Build 'Em all with CMake

By Alexy Pellegrini
About me

- Kitware Europe for 2+ years
- Kitware CMake trainer
- C++ dev
- Graphics programming
- Windows user
- Working on an LLVM backend for a VLIW processor designed by a friend
Kitware

Delivering Innovation
230 employees Worldwide
6 offices across USA/Europe

Software development
Based on open source tools
300+ active projects worldwide

Sustained Growth
Since creation of the company
100% employee-owned

65% staff with PhD or Master
High Level customer expertise

Revenue 2020
$39M consolidated

20+ years of expertise
Kitware USA, 1998
Kitware Europe, 2010

Leader in AI & scientific open source solutions
Customers / Various fields of application

**Academics**
- 70+ academic institutions worldwide

**Government agencies**
- 50+ government agencies and national laboratories

**Commercial companies**
- Over 500 commercial customers

**Medical**
- Image processing, multimodal visualization, image registration & segmentation, assisted surgery, custom software...

**Energy**
- HPC, in-situ simulation, scientific visualisation, particle flow, fluid mechanics, ground exploration...

**Intelligence**
- Scene analysis, big data analysis, scientific visualization, flow analysis...
Areas of expertise / Built on open source

Computer Vision

Data and Analytics

Scientific Computing

Medical Computing

Software Solutions
Open Source Benefits / Shifting Power

**Source code ownership**
- Source code ownership
- Integration with commercial software solutions

**Cost effectiveness**
- No license fee
- No vendor lock-in
- Shared maintenance costs

**Flexibility and Agility**
- Continuous development
- Up to date with new technologies
- Ability to customize and fix

**Security**
- Robust software and libraries
- Transparency
- Community effort
- Open Innovation mitigates risk
Kitware / Services

- Training
- Support
- Development
- Grant Collaboration
Why CMake
What is CMake?

- CMake is the cross-platform, open-source build system generator that lets you use the native development tools you love the most.

- It’s a build system generator

- It takes plain text files as input that describe your project and produces project files or make files for use with a wide variety of native development tools.

- Family of Software Development Tools
  - Build = CMake
  - Test = CTest/CDash
  - Package = CPack
CMake is the most popular C++ build tool at 55%

- Bryce Adelstein Lelbach, the chair of Standard C++ Library Evolution group, in his talk “What Belongs In The C++ Standard Library?” at C++Now in 2022, stated that **we actually have a standard build system! It’s CMake.**

### Job openings requiring CMake experience, June, 2022:
- Indeed.com: 900 jobs at Tesla Motors, DCS Corp, Mindsource, Quanergy, ...
- LinkedIn.com: >600 jobs at Samsung, Johnson Controls, Apple, Uber, Toyota, Microsoft ...

Jetbrains study in 2018

Jetbrains study 2017-2021
C++ modules

include vs import
The classic approach:
- **Header files**: declarations, template/inline code
- **Source files**: definitions
Example: foo.hpp and foo.cpp

// foo.hpp
#ifndef FOO_HPP
#define FOO_HPP
int foo(int i);
#endif

// foo.cpp
int foo(int i) {
    return i * 42;
}
Example: foo usage

// main.cpp
#include "foo.hpp"
int main() {
    return foo(4);
}

// main.cpp
int foo(int i);
int main() {
    return foo(4);
}
## File types of classic approach

<table>
<thead>
<tr>
<th>File</th>
<th>Example</th>
<th>Artifact</th>
<th>Notes</th>
</tr>
</thead>
</table>
| Headers (.hpp)    | `#ifndef X`  
                    `#define X`                          
                    ...                                      
                    `#endif`                                | (None)       | Never built, only copied into translation units using `#include`    |
| Source (.cpp)     | `#include "x.hpp"`  
                    ...                                      | Object file (.obj)     | Translation Units                                                     |
Issues with headers: Textual inclusion

- Increase compile-time (headers parsed multiple times)
  - Reduce as much as possible headers content
- No encapsulation, preprocessor leaks...
  - PIMPL pattern, avoid defines in headers, impl namespace
- `#includes` order matters
  - May break randomly
C++ modules (since C++20): include vs import

