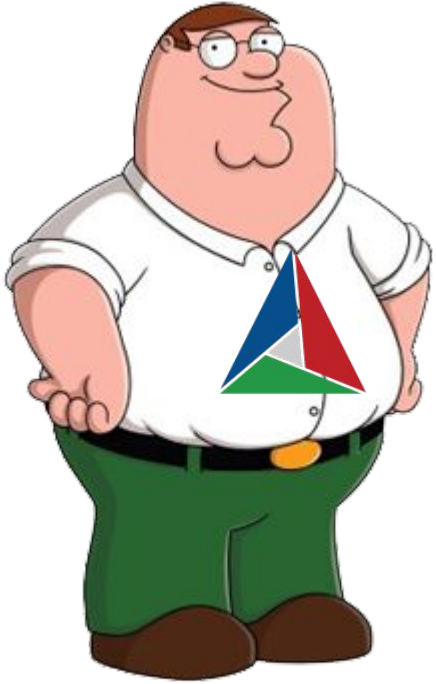
import CMake;
// Mastering C++ Modules

BILL HOFFMAN

CMake / Sandal / Kitware GUY



kitware

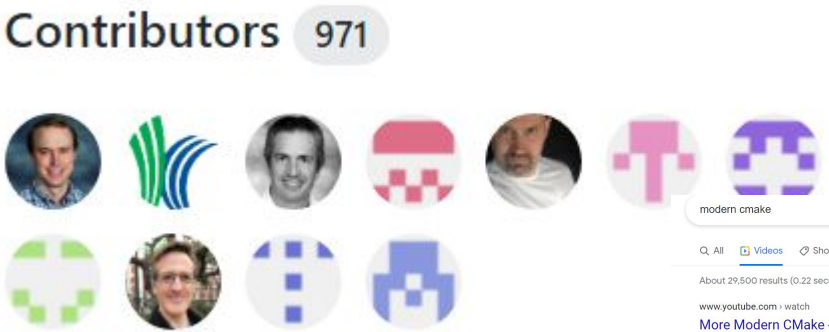
Bill Hoffman

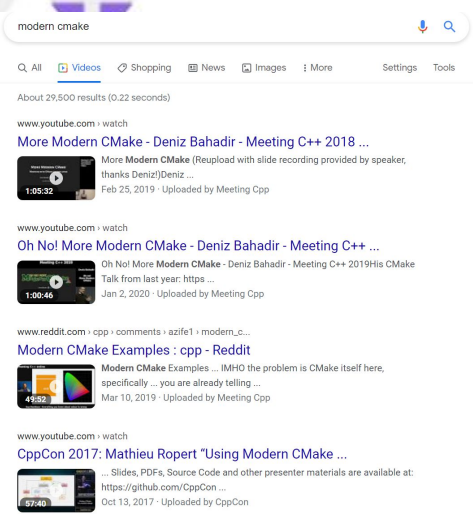
About Me

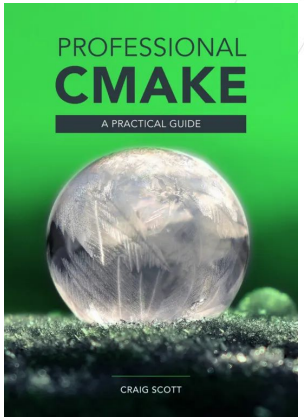
- 1990-1999 GE research in a Computer Vision group.
  - Moved them from Symbolics Lisp Machines to C++ on Solaris and later Linux/Windows
  - Was the build and software library guy (gmake/autotools)
- 1998-Present Kitware
  - Lots of development in CMake/ITK/VTK
  - Mostly management now, finding funding for CMake

kitware

CMake is a Community Effort



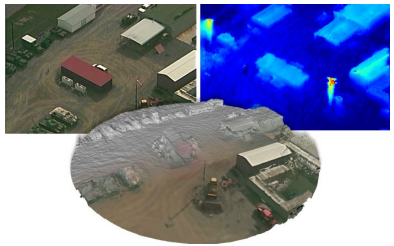


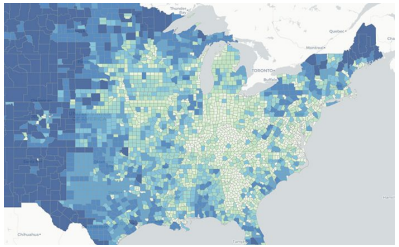


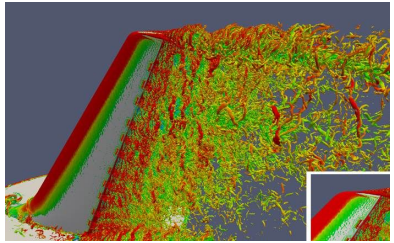
Brad King, Ben Boeckel, Craig Scott are the CMake guys!

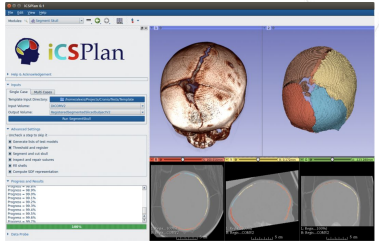
kitware

Kitware Overview / Built on open source









Computer Vision

Data and Analytics

Scientific Computing

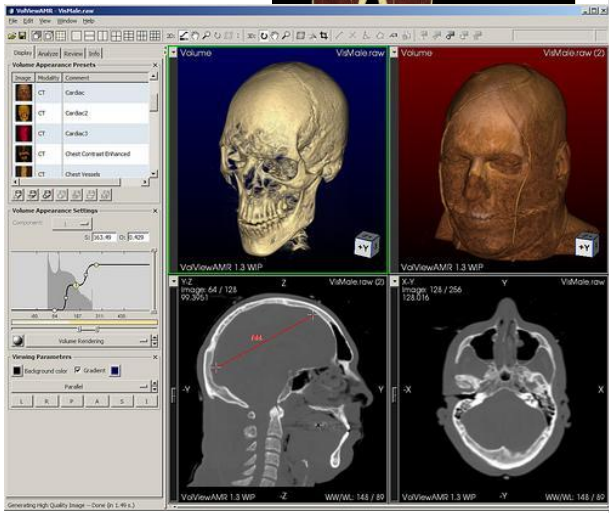
Medical Computing

Software Solutions

kitware

Where did CMake come from?

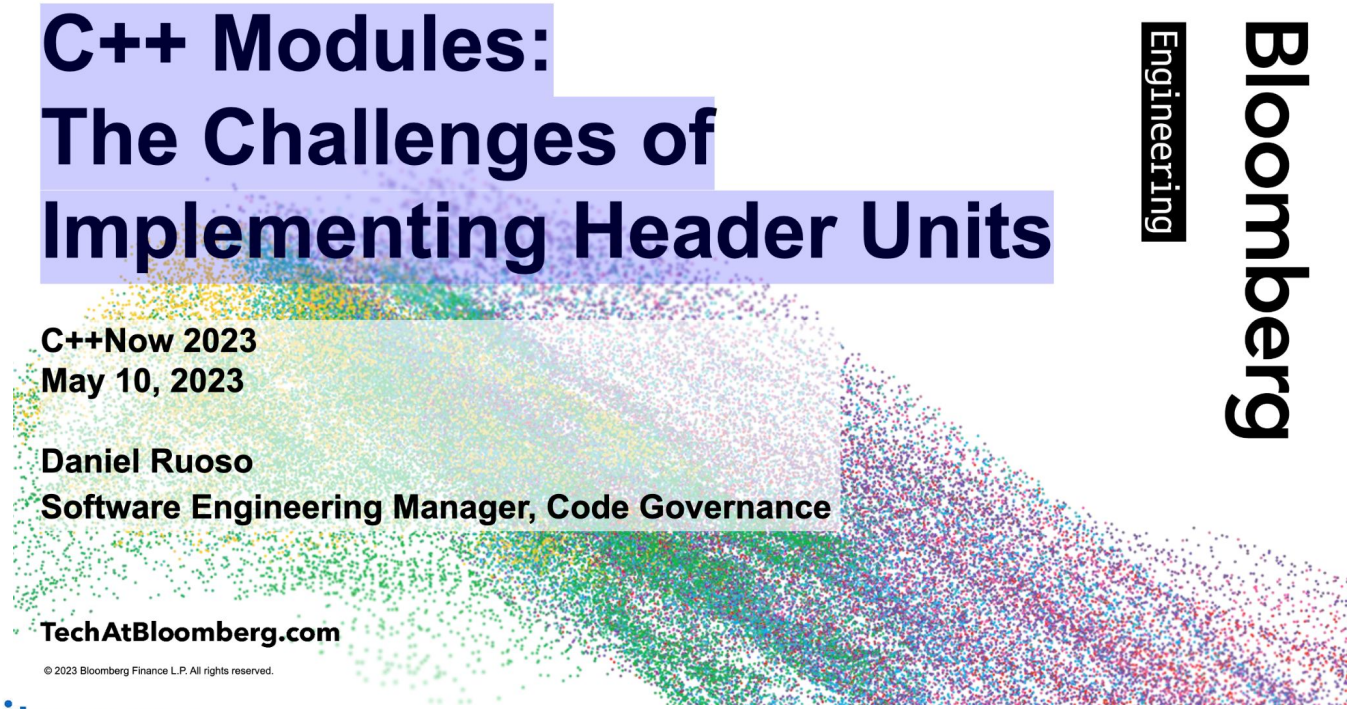
- Kitware was the lead engineering team for the Insight Segmentation and Registration Toolkit (ITK) http://www.itk.org
  - tasked with making it build on Unix/Windows/Mac
- Funded by National Library of Medicine (NLM): part of the Visible Human Project
  - Data CT/MR/Slice 1994/1995
  - Code (ITK) 1999
- CMake Release-1-0 branch created in 2001
- Since then funding from many projects and outside contributions



kitware

Enough with the Background Let's Talk Modules!

