Beyond Compilation Databases to Support C++ Modules:

Build Databases

BEN BOECKEL

Cppcon
The C++ Conference
2024
September 15 - 20

Who am I?

- 15 years working on build systems and software process
- CMake developer
  - Designing and implementing features, performance improvements
  - Maintaining and improving build systems which use CMake
  - C++ modules support
- Working with the community to improve building software
  - ISO standards
    - Dependency format
  - Pushing forward modules support in other build systems
    - `xmake`
    - `bazel`

Outline

1. What are compilation databases?
2. How do modules change the status quo?
3. Build databases

Compilation Databases: Overview

- JSON document
- Array of JSON objects
- Each object describes a single command
  - Working directory
  - Input file
  - Output file (optional)
  - Arguments (list of strings) or command (single string, shell escaped)

Today: Compilation Databases

- Specified by the Clang project
  - https://clang.llvm.org/docs/JSONCompilationDatabase.html
- Widely used and available
  - Generated by build systems
  - Used by static analysis to know how to analyze sources
  - IDEs to understand how a source is used (e.g., highlighting the right side of an `#if` block)

Kitware

Compilation Databases: Example

```
{
  "directory": "/path/to/build",
  "command": "/usr/lib64/ccache/c++ FLAG_SOUP 🥣 -o
    Source/CMakeFiles/CMakeLib.dir/cmMakefile.cxx.o -c
    /path/to/source/Source/cmMakefile.cxx",
  "file": "/path/to/source/Source/cmMakefile.cxx",
  "output": "Source/CMakeFiles/CMakeLib.dir/cmMakefile.cxx.o"
}
```

Compilation Databases: Generation and Usage

- Generated by the build system (CMake, Meson, etc.)
  - Can also be generated by ninja itself with `ninja -t compdb`
  - Other tools like `bear` can extract a database from a build via tracing
- Generally available at the same time as the build instructions (`Makefile`, `build.ninja`, etc.)
- Use it as soon as you have a configured build tree
  - Some exceptions apply

Compilation Databases: Limitations

- Without the output file, source files can be ambiguous
  - Multi-config builds compile the same source with different flags
  - Source files can be reused between targets with different flags
- Even with output files, given a source file, which flags should be used?
  - Need a config and/or target selection mechanism
  - Visual Studio does the configuration selection natively
    - Can know what target the file was opened “for” based on the associated project file
- Command-based values are “shell-escaped”
  - Which shell?
  - Bourne shell is a safe assumption on Unix
  - Powershell or cmd is very relevant on Windows

Compilation Databases: Limitations 2

- Extensibility
  - No reserved field names, so adding new fields can conflict
- Portability
  - Flag meanings depend on the compiler in use (vendor and version)
- Build graph is unknown
  - Generated headers?
  - Generated *sources*?
  - Generated input files (`-include`, `-fmodule-mapper=`, response files)
- C++ modules…

C++ Modules on the Scene

- Modules complicate C++ compilation
  - Surprise! 🎉
  - Basically inherit the Fortran 90 modules compilation model
    - Importing a module requires files generated during compilation of another TU (the “BMI”)
      - BMI: built module interface, binary module interface
      - Also “CMI” for “compiled module interface”
    - Similarities
      - BMIs are compiler-specific
      - Lookup based on in-source identification (filenames are meaningless)
    - Differences in details
      - Fortran supports “submodules” and exporting multiple modules per TU
      - C++ has “partitions” and flags need to agree between the BMI and importer

A Brief History of Building Fortran Modules

- After their initial introduction in Fortran 90, official documentation was "run a parallel build until dependencies are satisfied"
  - Good luck with cycles or modules that aren't provided by any source
- `makef90dep` came along and generated the `Makefile` bits needed to order compilations
- Vendored into CMake for its `Unix Makefiles` generator
- `dyndep` support added to `ninja`, merged into 1.10 (released Jan 2020)