- Textual inclusion is replaced with semantic import
- Only exported symbols are visible!
  - No macro leak, no need for “impl” namespace...
- Header-Source replaced by:
  - 1: “Module Interface Unit”
  - N >= 0: “Module Implementation Unit”
// foo.cppm

export module foo;

export int foo(int i) {
    return i * 42;
}

Example: foo usage

// main.cpp
import foo;
int main() {
    return foo(4);
}

# File types of modules

<table>
<thead>
<tr>
<th>File</th>
<th>Example</th>
<th>Artifact</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module interface unit (.cppm)</td>
<td><code>export module x;</code></td>
<td>Built Module Interface (.pcm) Object file (.obj)</td>
<td>One per module</td>
</tr>
<tr>
<td>Module implementation unit (.cppm)</td>
<td><code>module x;</code></td>
<td>Object file (.obj)</td>
<td>Optional, contains definitions</td>
</tr>
<tr>
<td>Non-module unit (.cpp)</td>
<td><code>import x;</code></td>
<td>Object file (.obj)</td>
<td>“Classic” Translation Units</td>
</tr>
</tbody>
</table>
Built Module Interface

The artifact created by a compiler to represent a module unit or header unit. The format [...] is implementation specific and holds C++ entities, which can be represented in the form of compiler specific data structures (e.g. ASTs), machine code or any intermediate representation chosen by the implementer.

File extension: .pcm (Clang) | .gcm (GCC) | .ifc (MSVC)
In short:

- `import foo` looks for foo’s BMI (e.g. `foo.pcm`)
- This file contains the **module definition**
Issues with headers solved by modules

- Increase compile-time (headers parsed multiple times)
  - Prebuilt representation used directly!
- No encapsulation, preprocessor leaks...
  - Explicit export, preprocessor is local to module units!
- `#includes` order matters
  - Imports order does not matter!
New issues created by modules

- **Build order of modules units matters**
  - Need the “BMI” build artifact to import a module

- **Build parallelism is lower**
  - Dependencies are stronger (per-file)
  - Mitigated by the fact that each translation unit is faster
Other features

- **Partition units**
  - Enable splitting modules in multiple files

- **Header units (not supported by CMake, yet)**
  - Translation units synthesized from headers
  - `import <header>` don’t have access to macros defined before import declaration

- **Global module fragment**
  - Fragment where we can use classic includes in modules
Other features (example)

```cpp
module; // global module fragment
#define NOMINMAX
#include <Windows.h> // have access to NOMINMAX
export module foo:math; // partition
import <algorithm>; // private header unit
export int min(int a, int b) {
    return std::min(a, b); // OK
}
```
Building modules
LLVM support
Of C++ modules
Clang Module Support

Main module proposal

Fixes and clarifications about parsing, linkage, semantics, interactions with preprocessor...
Clang Scan Deps

- Command line tool to scan module dependencies without full tokenizer for faster scan
- Added in LLVM 16
- JSON format defined by P1689R5
CMake concepts

Small reminders
Usage requirements (Modern CMake)

- Root
  - Directory
    - Executable
    - Library B
    - Library A
  - Directory
    - Executable
    - Library B
    - Library A

Root
- Executable
- Library A
- Library B

Executable
- Library B
- Library A

Library A
- Library B
Usage requirements (Modern CMake)

PRIVATE: Only *this* target will use it

INTERFACE: Only *consuming targets* use it

PUBLIC: PRIVATE + INTERFACE

$<BUILD_INTERFACE>: When this target is being built

$<INSTALL_INTERFACE>: After this target has been installed

*Consuming target*: `target_link_libraries`
**File sets (target_sources)**

```cpp
add_library(foo STATIC)
target_sources(foo PUBLIC
    FILE_SET name
    TYPE CXX_MODULES
    FILES files...
)
```
Compile features

```cpp
set(CMAKE_CXX_STANDARD 20)
add_library(foo STATIC)
add_library(foo STATIC)
target_compile_features(foo PUBLIC cxx_std_20)
```
CMake support
Using the Ninja build system
Building modules with CMake (wrong way)

# x.cppm imports y.cppm

add_library(foo STATIC y.cppm x.cppm)

target_compile_features(foo PUBLIC cxx_std_20)
Building modules with CMake

Build may fail due to missing dependency!

