Document Number: P2898R0
Date: 2023-05-18
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Importable Headers are Not Universally Implementable

Abstract

Importable Headers were included in the specification of C++ Modules in C++20. However, after considerable effort and discussion from the tooling perspective we currently don't have an acceptable approach for their implementation that fits into all cases where C++ can be used. This could lead to a bifurcation of the C++ Ecosystem, which is a profoundly undesirable outcome, therefore this paper recommends that part of the specification be removed. This paper also explores alternatives that would still allow us to achieve the underlying goals that were set for that feature.

1. State of Named Modules

Named modules introduce a fundamental change to how C++ code has to be built. We currently have a coherent understanding of the requirements for the support to Named Modules to be supported by build systems, even though we also acknowledge that an entire class of simplistic project build instructions will no longer be viable.

However, Named Modules also introduce an important limitation to the specification, which is that the preprocessor state is not inherited from a translation unit performing an import. This allows build systems to perform the dependency scanning and generation of the build plan to be performed in two distinct steps:

1. Identifying which modules are required and provided by a given translation: This can be done in an embarrassingly parallel way, since the only input needed for that is the compilation command for the translation unit and the source files.
2. Collation of dependencies into a unified build graph: Once all the dependency information is assembled, the build system can create a coherent build graph that is stable with all inputs and outputs clearly identified.

The industry also has significant experience in implementing this model, as it follows the same general design as Fortran modules. CMake has had support for Fortran modules since at list version 2.8, and that has been instrumental in the development of the support for C++ Modules.

Part of this effort has resulted in a output format for the dependency scanning in C++ module code that is already supported by Clang, MSVC and GCC[1].

However, the output file produced by the translation of a module interface establishes a new type of compatibility. While it is possible to link together object files produced by different compilers and compiler versions, Built Module Interface (BMI) files are significantly less compatible. Currently we don't expect compilers to be able to read a BMI produced by a different compiler, in fact, even different versions of the same compiler or specific compiler options may result in a BMI file that will result in a failure to import. This has also been a pain point in the implementation of Fortran modules.

That being said, we have reached a consensus on how to address those issues in C++ Named Modules [2] [3] [4]. Bloomberg has been working with Kitware to implement the required support for Named Modules according to that consensus in CMake. We also expect the draft C++ Modules Ecosystem Technical Report to formalize those requirements and implementation strategies. We don't foresee any major blocker to the implementation of those features, even though it still represents a significant investment.

2. Importable Headers are Fundamentally Different than Named Modules

There are two fundamental distinctions between Named Modules and Importable Headers for the purpose of the tooling implementation:

- Named modules create a new space for names where one didn't exist before, meanwhile Importable Headers share the same search space as Source Inclusion.
- Importable headers "leak" preprocessor state into the importing translation unit.

The following sections will explore the different ways in which those differences create problems for the tooling implementation:

2.1. Identity between "Importable Header" and "Source Inclusion"

There is no mechanism today that allows us to reasonably establish the identity between what an importable header and a header going through source inclusion looks like. Compound with the fact that the specification allows the compiler to replace an `#include` directive by an

[1] P1689R5: Format for describing dependencies of source files
[2] P2577R2: C++ Modules Discovery in Prebuilt Library Releases
[3] P2701R0: Translating Linker Input Files to Module Metadata Files
[4] P2581R2: Specifying the Interoperability of Built Module Interface Files

equivalent `import` can result in a situation where adding the information about a header unit in the module mapper for a translation unit could result in an entirely different file being processed, with entirely different semantics.

This identity problem has always been a complicated topic for the C++ specification, the `#pragma once` directive has been supported by various implementations in varying degrees of compatibility, but it cannot be universally implemented because we don't have a way of specifying what is the thing that should be included only "once" given the way that header search works.

2.2. Dependency Scanning Dependencies

The dependency scanning process needs a fully-capable preprocessor[5]. The shape of the dependency graph is, therefore, influenced by the state of the preprocessor. However, header units can change the state of the preprocessor, which means the dependency scanning process has to perform one of the following procedures:

- Accept the Built Module Interface for header units as an input to the dependency scanning[6].
- Emulate what the import would have done, by recursively creating a new preprocessor state using the command line arguments from the header unit in order to process it coherently and then merging the resulting preprocessor state[7].

Early implementations in Clang and MSVC dependency scanning simply processes those header units as if they were doing a source inclusion, accepting that it is "best practice" to only identify as header units the files that have no significant dependency on preprocessor state[8].

And while that approach has been successful in well-regulated environments (i.e.: monorepos), open-ended build systems (i.e.: mixing pre-built dependencies from a package manager) cannot guarantee that importable headers are well-behaved in that way, and therefore this will generate a level of friction for the tooling ecosystem that is not acceptable for those use cases.

It is possible to solve this problem, by doing the following:

- Include a list of all known importable headers as an input to the dependency scanning process

[5] The specification leaves room for an optimization where the preprocessor doesn't need to complete in its entirety, but instead can expand only the macros necessary for resolving what is required and provided by a translation unit.
[6] GCC currently only supports this mode.
[7] No compiler or dependency scanning tool currently supports this.
[8] This is the approach used in the adoption of explicit clang header modules

- Include the Local Preprocessor Arguments for those as an input to the dependency scanning process
- Communicate to the compiler or dependency scanning tool what are the Local Preprocessor Arguments for the current translation unit, such that the emulation can be performed correctly.

The cost of that approach, however, is that we create a significant bottleneck in the dependency chain of any given object. Changing the list of Importable Headers or the Local Preprocessor Arguments for any one of them will result in a complete invalidation of the dependency scanning for all translation units in the project.