C++ Named Modules Only



kitware

Does CMake support C++20 modules

Ask ChatGPT 2023

Yes, CMake has added support for C++20 modules in version 3.21. This means you can use CMake to build C++20 projects that make use of the new module system.

To use C++20 modules with CMake, you'll need to use a compiler that supports them, such as Clang 12 or later, or GCC 11 or later. You'll also need to specify the language standard and module generation options in your CMakeLists.txt file. Here's an example:

scss Copy code

```
cmake_minimum_required(VERSION 3.21)

project(MyProject)

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

add_executable(MyExecutable main.cpp)

target_compile_features(MyExecutable PRIVATE cxx_std_20)
```

Ask ChatGPT 2024

Yes, CMake does support C++20 modules, but the level of support can vary depending on the compiler and its version. As of CMake 3.19 and later, there are features to help you work with C++20 modules.

To use C++20 modules in CMake, you typically need to:

1. **Specify the C++** enable C++20 fe
2. **Use the** `CXX_MOI` the `MODULE` opti
3. **Check compiler** C++20 modules. may still be expe
4. **Use module inte** your CMake com

For example:

cmake — Copy code

```
cmake_minimum_required(VERSION 3.19)
project(MyProject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)

add_library(my_module MODULE my_module.cpp)
target_sources(my_module PRIVATE my_module.ixx)
```

Make sure to consult the CMake documentation and the specific compiler documentation for details on any limitations or additional settings required for using C++20 modules effectively.

kitware

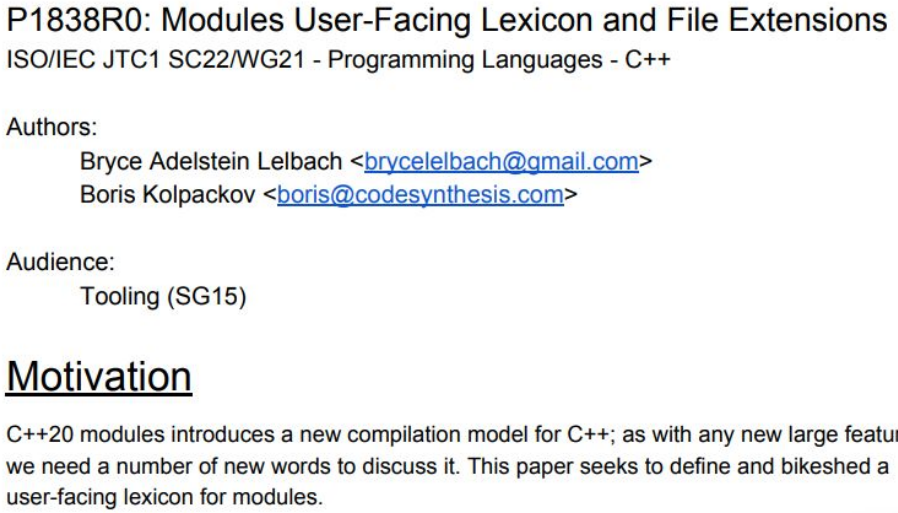
Simple Module

```
export module B;
export void b() { }
```

kitware

Built Module Interface (BMI)

- MSVC
  - .ifc file
- G++
  - .gcm file
- Clang
  - .pcm
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1838r0.pdf



P1838R0: Modules User-Facing Lexicon and File Extensions

ISO/IEC JTC1 SC22/WG21 - Programming Languages - C++

Authors:

Bryce Adelstein Lelbach <brycelelbach@gmail.com>

Boris Kolpackov <boris@codesynthesis.com>

Audience:

Tooling (SG15)

Motivation

C++20 modules introduces a new compilation model for C++; as with any new large feature, we need a number of new words to discuss it. This paper seeks to define and bikeshed a user-facing lexicon for modules.

kitware

Simple Example
(build order order matters)

B.cpp:

```
export module B;
export void b() { }
```

A.cpp:

```
export module A;
import B;
export void a(){ b();}
```

```
 cl -std:c++20  -interface -c A.cpp

A.cpp
A.cpp(2): error C2230: could not find module 'B'
A.cpp(3): error C3861: 'b': identifier not found
```

kitware

Simple Example (build order matters)

B.cpp:

```
export module B;
export void b() { }
```

A.cpp:

```
export module A;
import B;
export void a(){ b();}
```

```
cl -std:c++20  -interface -c B.cpp
B.cpp

cl -std:c++20  -interface -c A.cpp
A.cpp
$ ls
A.cpp  A.ifc  A.obj  B.cpp  B.ifc  B.obj
```

kitware

Chicken and the Egg

- Modules require the build system to know which files produce which BMI files and which files consume them
- Need to parse/compile file to find that out
- So… We need to compile the code before we can compile the code….

CMake had 17 years experience with modules, Fortran ones

- 2005 Initial makefile support for modules added to CMake:

commit 19f977bad7261d9e8f8d6c5d2764c079d35cc014
Author: Brad King <brad.king@kitware.com>
Date: Wed Jan 26 15:33:38 2005 -0500
ENH: Added Fortran dependency scanner implementation.

- Added support to ninja in 2015 for Fortran dep file depends - funded by the Trilinos project forked ninja
  - dyndep https://ninja-build.org/manual.html#ref_dyndep
- May 2019 ninja merged all of the changes to support Fortran because of C++ modules!

kitware

https://ninja-build.org/manual.html#ref_dyndep

Some use cases require implicit dependency information to be dynamically discovered from source file content *during the build* in order to build correctly on the first run (e.g., Fortran module dependencies). This is unlike header dependencies which are only needed on the second run and later to rebuild correctly.

How does CMake do this

- A Fortran parser based off of makedepf90
  - CAN NOT/WILL NOT DO THIS FOR C++
- Patches made to ninja build tool now upstreamed
- The dynamic dependency collator inside CMake

Feb 2019

Paper Describing CMake Fortran Modules

[Tooling] [D1483] How CMake supports Fortran modules and its applicability to C++

Ben Boeckel ben.boeckel at kitware.com
Fri Feb 8 16:55:20 CET 2019

- Previous message: [Tooling] Clang Modules and build system requirements
- Next message: [Tooling] Fwd: [D1483] How CMake supports Fortran modules and its applicability to C++
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hi,

Here is copy of Kitware's paper to be discussed at Kona. I have a PDF,
but it was too large to attach to the list. I'll be at Kona, but the
other authors are not able to make it.

An HTML version is hosted here:

    https://mathstuf.fedorapeople.org/fortran-modules/fortran-modules.html

Feedback welcome.

Thanks,

--Ben
```

---

https://mathstuf.fedorapeople.org/fortran-modules/fortran-modules.html

kitware

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1689r5.html

defining a format for compilers p1689

Format for describing dependencies of source files

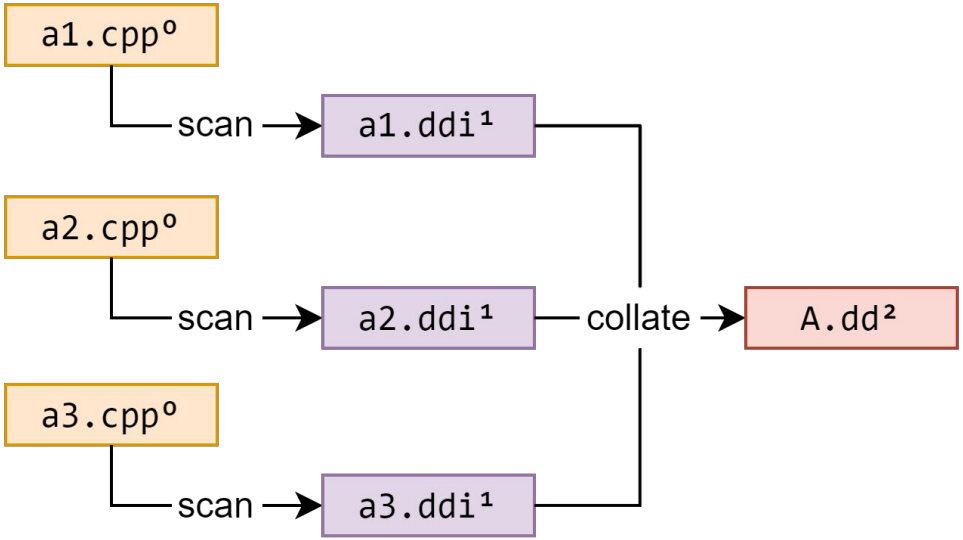
Ben Boeckel, Brad King
<ben.boeckel@kitware.com, brad.king@kitware.com>
version P1689R5, 2022-06-03

Table of Contents

Document number	ISO/IEC/JTC1/SC22/WG21/P1689R5
Date	2022-06-03
Reply-to	Ben Boeckel, Brad King, ben.boeckel@kitware.com, brad.king@kitware.com
Audience	EWG (Evolution), SG15 (Tooling)

kitware

Per target scanning



scan: done by compiler (i.e. cl -scanDependencies)

collate: cmake -E cmake_ninja_dyndep

kitware

Where we are today

- Support for p1689r5 in compiler releases
  - Visual studio 2022
  - Clang 16 and newer
  - GCC 14
- CMake 3.28 support for modules
- CMake 3.29 experimental support for import std

CMake (Module able) Generators

- -GNinja
- -G"Visual Studio 17 2022" (aka MSBuild)

back to A.cpp msvc