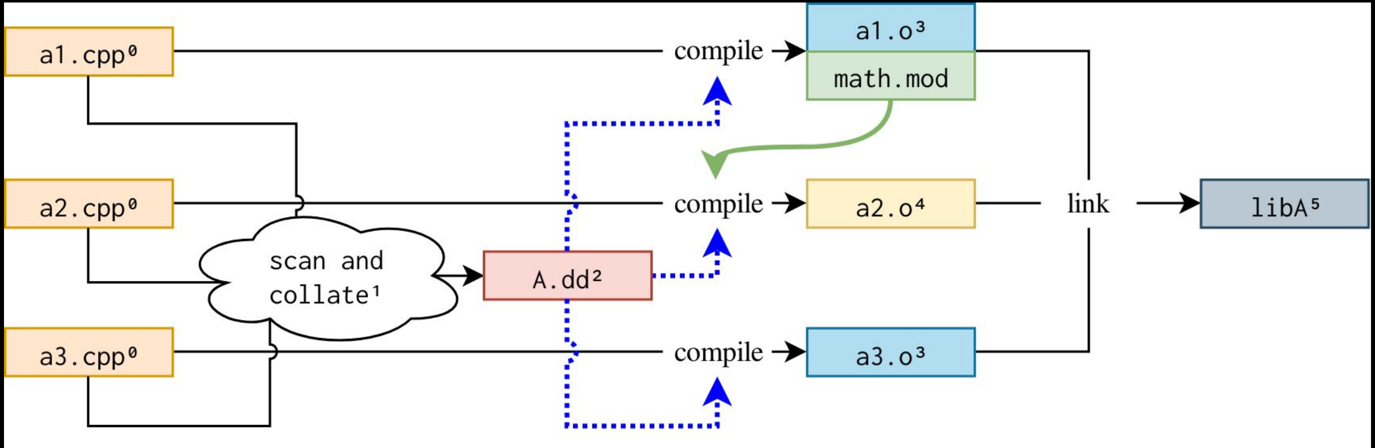
Kitware

C++ Modules: Example Target

```
add_library(A)
target_sources(A
  PRIVATE
    a2.cpp a3.cpp
  PRIVATE
    FILE_SET CXX_MODULES
    FILES
      a1.cpp)
target_compile_features(A PUBLIC cxx_std_20)
```

C++ Modules on the Scene: Compilation Strategy

- CMake uses the “scanning” approach to build Fortran and C++ modules



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

See: https://wg21.link/p1689r5 “Format for describing dependencies of source files”

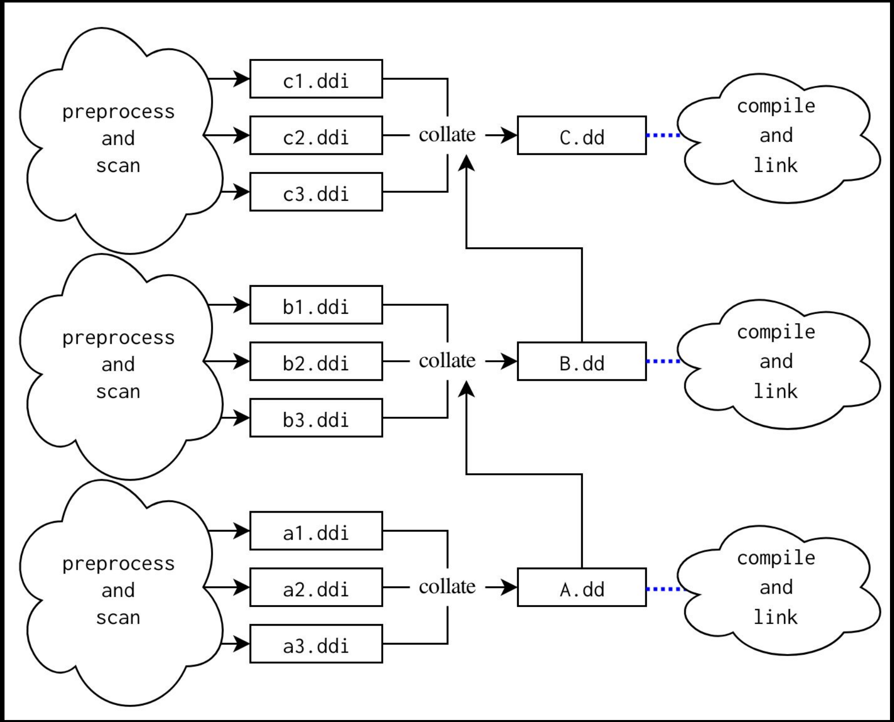
C++ Modules: Example Project

```cmake
add_library(A)
# add sources to A

add_library(B)
# add sources to B
target_link_libraries(B PRIVATE A)

add_library(C)
# add sources to C
target_link_libraries(C PRIVATE B)
```

