-tests/fmt/src/fmt.cc:73:8: internal compiler error: in core_vals, at
cp/module.cc:6262
   73 | export module fmt;
      |        ^~~~~~
```

Error reporting (cont'd)

Translating a module interface unit

```
[6/7] Building CXX object CMakeFiles/hello.dir/hello_world.cpp.o
FAILED: CMakeFiles/hello.dir/hello_world.cpp.o
/opt/gcc13/bin/g++ -DFMT_SHARED -isystem /cxx-modules/fmt-tests/fmt/build.gcc2/install/include -O3 -DNDEBUG -std=c++
20 -MD -MT CMakeFiles/hello.dir/hello_world.cpp.o -MF CMakeFiles/hello.dir/hello_world.cpp.o.d -fmodules-ts -fmodule
-mapper=CMakeFiles/hello.dir/hello_world.cpp.o.modmap -MD -fdeps-format=p1689r5 -x c++ -o CMakeFiles/hello.dir/hello
_world.cpp.o -c /cxx-modules/fmt-tests/fmt-package-consumer/hello_world.cpp
In file included from /opt/gcc13/include/c++/13.2.0/bits/locale_facets.h:2687,
                 from /opt/gcc13/include/c++/13.2.0/bits/basic_ios.h:37,
                 from /opt/gcc13/include/c++/13.2.0/ios:46,
                 from /opt/gcc13/include/c++/13.2.0/istream:40,
                 from /opt/gcc13/include/c++/13.2.0/sstream:40,
                 from /opt/gcc13/include/c++/13.2.0/chrono:45,
                 from /cxx-modules/fmt-tests/fmt/build.gcc2/install/lib/cxx/miu/src/fmt.cc:7,
of module fmt, imported at /cxx-modules/fmt-tests/fmt-package-consumer/hello_world.cpp:1:
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.tcc: In instantiation of 'const std::__numpunct_cache@fmt<_CharT>*
std::__use_cache@fmt<std::__numpunct_cache@fmt<_CharT> >::operator()(const std::locale@fmt&) const [with _CharT = ch
ar]':
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.tcc:384:33:   required from '_InIter std::num_get@fmt<_CharT, _InIt
er>::_M_extract_int(_InIter, _InIter, std::ios_base@fmt&, std::ios_base@fmt::iostate&, _ValueT&) const [with _ValueT
 = long int; _CharT = char; _InIter = std::istreambuf_iterator@fmt<char, std::char_traits@fmt<char> >; std::ios_base
@fmt::iostate = std::ios_base@fmt::iostate]'
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.h:2225:30:   required from 'std::num_get@fmt<_CharT, _InIter>::iter
_type std::num_get@fmt<_CharT, _InIter>::do_get(iter_type, iter_type, std::ios_base@fmt&, std::ios_base@fmt::iostate
&, long int&) const [with _CharT = char; _InIter = std::istreambuf_iterator@fmt<char, std::char_traits@fmt<char> >;
iter_type = std::istreambuf_iterator@fmt<char, std::char_traits@fmt<char> >; std::ios_base@fmt::iostate = std::ios_b
ase@fmt::iostate]'
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.h:2223:7:   required from here
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.tcc:67:17: error: invalid use of non-static member function 'std::_
_numpunct_cache@fmt<_CharT>::~__numpunct_cache() [with _CharT = char]'
   67 |                 delete __tmp;
      |                 ^~~~~~~~~~~~
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.h:1651:5: note: declared here
 1651 |     __numpunct_cache<_CharT>::~__numpunct_cache()
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/gcc13/include/c++/13.2.0/bits/locale_facets.tcc:67: confused by earlier errors, bailing out
ninja: build stopped: subcommand failed.
```

```
import fmt;
```

👎

```
#include <locale>
import fmt;
```

👍

Error reporting (cont’d)

Linker errors

```
FAILED: hello
: && /opt/gcc13/bin/g++ -O3 -DNDEBUG  CMakeFiles/hello.dir/hello_world.cpp.o -o hello
-Wl,-rpath,/cxx-modules/fmt-tests/fmt/build.gcc2/install/lib
/cxx-modules/fmt-tests/fmt/build.gcc2/install/lib/libfmt.so.10.1.0 && :
/usr/bin/ld: CMakeFiles/hello.dir/hello_world.cpp.o: in function `_GLOBAL__sub_I_main':
hello_world.cpp:(.text.startup+0x30): undefined reference to `initializer for module fmt'
collect2: error: ld returned 1 exit status
ninja: build stopped: subcommand failed.
```

Conclusions

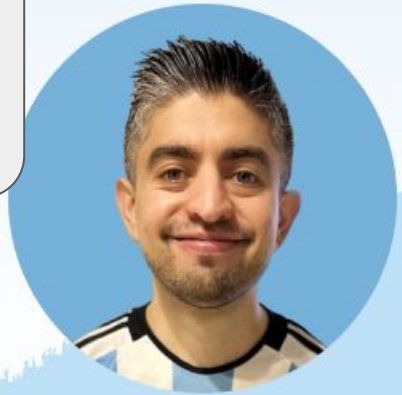
- Slowly, more libraries are adding experimental support for modules
  - Usually as a build time option
- Dynamic dependency scanning solves build order issues
  - Requires recent CMake, and recent compiler versions, or pure Visual Studio projects
  - If you build all your dependencies as part of the same project
- CMake is adding module support for imported targets
  - Requires CMake on both library and consumer sides
  - No standard way of communicating module metadata
- Can package managers help?
  - Yes, if you have **full control** of how your dependencies are built
  - Packaging BMIs is not practical due to high chance of BMI incompatibilities
  - Little help otherwise - build system support needed for dependency scanning and building BMIs for external modules
- Lots of questions around BMI compatibility
  - Even within the same project!

Questions?

C++20 Modules: The Packaging and Binary Redistribution Story

CONAN
C/C++ Package Manager

THANK YOU!
⭐⭐⭐



Luis Caro Campos
SW Tech Lead, JFrog