```
cl -std:c++20 -scanDependencies A.json -c A.cpp
```

```
{
   "version": 1,
   "revision": 0,
   "rules": [
      {
         "primary-output": "A.obj",
         "outputs": [
            "A.json",
            "A.ifc"
         ],
         "provides": [
            {
               "logical-name": "A",
               "source-path": "c:\\users\\hoffman\\work\\cxxmodules\\cxx-modules-examples\\simple\\a.cpp"
            }
         ],
         "requires": [
            {
               "logical-name": "B"
            }
         ]
      }
   ]
}
```

kitware

back to B.cpp msvc

```
cl -std:c++20 -scanDependencies B.json -c B.cpp
```

```
{
   "version": 1,
   "revision": 0,
   "rules": [
      {
         "primary-output": "B.obj",
         "outputs": [
            "B.json",
            "B.ifc"
         ],
         "provides": [
            {
               "logical-name": "B",
               "source-path": "c:\\users\\hoffman\\work\\cxxmodules\\cxx-modules-examples\\simple\\b.cpp"
            }
         ],
         "requires": []
      }
   ]
}
```

kitware

FILE_SET

File Sets

New in version 3.23.

```
target_sources(<target>
  [<INTERFACE|PUBLIC|PRIVATE>
   [FILE_SET <set> [TYPE <type>] [BASE_DIRS <dirs>...] [FILES <files>...]]...
  ]...)
```

Adds a file set to a target, or adds files to an existing file set. Targets have zero or more named file sets. Each file set has a name, a type, a scope of `INTERFACE`, `PUBLIC`, or `PRIVATE`, one or more base directories, and files within those directories. The only acceptable type is `HEADERS`. The optional default file sets are named after their type. The target may not be a custom target or `FRAMEWORK` target.

kitware

Header Only Libraries (with FILE_SET)

```
add_library(Eigen INTERFACE)

target_sources(Eigen INTERFACE
  FILE_SET HEADERS
  BASE_DIRS src
  FILES src/eigen.h src/vector.h src/matrix.h
)

install(TARGETS Eigen EXPORT eigenExport
  FILE_SET HEADERS DESTINATION include/Eigen)
install(EXPORT eigenExport NAMESPACE Upstream::
  DESTINATION lib/cmake/Eigen
)
add_executable(exe1 exe1.cpp)
target_link_libraries(exe1 Eigen)
```

Running Install

```
# running this
cmake.exe --install-prefix=$HOME/Work/cxxmodules/file_set/b/inst .
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/hoffman/Work/cxxmodules/file_set/b


$ ninja install
[0/1] Install the project...
-- Install configuration: "Debug"
-- Installing: C:/Users/hoffman/Work/cxxmodules/file_set/b/inst/include/Eigen/eigen.h
-- Installing: C:/Users/hoffman/Work/cxxmodules/file_set/b/inst/include/Eigen/vector.h
-- Installing: C:/Users/hoffman/Work/cxxmodules/file_set/b/inst/include/Eigen/matrix.h
-- Installing: C:/Users/hoffman/Work/cxxmodules/file_set/b/inst/lib/cmake/Eigen/eigenExport.cmake
```

File Sets

New in version 3.23.

```
target_sources(<target>
  [<INTERFACE|PUBLIC|PRIVATE>
   [FILE_SET <set> [TYPE <type>] [BASE_DIRS <dirs>...] [FILES <files>...]]...
  ]...)
```

Adds a file set to a target, or adds files to an existing file set. Targets have zero or more named file sets. Each file set has a name, a type, a scope of `INTERFACE`, `PUBLIC`, or `PRIVATE`, one or more base directories, and files within those directories. The acceptable types include:

`HEADERS`

Sources intended to be used via a language's `#include` mechanism.

`CXX_MODULES`

New in version 3.28.

Sources which contain C++ interface module or partition units (i.e., those using the `export` keyword). This file set type may not have an `INTERFACE` scope except on `IMPORTED` targets.

Compile A.cpp and B.cpp with CMake

```
cmake_minimum_required(VERSION 3.23)
project(simple CXX)
set(CMAKE_CXX_STANDARD 20)
add_library(simple)

target_sources(simple
  PRIVATE
  FILE_SET cxx_modules TYPE CXX_MODULES FILES
  A.cpp B.cpp
)
```

Compile A.cpp and B.cpp configure with CMake

```
cmake -GNinja ..
-- The CXX compiler identification is Clang 20.0.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Users/hoffman/Work/llvm/llvm-inst/bin/clang++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.8s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/hoffman/Work/modules/simple/b
```

Compile A.cpp and B.cpp with CMake Clang/ninja

```
$hoffman@caprica b % ninja -v
[1/10] "/Users/hoffman/Work/llvm/llvm-inst/bin/clang-scan-deps" -format=p1689 -- /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot
/Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -x c++ /Users/hoffman/Work/modules/simple/main.cxx -c -o CMakeFiles/hello.dir/main.cxx.o -resource-dir
"/Users/hoffman/Work/llvm/llvm-inst/lib/clang/20" -MT CMakeFiles/hello.dir/main.cxx.o.ddi -MD -MF CMakeFiles/hello.dir/main.cxx.o.ddi.d >
CMakeFiles/hello.dir/main.cxx.o.ddi.tmp && mv CMakeFiles/hello.dir/main.cxx.o.ddi.tmp CMakeFiles/hello.dir/main.cxx.o.ddi
[2/10] "/Users/hoffman/Work/llvm/llvm-inst/bin/clang-scan-deps" -format=p1689 -- /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot
/Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -x c++ /Users/hoffman/Work/modules/simple/A.cpp -c -o CMakeFiles/foo.dir/A.cpp.o -resource-dir
"/Users/hoffman/Work/llvm/llvm-inst/lib/clang/20" -MT CMakeFiles/foo.dir/A.cpp.o.ddi -MD -MF CMakeFiles/foo.dir/A.cpp.o.ddi.d > CMakeFiles/foo.dir/A.cpp.o.ddi.tmp && mv
CMakeFiles/foo.dir/A.cpp.o.ddi.tmp CMakeFiles/foo.dir/A.cpp.o.ddi
[3/10] "/Users/hoffman/Work/llvm/llvm-inst/bin/clang-scan-deps" -format=p1689 -- /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot
/Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -x c++ /Users/hoffman/Work/modules/simple/B.cpp -c -o CMakeFiles/foo.dir/B.cpp.o -resource-dir
"/Users/hoffman/Work/llvm/llvm-inst/lib/clang/20" -MT CMakeFiles/foo.dir/B.cpp.o.ddi -MD -MF CMakeFiles/foo.dir/B.cpp.o.ddi.d > CMakeFiles/foo.dir/B.cpp.o.ddi.tmp && mv
CMakeFiles/foo.dir/B.cpp.o.ddi.tmp CMakeFiles/foo.dir/B.cpp.o.ddi
[4/10] "/Users/hoffman/Work/My Builds/cmake-ninja/bin/cmake" -E cmake_ninja_dyndep --tdi=CMakeFiles/foo.dir/CXXDependInfo.json --lang=CXX --modmapfmt=clang
--dd=CMakeFiles/foo.dir/CXX.dd @CMakeFiles/foo.dir/CXX.dd.rsp
[5/10] "/Users/hoffman/Work/My Builds/cmake-ninja/bin/cmake" -E cmake_ninja_dyndep --tdi=CMakeFiles/hello.dir/CXXDependInfo.json --lang=CXX --modmapfmt=clang
--dd=CMakeFiles/hello.dir/CXX.dd @CMakeFiles/hello.dir/CXX.dd.rsp
[6/10] /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -MD -MT
CMakeFiles/foo.dir/B.cpp.o -MF CMakeFiles/foo.dir/B.cpp.o.d @CMakeFiles/foo.dir/B.cpp.o.modmap -o CMakeFiles/foo.dir/B.cpp.o -c /Users/hoffman/Work/modules/simple/B.cpp
[7/10] /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -MD -MT
CMakeFiles/foo.dir/A.cpp.o -MF CMakeFiles/foo.dir/A.cpp.o.d @CMakeFiles/foo.dir/A.cpp.o.modmap -o CMakeFiles/foo.dir/A.cpp.o -c /Users/hoffman/Work/modules/simple/A.cpp
[8/10] /Users/hoffman/Work/llvm/llvm-inst/bin/clang++   -std=gnu++20 -arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -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
/Users/hoffman/Work/modules/simple/main.cxx
[9/10] : && "/Users/hoffman/Work/My Builds/cmake-ninja/bin/cmake" -E rm -f libfoo.a && /usr/bin/ar qc libfoo.a  CMakeFiles/foo.dir/A.cpp.o CMakeFiles/foo.dir/B.cpp.o &&
/Users/hoffman/Work/llvm/llvm-inst/bin/llvm-ranlib libfoo.a && "/Users/hoffman/Work/My Builds/cmake-ninja/bin/cmake" -E touch libfoo.a && :
[10/10] : && /Users/hoffman/Work/llvm/llvm-inst/bin/clang++ -arch arm64 -isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX14.4.sdk -Wl,-search_paths_first
-Wl,-headerpad_max_install_names  CMakeFiles/hello.dir/main.cxx.o -o hello  libfoo.a && :
```

Installing Targets