C++ Modules on the Scene: Compilation Strategy 2



C++ Modules on the Scene: Whither Compile DB?

- What information is missing to accurately represent this new approach?
  - ```
    {
      "directory": "/path/to/build",
      "command": "/usr/lib64/ccache/c++ …",
      "file": "/path/to/source/Source/cmMakefile.cxx",
      "output": "Source/CMakeFiles/CMakeLib.dir/cmMakefile.cxx.o"
    }
    ```
- Let’s brainstorm! 🧠🌩️
    ```

C++ Modules on the Scene: Whither Compile DB? 2

- What is missing?
  - Ordering between commands
  - Information about module usage
    - Currently CMake “smuggles” through module mapper files (basically response files)
    - These files are referenced by but not necessarily present with the compilation database
  - Visibility of modules
    - Just because we have `A.mod` doesn’t mean anything can use it
      - Might be private to its target
      - Might not be linked by the target owning the source in question
  - Flag compatibility questions
    - `-std=c++26` in importer P and `-std=c++23` in importer Q
    - Different BMIs for different flags!

Tomorrow: Build Databases

- Still JSON
  - Simple format
  - Parsers are widely available
  - Mainly tooling-oriented (humans are unlikely to write these files)
- ISO C++ paper: https://wg21.link/P2977R1 (needs a new revision)
- Provided during the build (at least needs scanning to collect module usage information)

Build Databases: Covering the Gaps

- Group commands into “sets”
- Ordering and module usage
  - Includes information on modules provided and required by the TU in question
- Visibility of modules
  - TUs are tagged with a flag to indicate whether it can be used outside of its target
- Flag compatibility
  - Sets belong to “families”
  - Each instance of a family’s set is a different flag compatibility view of the set (e.g., CMake configuration or importer-influenced flags)

Kitware

Build Databases: The Skeleton

- Top level
  - `version`, `revision`
  - `sets`
- Sets
  - `family-name`, `name`
  - `visible-sets`, `translation-units`
- Translation units
  - `object`, `source`, `work-directory`
  - `arguments`, `baseline-arguments`, `local-arguments`
  - `provides`, `requires`, `private`

Kitware

Build Databases: Versioned

- Major and minor version numbers
- Major is bumped when fields are added or modified that change the semantic meaning of the contents
  - Better command line representations
  - Representations of code generation
- Minor is bumped for additional information that doesn't affect a correct interpretation of its contents
  - Compatibility hashes (see https://wg21.link/P2581 "Specifying the Interoperability of Built Module Interface Files" by Daniel Ruoso)
  - Input hashes
  - Modification times
  - Scanning performance metrics

Build Databases: Sets

- Each set has:
  - Name
  - Family name
  - Translation units
  - Visible sets (names of sets that provide modules that may be imported in this set)
- In CMake, families map to (per configuration) targets
  - Differently named sets represent “synthetic targets”
  - CMake will create such a “synthetic target” for each unique set of importer flags of a module-providing target (as of CMake 3.30, only done for `IMPORTED` targets)
- Using “visible sets”, the target graph can be known by tooling

Build Databases: Translation Units

- This is where existing compile database entries start showing up
  - Object (uniqueness constraint)
  - Source file
  - Work directory
  - Arguments (no “command” support)
- Additional fields
  - For modules build graph
    - Provides (mapping of module name to BMI path)
    - Requires (list of module names)
    - Private (boolean)
  - For compatible BMI
    - Baseline arguments
    - Local arguments

Build Databases: Translation Units 2

- What are “baseline” arguments?
  - Flags to apply to modules imported by the TU
- What are “local” arguments?
  - ```
    export module foo;
    #include “zlib.h”
    ```
  - Consumers do not need include paths for `zlib.h`
    ```