While theoretically it would be possible for the build system to realize that the change didn't affect the results of the dependency scanning step, most build systems are not capable of interrupting the invalidation of artifacts based on that, resulting in an unacceptable incremental cost for the build of C++ projects.

The other alternative is for the compiler to collaborate with the build system to implicitly produce the information when a use of an importable header is found[9]. However, that is not a viable approach for environments where remote execution is used, since in those cases a complete list of inputs and outputs needs to be established prior to the execution.

2.3. Reasoning About the Preprocessor State

In the case of Named Modules, we have a significant advancement in the C++ ecosystem, as an import does not in any way affect the preprocessor state. With Importable Headers, on the other hand, a programmer looking at a `import` directive will need to step out to the build system in order to identify the command line that will be used to translate that header, and then create an understanding of which state gets merged back into the translation unit doing the import.

Again, considering the specification allows an `#include` to be replaced by an equivalent `import`, it means the programmer will not be able to reason what the preprocessor state may be without investigating whether or not that particular header is importable or not. This presents a significant challenge, particularly to those teaching C++. And that challenge is made even more acute considering that the specifics of how to answer that question will be entirely dependent on the specific build system being used.

This is going to be particularly challenging if the ecosystem ends up in a situation where different compilers make different choices about how to handle the implicit replacement of `#include` by the equivalent `import`.

[9] This is the approach used by implicit clang header modules.

3. The Goals of Importable Headers

Given all those challenges, before we identify a better solution, we need to take a step back and understand what are the goals that motivated the inclusion of Importable Headers into the specification:

- Take the lessons from the independent implementations of pre-compiled headers and make a coherent specification.
- Take the lessons from Clang Header Modules and make a coherent specification.
- Provide an easier adoption path for modules.

It is the position of the author that those are all worthwhile and important goals. However, the current specification does not actually deliver on those, and for environments where the use of Clang Header Modules was not viable it represents a significant risk of bifurcating the C++ ecosystem, or significantly increasing the incremental costs of the build of C++ projects.

In the next sections, I will elaborate on how the specification for Importable Headers fails to achieve those objectives:

3.1. Restrictions From Pre-Compiled Headers Were Not Preserved

While there are substantial differences on how different compilers support pre-compiled headers, there is a common subset of requirements that, if the user complies to, are consistently supported by the various implementers.

The main restriction that enables a interoperable use of pre-compiled headers is that the translation unit has to use it as a preamble to the translation unit, meaning the precompiled header is guaranteed not to be influenced by any other code in the translation unit.

This guarantees that when a compiler does not yet have the precompiled header available, doing the source inclusion is guaranteed to have the same effect as using the precompiled header when it is available.

3.2. Clang Header Modules Have A Narrower Scope

While they don't share the same restriction as pre-compiled headers, Clang Header Modules are implemented in situations where there is an assumption that the headers are not supposed to be influenced by the state of the preprocessor, and it is considered an user error if that restriction is violated. In other words, they are applicable in code-bases that are using a subset of the C++ Language. For code that follows that convention, falling back to plain source inclusion is considered a valid interpretation of the code.

Importable Headers, as specified, however, do not address the aspect of how we take the experience from Clang Header Modules and apply to the entire C++ ecosystem, where it would be unreasonable to support only a subset of the language.

3.3. Header Import Syntax Does Not Help With Migrations

The experience in the ecosystem on adopting the optimization of both Pre-Compiled Headers and Clang Header Modules rely extensively on the backwards-compatibility syntax and the fact that the intended canonical interpretation of the code is still the source inclusion. The operation performed by the tooling is currently understood as an optimization.

The addition of the header import syntax means that if code is switched to the new syntax, the canonical interpretation depends on the semantics of importing the header without it being affected by the preprocessor state at the point of the import.

However, adding a new syntax to a commonly used header would restrict the usability of that header only on compilers and build systems that support it. At that point, the difference in effort between transitioning to Importable Headers or switching to Named Modules is actually not substantially different.

In fact, for a codebase that wants to use module semantics, it would be trivial to automatically generate a wrapper module for a given header and export the entities from the header in that named module.

This, of course, doesn’t allow for interfaces that depend on macros, but refactoring the code such that the macros are placed in a header that doesn’t contain any other syntactic entities and continue to rely on source inclusion for those is going to provide us with a better migration path for named modules, particularly when you combine with the automatic generation of wrapper modules.

3.4. Header Units Performance Benefits Depends on Bottom-Up Adoption

Early efforts have also shown that if the adoption of Header Units does not start from the most low-level headers, the compiler will actually perform worse, because of how effective compilers have become at handling include guards during source inclusion.

If the adoption starts from higher-level headers that include the same lower-level headers, it means the compiler will now have to de-duplicate entities that otherwise wouldn't have been present in the translation unit.

4. Recommendation

All early experiences with importable headers, either in Clang or MSVC, rely on the transparent replacement of the `#include` directive by the equivalent `import`. In fact, the transparent fallback into the source inclusion route is a fundamental aspect for the successes in the adoption of those.

The characteristic of making that translation an implementation-defined aspect with semantics that are expected to be mostly equivalent to that of the source inclusion gives us the hint of what is the best way to redirect the specification.

Specifically, we don't need to specify a separate set of semantics for Importable Headers in order for that to be viable. We will achieve all the goals that were set before by instead specifying importable headers as a way in which implementations can optimize the source inclusion process.

The most significant outcome of this alternative approach is that the semantics of source inclusion are still the canonical behavior, and instead we would give permission to implementations to do something different in cases where it would be advantageous to do so, allowing the implementations to define their own caveats for that optimization.

A follow-up paper, or revision of this paper, will include suggested word changes once we have consensus on SG15 on the direction.