```
install(TARGETS targets... [EXPORT <export-name>]
        [RUNTIME_DEPENDENCIES args...|RUNTIME_DEPENDENCY_SET <set-name>]
        [[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE|FILE_SET <set-name>|CXX_MODULES_BMI]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [NAMELINK_COMPONENT <component>]
         [OPTIONAL] [EXCLUDE_FROM_ALL]
         [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...]
        [INCLUDES DESTINATION [<dir> ...]]
        )
```

`CXX_MODULES_BMI`

New in version 3.28.

Any module files from C++ modules from `PUBLIC` sources in a file set of type `CXX_MODULES` will be installed to the given `DESTINATION`. All modules are placed directly in the destination as no directory structure is derived from the names of the modules. An empty `DESTINATION` may be used to suppress installing these files (for use in generic code).

Only pay for scanning if modules

CXX_SCAN_FOR_MODULES

New in version 3.28.

`CXX_SCAN_FOR_MODULES` is a boolean specifying whether CMake will scar odule de-pendencies. See also the **`CXX_SCAN_FOR_MODULES`** for per-source setting get-wide settings.

This property is initialized by the value of the **`CMAKE_CXX_SCAN_FOR_MODULES`** variable if it is set when a target is created.

When this property is set `ON` or unset, CMake will scan the target's `CXX` sources at build time and add module dependency information to the compile line as necessary. When this property is set `OFF`, CMake will not scan the target's `CXX` sources at build time.

Note that scanning is only performed if C++20 or higher is enabled for the target. Scanning for modules in the target's sources belonging to file sets of type `CXX_MODULES` is always performed.

Must be on for any file that has import or export

```
export(TARGETS <target>... [NAMESPACE <namespace>]
       [APPEND] FILE <filename> [EXPORT_LINK_INTERFACE_LIBRARIES]
       [CXX_MODULES_DIRECTORY <directory>])
```

Creates a file `<filename>` that may be included by outside projects to import targets named by `<target>...` from the current project's build tree. This is useful during cross-compiling to build utility executables that can run on the host platform in one project and then import them into another project being compiled for the target platform.

The file created by this command is specific to the build tree and should never be installed. See the **`install(EXPORT)`** command to export targets from an install tree.

The options are:

`NAMESPACE <namespace>`
: Prepend the `<namespace>` string to all target names written to the file.

`APPEND`
: Append to the file instead of overwriting it. This can be used to incrementally export multiple targets to the same file.

`EXPORT_LINK_INTERFACE_LIBRARIES`
: Include the contents of the properties named with the pattern `(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?` in the export, even when policy **`CMP0022`** is NEW. This is useful to support consumers using CMake versions older than 2.8.12.

`CXX_MODULES_DIRECTORY <directory>`
: *New in version 3.28.*

  Export C++ module properties to files under the given directory. Each file will be named according to the target's export name (without any namespace). These files will automatically be included from the export file.

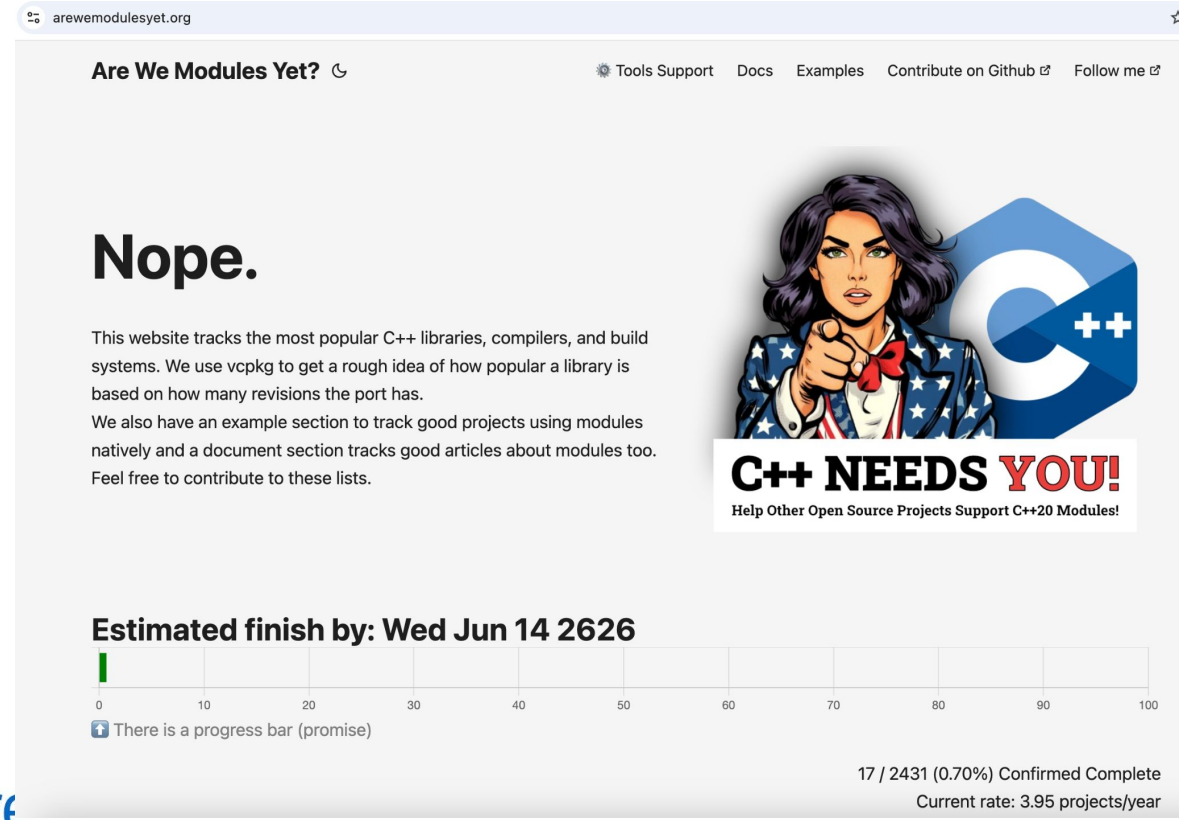
Anatomy of a named module

```
module;   // Global module fragment start
// If a module-unit has a global module fragment, then its first declaration must be
// module;. Then, only preprocessing directives can appear in the global module fragment.
// Then, a standard module declaration marks the end of the global module fragment and
// the start of the module content.
#define TURN_ON_FOO
#include <foo.h>

export module A;   // declare the module A after here, it's all modules

export void A() { // do A stuff }
```

Global module fragment

- NOTE: Any headers included here must be PUBLIC because the module will need to be used to create a BMI by consumers.

Are We Modules Yet?



List Of Libraries:

Status
- ❔ Default: No set. Help wanted for every lib that has this status!
- ✅ Full support
- ⚙️ Worked on
- ⚠ Reported
- ❌ Corresponding tracking issue was closed and no planned support anytime soon.
- 💀 Needs support old library/compiler/build tool to even start adding support for modules.
- 🤡 Maintainer does want to keep it oldschool. No plan to update anytime soon.

Help Wanted
- ✅ Yes
- ❌ No
- 💀 Does not allow outside contributions

Popularity based on vcpkg port update: Generated Date: 2 May 2024 Count parsed from this git commit.

Search for names..

☑✅ ☐❔ ☐⚙️ ☐⚠ ☐❌ ☐💀 ☐🤡 Status
☐✅ ☐❔ ☐❌ ☐💀 Help Wanted

Status	Help Wanted	Name	Import Statement	Popularity	Version	Min C++ Version	Modules Support Since	Tracking Issue
✅	❔	fmt	`import fmt;`	60	10.2.1	C++11	2022/10/11	🔗
✅	❔	glm	`import glm;`	27	1.0.1	C++11	2023/12/30	🔗
✅	❔	magic-enum	`import magic_enum;`	21	0.9.5	C++17	2024/5/9	🔗
✅	❔	tgui	`import tgui;`	16	1.1.0		2024/1/13	
✅	❔	boost-pfr	`import Boost.PFR;`	12	1.84.0		2024/08/01	🔗
✅	❔	ctre	`import ctre;`	11	3.8.1	C++17	2024/5/17	

✅		argparse	`import argparse;`	10	3.0	C++17	2023/10/15 🔗
✅		fastgltf	`import fastgltf;`	8	0.7.1	C++17	2024/5/15 🔗
✅		vulkan-hpp	`import vulkan_hpp;`	8	deprec...	C++11	2020/6/23 🔗
✅		mp-units	`import mp_units;`	6	2.1.0		2024/1/6 🔗
✅		vulkan-memory-allocator-hpp	`import vk_mem_alloc_hpp;`	3	3.0.1.1	C++20	2023/9/11 🔗
✅		async-simple	`import async_simple;`	2	1.3	C++20	2023/6/19
✅		flux	`import flux;`	1	Unkno...	C++20	2023/8/3
✅		vladimirshaleev-ipaddress	`import ipaddress;`	1	1.0.1		2024/6/2 🔗
✅		z4kn4fein-semver	`import semver;`	1	0.2.1	C++17	2024/8/21 🔗
✅		seastar	`import seastar;`			C++20	2023/3/24
✅		cpp-lazy	`import lz;`			C++11	2024/4/9

OK, so your telling me there's a chance

17 libraries with module support, AWESOME!

Let's build them

kitware

Get a new enough compiler

- Most Recent Visual Studio Preview
- Clang origin/main - be ready to build some C++