Build Databases: Current Status

- ISO
  - SG15 Tooling subgroup
  - Targeting the Ecosystem IS (https://wg21.link/P2656 “C++ Ecosystem International Standard” by René Rivera, et al.)
  - https://wg21.link/P2977 “Build Database Files” by myself and Daniel Ruoso
- CMake
  - 3.31 will have experimental support for creating build databases
  - Ninja generators only
  - Only exports C++ translation units

Kitware

Build Databases: Creating With CMake

```
set(CMAKE_EXPERIMENTAL_EXPORT_BUILD_DATABASE
  4bd552e2-b7fb-429a-ab23-c83ef53f3f13)
# Initialize the EXPORT_BUILD_DATABASE property on targets
set(CMAKE_EXPORT_BUILD_DATABASE 1)

find_package(WithModules)

add_library(A)
# add sources to A
target_link_libraries(A PRIVATE WithModules::WithModules)
```

Build Databases: Creating With CMake 2

```
$ cmake -DCMAKE_BUILD_TYPE=Release -GNinja -Ssource -Bbuild
-- Configuring done (0.0s)
CMake Warning (dev) in CMakeLists.txt:
  CMake's support for exporting build databases is experimental.  It is meant
  only for experimentation and feedback to CMake developers.

This warning is for project developers.  Use -Wno-dev to suppress it.
-- Generating done (0.0s)
-- Build files have been written to: build
$ ninja cmake_build_database
[0-1->6/6@122.4] Combining all module command databases
$ cat build_database.json | json4slides
```

Build Databases: Creating With CMake 3


```
{ "version": 1, "revision": 0,
  "sets": [
    { "family-name": "A", "name": "A@Release",
      "translation-units": [
        { "arguments": [ … ], "baseline-arguments": [ … ], "local-arguments": [ … ],
          "source": "source/a.cpp", "object": "CMakeFiles/A.dir/a.cpp.o", "private": true,
          "provides": { "a": "build/CMakeFiles/A.dir/a.gcm" },
          "requires": [ "with_modules" ],
          "work-directory": "build" } ],
      "visible-sets": [ "WithModules__WithModules@synth_eb703124e74f@Release" ] },
    { "family-name": "WithModules::WithModules@971c9b3b2695",
      "name": "WithModules__WithModules@synth_eb703124e74f@Release",
      "translation-units": [
        { "arguments": [ … ], "baseline-arguments": [ … ], "local-arguments": [ … ],
          "source": "prefix/lib/cxx/with_modules.cpp","private": false, requires: [],
          "provides": { "with_modules":
                        "CMakeFiles/WithModules__WithModules@synth_eb703124e74f.dir/b2e735ec3fff.bmi" },
          "work-directory" : "build" } ],
      "visible-sets": [] } ] }
```

Build Databases: Future Work

- Tooling
  - `clang-scan-build`
  - `clang-tidy`
  - IDEs
- Structured response files for argument representation
  - https://wg21.link/P3051 “Structured Response Files” by René Rivera
- Header unit support
- Adding non-C++ sources to the database

Q&A

Any questions? Comments? Concerns?

Contact:

- @mathstuf on Github, Reddit
- CMake Discourse and Kitware GitLab
- ben.boeckel@kitware.com

Beyond Compilation Databases

Supporting C++ modules with Build Databases

Ben Boeckel
CppCon 2024

Kitware