Document Number: P2962R0
Date: 2023-08-29
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Communicating the “Baseline Compile Command” for C++ Modules support

Abstract

This paper discusses what the “Baseline Compile Command” is, as well as the requirement that build systems need to communicate with both the dependency scanning process and static analysis tools. It also offers specific solutions on how that information should be communicated.

1. Introduction

In the discussion of P2898R1[1], it became clear that the dependency scanning process will need to have access to whatever the “Baseline Compile Command” is for a given translation unit. This is necessary because header units should be parsed without any of the “Local Preprocessor Arguments” that are applied to the translation unit performing the import.

While the semantic distinction for build systems that need to assemble compile commands likely needs to be done in terms of what the “Local Preprocessor Arguments” are for a given translation unit, the general consensus seems to be that only the build system must be concerned with the assembling of command lines.

The consequence of that approach is that the dependency scanning process should be given two command lines, one being the command line for the translation unit itself, and the other the “Baseline Command” that will be used to translate all imported header units.

This paper will explore the concept of “Baseline Command,” the benefits of using that concept and propose mechanisms for its use within build systems, toolchains, and static analysis tools (which include IDEs, code generators, and any other tool that needs to parse C++).

2. What is the “Baseline Command”

As discussed in P2581R2[2], the Built Module Interface has a much narrower interoperability across translation units than the produced object files. Any given module (named or header unit) may need to be translated multiple times in the context of a program. That paper makes the

[1] Ruoso, Daniel. Build System Requirements for Importable Headers. https://wg21.link/P2898R1
[2] Ruoso, Daniel. Specifying the Interoperability of Build Module Interface Files. https://wg21.link/P2581R2

distinction between the different types of arguments that are given to the compiler, and expands on the notion that “Local Preprocessor Arguments” should be allowed to be different between a translation unit that does the import and the translation unit being imported.

In the context of the discussion of P2898R1, SG15 has reached the consensus that Importable Headers should never have any Local Preprocessor Arguments. This is important because it will guarantee that translation units of different named modules that import the same header unit will always have the importable header parsed the same way. This is particularly relevant when you consider transitive imports that contribute to the same translation unit.

Also in the context of the discussion of P2898R1, SG15 has reached the consensus that we shouldn’t expect any tool outside of the build system to have the responsibility to extract the “Local Preprocessor Arguments” from a complete command, and that the Build System therefore has the responsibility of providing that information to the dependency scanning process.

Following those decisions, we can arrive at a definition:

The Baseline Compile Command is a fragment of the compilation command that does not contain: 1) which file is being translated, 2) which outputs should be produced, and 3) any Local Preprocessor Arguments.

The Build System is responsible for assembling both the complete Compile Command for each translation unit, as well as specifying the Baseline Compile Command that should be used when dealing with imports.

3. Communicating the Baseline Compile Command to the Dependency Scanning Process

Although the Baseline Compile Command should be a strict subset of the actual Compile Command used in the translation unit itself, prior attempts to design a mechanism to communicate the distinction of what arguments should or should not be considered part of the baseline ran into several issues:

- Introducing a new argument for each of the Local Preprocessor Arguments would have significant downstream impact on existing tooling that analyzes compile commands.
- Creating a position-dependent argument that denotes which fragment of the Compile Command should be considered Local Preprocessor Arguments creates a significant amount of difficulty for build systems that frequently need to assemble multiple fragments from different sources.

- There is no characteristic that determines what should be a Local Preprocessor Argument or not. Therefore, that information should always be authored by the developer and communicated to the build system and all other tools.

With that in mind, the solution instead seems to be to communicate the Baseline Compile Command independently from the rest of the command line. Thus, this paper proposes:

The Baseline Compile Command for a translation unit should be saved to a file, and the path to that file should be given to the dependency scanning tool as an argument in addition to the Compile Command for the translation unit being scanned.

Given the variety of tools that perform dependency scanning, this paper will not attempt to specify what that argument should be. Tooling implementers should attempt to converge whenever possible to provide a uniform user interface.

4. Communicating the Baseline Compile Command to Static Analysis Tools

In the context of Static Analysis Tools, the Baseline Compile Command is not only used for the correct interpretation of header units, but also it enables the tool to create its own plan for the parsing of dependent modules. It is very likely that the reusability of BMI files may have a different profile between the actual compilation used in the project and the parsing done by the static analysis tool.

There are two primary approaches used by Static Analysis Tools to introspect build systems in order to understand how the source code should be parsed. The next sections will cover those two approaches.

4.1. JSON Compilation Database

The JSON Compilation Database[3] file was introduced by the LLVM project as a way for build systems to communicate with Clang-based tools, such as clang-tidy, what the translation units in the project are and how those should be parsed. This format has subsequently been adopted by various other static analysis tools and frameworks.

The database allows Static Analysis Tools to be decoupled from build systems. The build system doesn't need any special instruction for invoking the static analysis tool, and the static analysis tool doesn't need to know what the build system is.

[3] https://clang.llvm.org/docs/JSONCompilationDatabase.html

Since the Static Analysis Tool will need to perform its own dependency scanning, it will need access to the Baseline Compile Command to be able to both correctly perform the dependency scanning and be able to produce its own BMI files.

Given those requirements, this paper proposes that:

The JSON Compilation Database format should add a new field to communicate the Baseline Compile Command; that field will complement the current `command` and `arguments`. Considering the format of `arguments` is already preferred, the new field should have that same format.

An example entry for a compilation database with the new field follows:

```
{
    "directory": "/path/to/build/dir",
    "file": "/path/to/source/main_translation_unit.cpp",
    "arguments": [ "g++", "-o" ,"main_translation_unit.o",
                   "-DFOO=1", "-DBAR=2", "-I/one/path",
                   "-I/other/path" ],
    "output": "main_translation_unit.o",
    "baseline-arguments": ["g++", "-DFOO=1", "-I/one/path" ]

}
```

This provides enough information for a system that observes the build system to correctly perform an independent header-unit-aware dependency scanning, in addition to correctly identifying how to parse dependent named modules for a given translation unit[4].

4.2. Introspection in subprocess calls from the build system

Tools such as Bear[5], compiledb[6] and Coverity observe the compiler invocations done by the build system in order to generate a compilation database.

When using this approach, it is possible that the invocation of the dependency scanner itself could be captured. If the dependency scanning process already receives the Baseline Compile Command, the processes observing the build process should be able to extract that information and propagate it to the specific tools that need it.

This paper recommends that:

[4] There is an additional requirement that a named module should be able to communicate its own Local Preprocessor Arguments, but that is outside of the scope of this paper.
[5] https://github.com/rizsotto/Bear
[6] https://github.com/nickdiego/compiledb

Document Number: P2962R0
Date: 2023-08-29
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Tools like Bear and compiledb should extract the Baseline Compile Command by observing the dependency scanning invocation and add the baseline-arguments field to the JSON Compilation Database they produce.

In order to correctly communicate the Baseline Compile Command to the underlying tools that need to parse the code, more specialized tools should be able to do something similar when they don't use the JSON Compilation Database as an intermediate format