```
%cmake -DLLVM_ENABLE_PROJECTS=clang
-DCMAKE_INSTALL_PREFIX=/Users/hoffman/Work/llvm-main-inst
-DCMAKE_BUILD_TYPE=Release  -DLLVM_ENABLE_RUNTIMES="libcxx;libcxxabi;libc;libunwind"
-G Ninja ../llvm-project/llvm
%ninja
```

- Get CMake origin/master

kitware

Setup an env

```
export CC=/Users/hoffman/Work/llvm/llvm-inst/bin/clang
export CXX=/Users/hoffman/Work/llvm/llvm-inst/bin/clang++
export PATH=
"/Users/hoffman/Work/My Builds/cmake-ninja/bin":$PATH
export
CMAKE_INSTALL_PREFIX=/Users/hoffman/Work/modules/install
```

Adv

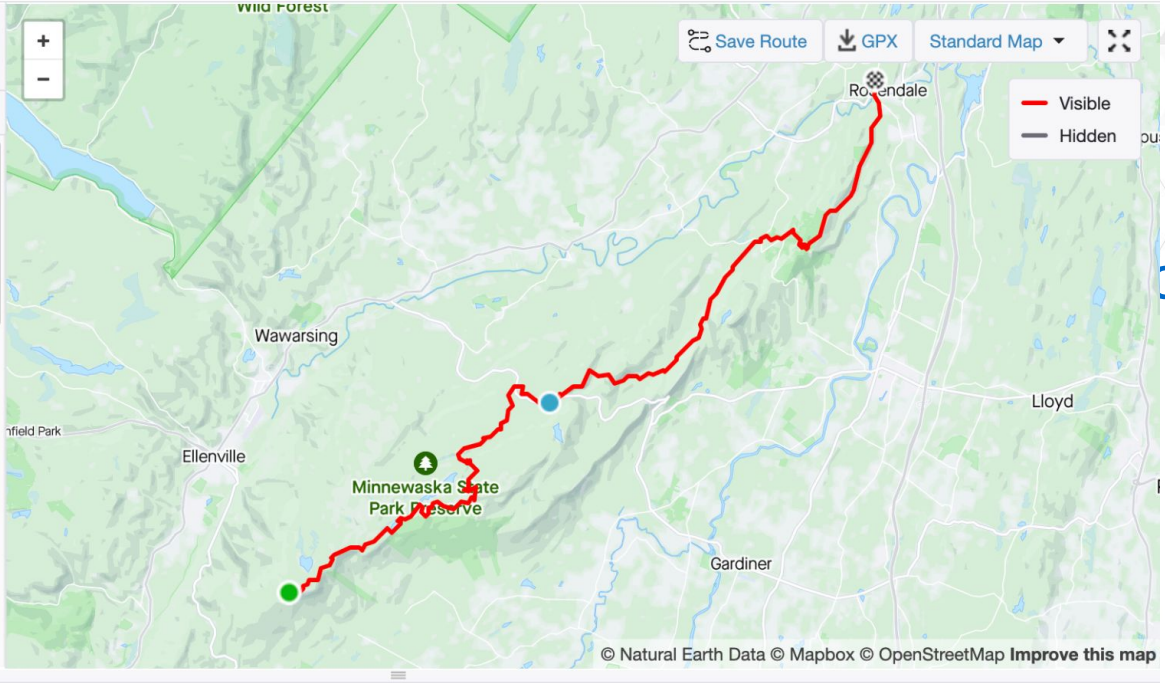
https:/

ons

kitv

Splits

Mile	Pace	GAP	Elev
1	10:02 /mi	8:30 /mi	220 ft
2	12:35 /mi	12:57 /mi	-149 ft
3	15:33 /mi	15:51 /mi	-306 ft
4	16:48 /mi	13:21 /mi	84 ft
5	14:48 /mi	12:55 /mi	61 ft
6	13:19 /mi	12:18 /mi	28 ft
7	15:10 /mi	13:27 /mi	-75 ft
8	16:35 /mi	12:43 /mi	227 ft
9	14:38 /mi	14:19 /mi	-156 ft
10	20:52 /mi	17:04 /mi	-183 ft
11	14:39 /mi	13:28 /mi	-47 ft
12	14:20 /mi	15:10 /mi	-203 ft



CTRE no install for modules :(

- Just add a hello executable into the CTRE project cmake file itself to test
- Didn't work
- Could not find the module ctre
- CXX_SCAN_FOR_MODULES was the culprit!

5 CMakeLists.txt

```
@@ -1,4 +1,4 @@
- cmake_minimum_required(VERSION 3.14.0)
+ cmake_minimum_required(VERSION 3.14...3.29)

if(CMAKE_VERSION VERSION_GREATER_EQUAL "3.29.20240416")
        set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
@@ -92,7 +92,7 @@ if(CTRE_MODULE)
```

hanickadot commented 2 days ago — Owner

why is there maximum version of 3.29?

mathstuf commented 2 days ago — Contributor, Author

That is not a "maximum version". It is a "maximum *tested* version". It is saying "CMake 3.14 is required, but the project has been tested against all new policy behaviors up to 3.29 and accepts the new behaviors".

```
std;`
LESS 23)
XX_STANDARD OR CTRE_CXX_STANDARD LESS 23)
TANDARD 23)

endif()
114 114
115 115         target_compile_features(${PROJECT_NAME} INTERFACE cxx_std_${CTRE_CXX_STANDARD})
    116 +       set_property(TARGET ${PROJECT_NAME} PROPERTY CXX_SCAN_FOR_MODULES 0)
116 117
117 118         install(TARGETS ${PROJECT_NAME} EXPORT ${PROJECT_NAME}-targets)
118 119 endif()
```

kitw

Let's get it to install

Install module rules #323

Merged **hanickadot** merged 3 commits into `hanickadot:main` from `mathstuf:install-module-rules` 32 minutes ago

Conversation 1 | Commits 3 | Checks 62 | Files changed 1

Changes from all commits | File filter | Conversations | Jump to | 0 / 1 files viewed | Review

26 CMakeLists.txt

```
@@ -38,7 +38,10 @@ if(CTRE_MODULE)
38 38                   add_library(${PROJECT_NAME})
39 39
40 40                   target_sources(${PROJECT_NAME} PUBLIC FILE_SET CXX_MODULES TYPE CXX_MODULES FILES ctre.cppm)
41    -                target_sources(${PROJECT_NAME} PUBLIC FILE_SET HEADERS TYPE HEADERS FILES
      41 +                target_sources(${PROJECT_NAME} PUBLIC FILE_SET HEADERS TYPE HEADERS
      42 +                    BASE_DIRS
      43 +                        "${CMAKE_CURRENT_SOURCE_DIR}/include"
      44 +                    FILES
42 45                           include/ctll.hpp
43 46                           include/ctre/functions.hpp
44 47                           include/ctre/utility.hpp
@@ -96,8 +99,11 @@ if(CTRE_MODULE)
96 99                           set(CTRE_CXX_STANDARD 23)
97 100                  endif()
98 101
99    -                target_include_directories(${PROJECT_NAME} PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
100 102                 target_compile_features(${PROJECT_NAME} PUBLIC cxx_std_${CTRE_CXX_STANDARD})
      103 +
      104 +                install(TARGETS ${PROJECT_NAME} EXPORT ${PROJECT_NAME}-targets
      105 +                        FILE_SET CXX_MODULES DESTINATION "${CMAKE_INSTALL_LIBDIR}/cxx/${PROJECT_NAME}"
      106 +                        FILE_SET HEADERS DESTINATION "include")
```

kitware

```
import std;
import ctre;

std::optional<std::string_view> extract_number(std::string_view s) noexcept {
      if (auto m = ctre::match<"[a-z]+([0-9]+)">(s)) {
        return m.get<1>().to_view();
    } else {
        return std::nullopt;
    }
}

int main()
{
  auto opt = extract_number("hello123");
  if (opt) {
    std::string s(*opt);
    std::cout << s << "\n";
  }
  return 0;
```

Hello modular CTRE

```
cmake_minimum_required(VERSION 3.29)
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
project(import_ctre)
find_package(ctre REQUIRED)
add_executable(ctre_hello ctre_hello.cpp)
target_link_libraries(ctre_hello PRIVATE ctre::ctre)
```