You can start the build multiple times until it works :)
Building modules with CMake (good way)

```
add_library(foo STATIC)
target_compile_features(foo PUBLIC cxx_std_20)
target_sources(foo PUBLIC
   FILE_SET modules TYPE CXX_MODULES FILES
   y.cppm x.cppm)
```
Building modules with CMake

For each target, scan module units dependencies.

Then collate them into a single, per-target, file.
Building modules with CMake

Build system use this file to know the right build order

```
build-target dependence

x.cppm -> x.obj
y.cppm -> y.obj
x.obj import y.pcm

tgt.lib
```

Order:
- First: y.cppm
- Second: x.cppm

Linking order:
- tgt.lib
### Build output example

<table>
<thead>
<tr>
<th>Step</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>[1/6] Scanning y.cppm for CXX dependencies</td>
</tr>
<tr>
<td></td>
<td>[2/6] Scanning x.cppm for CXX dependencies</td>
</tr>
<tr>
<td>Build</td>
<td>[4/6] Building CXX object y.cppm.obj</td>
</tr>
<tr>
<td></td>
<td>[5/6] Building CXX object x.cppm.obj</td>
</tr>
<tr>
<td>Link</td>
<td>[6/6] Linking CXX static library foo.lib</td>
</tr>
</tbody>
</table>
Build output example (verbose)

Scan
[1/6] clang-scan-deps -format=p1689 -- clang -O0 -std=c++20 y.cppm -c -o CMakeFiles\foo.dir\y.cppm.obj -MT CMakeFiles\foo.dir\y.cppm.obj.dii -MD -MF CMakeFiles\foo.dir\y.cppm.obj.dii.d > CMakeFiles\foo.dir\y.cppm.obj.dii

[2/6] clang-scan-deps -format=p1689 -- clang -O0 -std=c++20 x.cppm -c -o CMakeFiles\foo.dir\x.cppm.obj -MT CMakeFiles\foo.dir\x.cppm.obj.dii -MD -MF CMakeFiles\foo.dir\x.cppm.obj.dii.d > CMakeFiles\foo.dir\x.cppm.obj.dii

Collate
[3/6] 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

Build
[4/6] clang -O0 -std=c++20 -MD -MT CMakeFiles/\foo.dir\y.cppm.obj -MF CMakeFiles/\foo.dir\y.cppm.obj.d @CMakeFiles/\foo.dir\y.cppm.obj.modmap -o CMakeFiles/\foo.dir/\y.cppm.obj -c C:/dev/eurollvm/\y.cppm

[5/6] clang -O0 -std=c++20 -MD -MT CMakeFiles/\foo.dir\x.cppm.obj -MF CMakeFiles/\foo.dir\x.cppm.obj.d @CMakeFiles/\foo.dir\x.cppm.obj.modmap -o CMakeFiles/\foo.dir\x.cppm.obj -c C:/dev/eurollvm/\x.cppm

Link
[6/6] llvm-ar qc foo.lib CMakeFiles/\foo.dir\y.cppm.obj CMakeFiles/\foo.dir/\x.cppm.obj
Importing modules?
Exporting modules

```bash
install(TARGETS foo
  EXPORT footargets
  FILE_SET modules DESTINATION include
)
```
Build output (importing foo in another project)

1/6 Scanning foo.cppm for CXX dependencies
2/6 Generating CXX dyndep file foo.dir/CXX.dd
3/6 Scanning main.cpp for CXX dependencies
4/6 Generating CXX dyndep file test.dir/CXX.dd
5/7 Building CXX object foo.dir/foo.bmi
6/7 Building CXX object test.dir/main.cpp.obj
7/7 Linking CXX executable test.exe
Build output (importing foo in another project)

The module interface unit is precompiled once to generate the BMI

```
clang++ -O0 -std=gnu++20 --precompile [...]
-o foo.dir/foo.bmi -c .../include/foo.cppm
```

import foo -> foo.bmi

Link against prebuilt foo.lib/a
Questions?

- Kitware blog on CMake support of modules
- P2473R: Distributing C++ Module Libraries
- CMake Header Units support
- CMake 3.28 Release Notes