```
# Windows build with msvc
Z:\Users\hoffman\Work\modules\realworld\ctre_hello\vsb>.\ctre_hello.exe
123
# mac build with clang
hoffman@caprica b % ./ctre_hello
123
```

kitware

Clang and CMake have an issue with "import std"

CMake Error in CMakeLists.txt:

The "CXX_MODULE_STD" property on the target "ctre" requires that the "__CMAKE::CXX23" target exist, but it was not provided by the toolchain.

Reason:

`libc++.modules.json` resource does not exist

```
diff --git a/Modules/Compiler/Clang-CXX-CXXImportStd.cmake b/Modules/Compiler/Cl
ang-CXX-CXXImportStd.cmake
index f58f17ea65..e48c4a85fc 100644
--- a/Modules/Compiler/Clang-CXX-CXXImportStd.cmake
+++ b/Modules/Compiler/Clang-CXX-CXXImportStd.cmake
@@ -10,7 +10,7 @@ function (_cmake_cxx_import_std std variable)
       COMMAND
         "${CMAKE_CXX_COMPILER}"
         ${CMAKE_CXX_COMPILER_ID_ARG1}
-        -print-file-name=libc++.modules.json
+        -print-file-name=../../libc++.modules.json
```

HACK!!

C++ modules: 'import std;' with libc++ does not work

Open Issue created 4 months ago by **PinieP**

I'm using CMake 3.29.2040507 and a clang/libc++ 19.0 build

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.29.20240507-gf2b76d7)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED YES)

set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
set(CMAKE_CXX_MODULE_STD ON)

project(example LANGUAGES CXX)
add_executable(main main.cpp)
```

```
# main.cpp
```

Note: Same error someone else's fault

CMake Error in CMakeLists.txt:

The "CXX_MODULE_STD" property on the target "ctre" requires that the "__CMAKE::CXX23" target exist, but it was not provided by the toolchain.

Reason:

`libc++.modules.json` resource does not exist

```
cmake_minimum_required(VERSION 3.29)
#set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
project(import_ctre)
find_package(ctre REQUIRED)
add_executable(ctre_hello ctre_hello.cpp)
target_link_libraries(ctre_hello PRIVATE ctre::ctre)
```

```
cmake_minimum_required(VERSION 3.29)
project(import_ctre)
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")
find_package(ctre REQUIRED)
add_executable(ctre_hello ctre_hello.cpp)
target_link_libraries(ctre_hello PRIVATE ctre::ctre)
```

Where did the expression "winner winner chicken dinner" come from?

In the late 1920s (the Depression Era in the United States), a chicken dinner from a casino cafeteria cost just under $2—and $2 was the standard bet for many gamblers. So, **anyone who won a bet was effectively able to buy dinner that day**, hence the phrase “Winner winner chicken dinner.” Jun 3, 2024

✅	fmt	`import fmt;`	60	+11	2022/10/11	🔗
✅	vladimirshaleev-ipaddress	`import ipaddress;`	1		2024/6/2	🔗
✅	z4kn4fein-semver	`import semver;`	1	C++17	2024/8/21	🔗

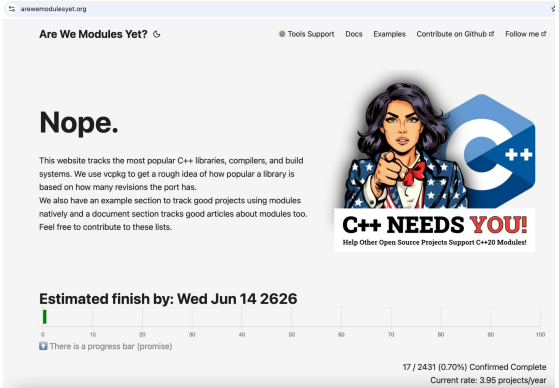
kitware

Other things I learned

- Treat your C++ like cattle not pets = Treat your CMake code like real code, have proper indentation and comments
- Be very careful with versions of things
  - a wrong version of cmake, ninja, compiler can result in a vomiting of not so useful errors at build time!

Don't Despair Code!

- import std will pave the way!



kitware

import std;

- A named module that is built around existing header files.
- Let's take a look under the hood.

kitware

"import std" model

- Named module for an existing library little change to existing code
- Just need to create a module file like std.cppm, but what goes in that file?

Clang - std.cppm

```
// WARNING, this entire header is generated by
// utils/generate_libcxx_cppm_in.py
// DO NOT MODIFY!

module;

#include <__config>

// The headers of Table 24: C++ library headers_[tab:headers.cpp]
// and the headers of Table 25: C++ headers for C library facilities_[tab:headers.cpp.c]
#include <algorithm>
#include <any>
#include <array>
#if !defined(_LIBCPP_HAS_NO_ATOMIC_HEADER)
#  include <atomic>
#endif
#include <barrier>
#include <bit>
#include <bitset>
#include <cassert>
#include <cctype>
#include <cerrno>
#include <cfenv>
#include <cfloat>
#include <charconv>
#include <chrono>
#include <cinttypes>
#include <climits>
#if !defined(_LIBCPP_HAS_NO_LOCALIZATION)
#  include <clocale>
#endif
```

kitware

Clang - std.cppm

```
export module std;


#include "std/algorithm.inc"
#include "std/any.inc"
#include "std/array.inc"
#include "std/atomic.inc"
#include "std/barrier.inc"
#include "std/bit.inc"
#include "std/bitset.inc"
#include "std/cassert.inc"
#include "std/cctype.inc"
#include "std/cerrno.inc"
#include "std/cfenv.inc"
#include "std/cfloat.inc"
#include "std/charconv.inc"
#include "std/chrono.inc"
#include "std/cinttypes.inc"
#include "std/climits.inc"
#include "std/clocale.inc"
#include "std/cmath.inc"
#include "std/codecvt.inc"
#include "std/compare.inc"
#include "std/complex.inc"
#include "std/concepts.inc"
#include "std/condition_variable.inc"
#include "std/coroutine.inc"
#include "std/csetjmp.inc"
#include "std/csignal.inc"
#include "std/cstdarg.inc"
#include "std/cstddef.inc"
#include "std/cstdint.inc"
#include "std/cstdio.inc"
```

kitware

Clang - std.cppm

algorithm.inc

```
// -*- C++ -*-
//===----------------------------------------------------------------------===//
//
// Part of the LLVM Project, under the Apache License v2.0 with LLVM Exceptions.
// See https://llvm.org/LICENSE.txt for license information.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
//
//===----------------------------------------------------------------------===//

export namespace std {
  namespace ranges {
    // [algorithms.results], algorithm result types
    using std::ranges::in_found_result;
    using std::ranges::in_fun_result;
    using std::ranges::in_in_out_result;
    using std::ranges::in_in_result;
    using std::ranges::in_out_out_result;
    using std::ranges::in_out_result;
#if _LIBCPP_STD_VER >= 23
    using std::ranges::in_value_result;
#endif
    using std::ranges::min_max_result;
    // using std::ranges::out_value_result;
  } // namespace ranges

  // [alg.nonmodifying], non-modifying sequence operations
  // [alg.all.of], all of
  using std::all_of;
  namespace ranges {
    using std::ranges::all_of;
  }

  // [alg.any.of], any of
  using std::any_of;
  namespace ranges {
    using std::ranges::any_of;
  }
```

kitware

MSVC - std.ixx

```
// Copyright (c) Microsoft Corporation.
// SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception

// In a module-file, the optional `module;` must appear first; see [cpp.pre].
module;

// This named module expects to be built with classic headers, not header units.
#define _BUILD_STD_MODULE

// The subset of "C headers" [tab:c.headers] corresponding to
// the "C++ headers for C library facilities" [tab:headers.cpp.c]
#include <assert.h>
#include <ctype.h>
#include <errno.h>
#include <fenv.h>
#include <float.h>
#include <inttypes.h>
#include <limits.h>
#include <locale.h>
#include <math.h>
#include <setjmp.h>
#include <signal.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <uchar.h>
#include <wchar.h>
#include <wctype.h>
```

kitware

MSVC - std.ixx

```
export module std;

#pragma warning(push)
#pragma warning(disable : 5244) // '#include <meow>' in the purview of module 's
td' appears erroneous.

// "C++ library headers"
#include <algorithm>
#if _HAS_STATIC_RTTI
#include <any>
#endif // _HAS_STATIC_RT
#include <array>
#include <atomic>
#include <barrier>
#include <bit>
#include <bitset>
#include <charconv>
#include <chrono>
#include <codecvt>
#include <compare>
#include <complex>
#include <concepts>
#include <condition_vari
#include <coroutine>
#include <deque>
#include <exception>
#include <execution>
```

??

MSVC - std.ixx

```
_EXPORT_STD template <class _Ty, class _Alloc = allocator<_Ty>>
class vector { // varying size array of values
private:
    template <class>
    friend class _Vb_val;
    friend _Tidy_guard<vector>;

    using _Alty         = _Rebind_alloc_t<_Alloc, _Ty>;
    using _Alty_traits = allocator_traits<_Alty>;

public:
    static_assert(!_ENFORCE_MATCHING_ALLOCATORS || is_same_v<_Ty, typename _Alloc::value_type>,
        _MISMATCHED_ALLOCATOR_MESSAGE("vector<T, Allocator>", "T"));
    static_assert(is_object_v<_Ty>, "The C++ Standard forbids containers of non-object types "
                                    "because of [container.requirements].");

    using value_type      = _Ty;
    using allocator_type  = _Alloc;
    using pointer         = typename _Alty_traits::pointer;
    using const_pointer   = typename _Alty_traits::const_pointer;
    using reference       = _Ty&;
    using const_reference = const _Ty&;
    using size_type       = typename _Alty_traits::size_type;
    using difference_type = typename _Alty_traits::difference_type;
```

MSVC- std.ixx

```
export template <class _Ty, class _Alloc = allocator<_Ty>>
class vector {
private:
    template <class>
    friend class _Vb_val;
    friend _Tidy_guard<vector>;

    using _Alty        = _Rebind_alloc_t<_Alloc, _Ty>;
    using _Alty_traits = allocator_traits<_Alty>;

public:
    static_assert(!1 || is_same_v<_Ty, typename _Alloc::value_type>,
        "vector<T, Allocator>" " requires that Allocator's value_type match " "T" " (See N4950 [container.a
lloc.reqmts]/5)" " Either fix the allocator value_type or define _ENFORCE_MATCHING_ALLOCATORS=0" " to suppr
ess this error.");
    static_assert(is_object_v<_Ty>, "The C++ Standard forbids containers of non-object types "
                                    "because of [container.requirements].");

    using value_type      = _Ty;
    using allocator_type  = _Alloc;
    using pointer         = typename _Alty_traits::pointer;
    using const_pointer   = typename _Alty_traits::const_pointer;
    using reference       = _Ty&;
    using const_reference = const _Ty&;
    using size_type       = typename _Alty_traits::size_type;
    using difference_type = typename _Alty_traits::difference_type;
```

kitware.com/import-std-in-cmake-3-30/

kitware

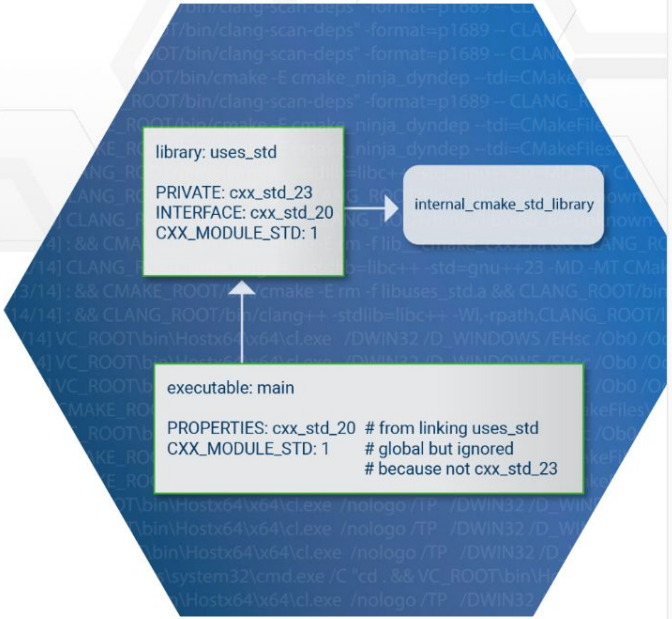
about solutions news careers contact EU Enter Keyword

news › blog › post

import std in CMake 3.30

April 18, 2024 Bill Hoffman and Ben Boeckel

A New Experiment!



Since C++ named modules left the experimental state in CMake (see import CMake; the Experiment is Over!), CMake development has continued to add more support for modules. The most recent development is the experimental support of import std;. C++23 introduced support for APIs to be available via the named module std. This means that `import std;` is a valid way to get access to all non-macro C++ APIs instead of including headers one-by-one. This is great news, however to use modules the build system needs to be involved to create the BMI files before the import can happen.

The CMake support for named modules included the ability to export targets for later import. The solution leverages that support with toolchains declaring a target to represent the standard library for each standard. This target is used inside of CMake's own logic and added to projects automatically; projects do not need to mention the targets in project code at all. When CMake encounters a target which has at least C++23 enabled that also supports C++ modules[1], it will query the `CXX_MODULE_STD` property on the target (which may contain generator expressions[2]). This property is initialized by the `CMAKE_CXX_MODULE_STD` variable, if defined. If it is unset, the current behavior is to not use the target. The internal target is not

kitware

CMake import std

```
# CMake 3.30 is required for C++23 `import std` support; we use 3.29.20240416
# here so that in-development versions satisfy it.
cmake_minimum_required(VERSION 3.29.20240416 FATAL_ERROR)

# Set experimental flag to enable `import std` support from CMake.
# This must be enabled before C++ language support.
set(CMAKE_EXPERIMENTAL_CXX_IMPORT_STD
  # This specific value changes as experimental support evolves. See
  # `Help/dev/experimental.rst` in the CMake source corresponding to
  # your CMake build for the exact value to use.
  "0e5b6991-d74f-4b3d-a41c-cf096e0b2508")

# C++ needs to be enabled.
project(import_std LANGUAGES CXX)
```

kitware

```
# Tell CMake that we explicitly want `import std`. This will initialize the
# property on all targets declared after this to 1
set(CMAKE_CXX_MODULE_STD 1)

# Make a library.
add_library(uses_std STATIC)
# Add sources.
target_sources(uses_std
  PRIVATE
    uses_std.cxx)
# Tell CMake we're using C++23 but only C++20 is needed to consume it.
target_compile_features(uses_std
  PRIVATE   cxx_std_23
  INTERFACE cxx_std_20)

# Make an executable.
add_executable(main)
# Note that this source is *not* allowed to `import std` as it ends up
# with only C++20 support due to the `uses_std` INTERFACE requirements.
target_sources(main
  PRIVATE
    main.cxx)
target_link_libraries(main PRIVATE uses_std)
```

With the following `uses_std.cxx` source:

```
import std;

void hello_world(std::string const& name)
{
  std::cout << "Hello World! My name is " << name << std::endl;
}
```

And the executable's source:

```
#include <string>

void hello_world(std::string const& name);

int main(int argc, char* argv[])
{
  hello_world(argv[0] ? argv[0] : "Voldemort?");
  return 0;
}
```

CMake import std

```
[1/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=g
[2/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../li
[3/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=g
[4/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/main.dir/CXXDependInfo.json --lang=C
[5/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../li
[6/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/__cmake_cxx23.dir/CXXDependInfo.jso
[7/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/uses_std.dir/CXXDependInfo.json --la
[8/14] CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=gnu++20 -MD -MT CMakeFiles/main.dir/main.cxx.o -MF
[9/14] CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../lib/x86_64-unknown-linux-gnu/../../share/libc++/v1 -
[10/14] CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../lib/x86_64-unknown-linux-gnu/../../share/libc++/v1
[11/14] : && CMAKE_ROOT/bin/cmake -E rm -f lib__cmake_cxx23.a && CLANG_ROOT/bin/llvm-ar qc lib__cmake_c
[12/14] CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=gnu++20 -MD -MT CMakeFiles/uses_std.dir/uses_std.c
[13/14] : && CMAKE_ROOT/bin/cmake -E rm -f libuses_std.a && CLANG_ROOT/bin/llvm-ar qc libuses_std.a  CM
[14/14] : && CLANG_ROOT/bin/clang++ -stdlib=libc++ -Wl,-rpath,CLANG_ROOT/lib/x86_64-unknown-linux-gnu C
```

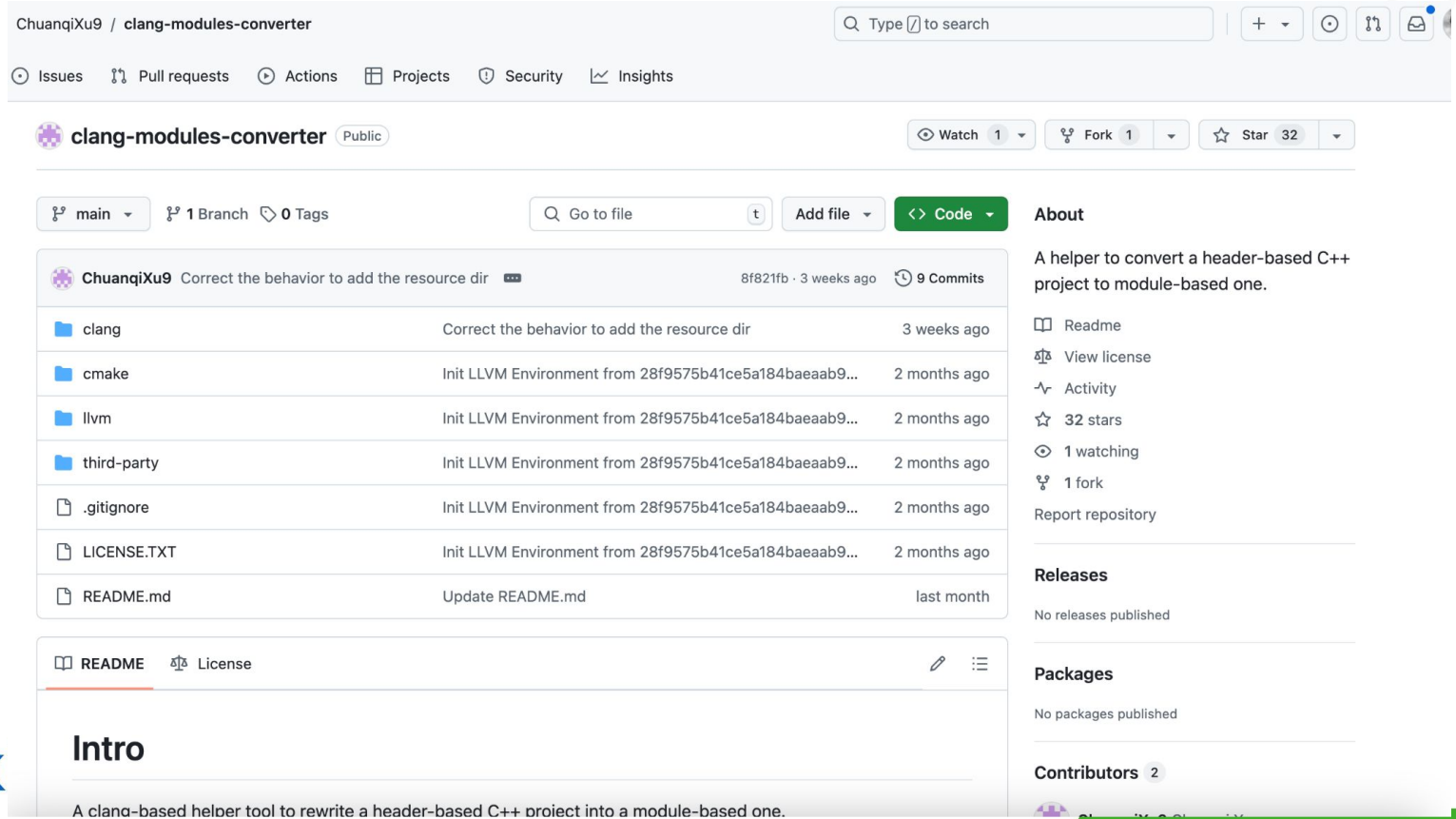
kitware

All I am saying is give modules a chance

- Try "import std"
- Create a named module for your project with install rules
- Be prepared to build compilers, cmake, and other tools and install previews.
- Be prepared for other errors from the bleeding edge compilers you will be using

Resources

https://github.com/ChuanqiXu9/clang-modules-converter

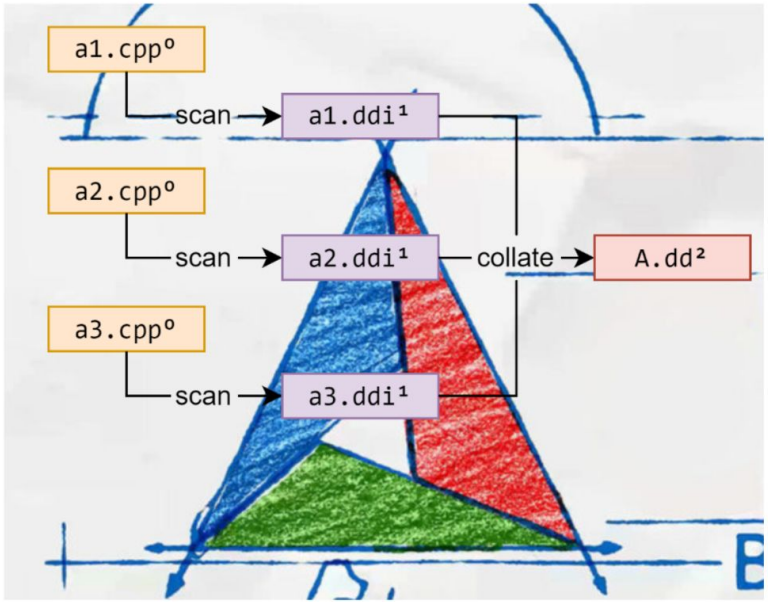


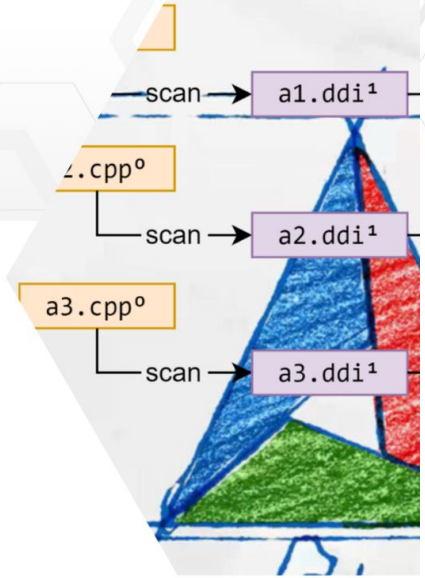
kitware.com/import-cmake-the-experiment-is-over/

news › blog › post

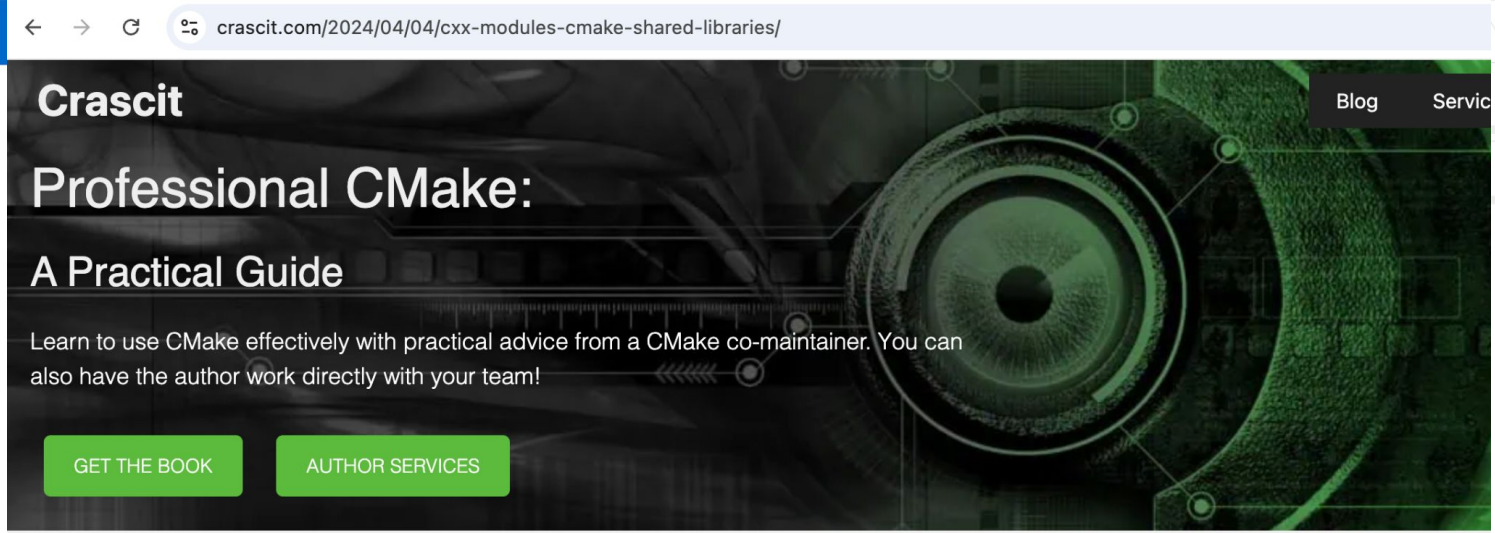
import CMake; the Experiment is Over!

October 18, 2023 Bill Hoffman, Brad King and Ben Boeckel





At last the experiment is over, and CMake 3.28 has official support for C++ 20 named modules enabled without having to set



C++20 Modules, CMake, And Shared Libraries

April 4, 2024 by Craig Scott

CMake 3.28 was the first version to officially support C++20 modules. Tutorials and examples understandably tend to focus on the fairly simple scenario of building a basic executable, perhaps also adding in a static library. The import CMake; the Experiment is Over! blog article from Kitware is perhaps one of the best known, and it covers exactly these things. And while these are important first steps, stopping there sells the reader short. The real fun starts when building, installing, and consuming shared libraries. This quickly exposes current limitations of toolchains and the build system, and it also highlights common misconceptions about what modules provide.

Estimated reading time: 13 minutes

links for resources

- https://gitlab.kitware.com/cmake/cmake/-/tree/master/Tests/RunCMake/CXXModules/examples
- https://www.kitware.com/import-std-in-cmake-3-30/
- https://www.kitware.com/import-cmake-the-experiment-is-over/
- https://github.com/ChuanqiXu9/clang-modules-converter
- https://crascit.com/2024/04/04/cxx-modules-cmake-shared-libraries/

Special Thanks

Bret Brown

Daniel Ruoso

Ben Boeckel

Chuanqi Xu

Bloomberg

Engineering

kitware

Thank You

- bill.hoffman@kitware.com
- Read “how to write a CMake buildsystem”
  - https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html/Explore the CMake documentation
- Explore the CMake documentation