Document Number: 2702R0
Date: 2022-11-07
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Specifying Importable Headers

Abstract

This paper proposes guidelines for the way in which the build system and the compiler will communicate with each other about the importable headers. It examines some common requirements and suggests specific semantics for how tooling handles header units. This proposal focuses on scenarios where the approach of using a dynamic module mapper is not valid.

Introduction

The C++ Standard defines a distinction between “importable headers” and “non-importable” headers, but that distinction cannot be determined by the contents of the files. Therefore, the tooling will need to have additional metadata and communicate the list of importable headers between the build system and the compiler.

In addition, the interaction of the semantics of importing header units and including header files do not have an obvious interpretation. The goal of this paper is to provide some guidelines, such that we improve the interoperability between different tool providers.

While there are some implementations using a dynamic module mapper, there are scenarios where that approach is not valid, in particular in organizations that are adopting the Remote Execution Protocol[1] to distribute and cache the compiler execution. Therefore this proposal will focus on a convention that can be rendered entirely in terms of files on disk and command line arguments.

Recommendations

This paper makes a set of recommendations to drive better interoperability between implementations, and to make the overall experience of using header units less likely to generate user confusion. The term “implementation” here is used in a broad sense to describe the entire set of tooling presented to the user, so while those recommendations will require specific behaviors, it is not the intent of this paper to ascribe which particular part of the tooling will implement what requirements.

[1] https://github.com/bazelbuild/remote-apis

Document Number: 2702R0
Date: 2022-11-07
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Symmetry between import and include

The C++ standard gives a few hints that the token following an “import” and the token following an “#include” should refer to the same header, but it falls short of specifying that. Particularly, the standard allows an implementation to silently replace the usage of “#include <someheader.h>” by “import <someheader.h>”.

One other way that this expectation materializes is that there is a general understanding that a codebase may transition from using include to importing header units without changes in behavior.

This paper recommends that implementations guarantee that when a given token is used in both import and include directives, it should semantically map to the same header, either via source inclusion, or by consuming the bmi that was produced from the translation of a header unit that has the same file as its primary source file.

Alternative tokens for the same header

The common implementation of the lookup for include directives is to receive an ordered list of directories (starting with the directory of the source file, in the case of include with quotes), and traverse those in order to find a header file that should be included.

However, since it is possible for that ordered list to contain directories that overlap in contents (e.g.: include both `/usr/include` and `/usr/include/foo`), it is possible for the same header file to be addressed by more than one token. Likewise, symbolic links (in architectures that support that), further increases the possibility that the same header file is included in multiple ways.

The same applies to mixed usage of include with quotes, where `#include “foo.h”` may end up addressing the same file as `#include <a/b/foo.h>`.

The possibility that a header is addressable via more than one token sometimes contradicts the intended interface of a library, whose authors would prefer all their users to include the header using the same token.

On the other hand, it is not possible to specify that every header should be addressable by only one token. In some cases it is necessary to support the opposite.

This paper recommends that implementations support both a strict definition of which token should be used when importing a header, as well as deferring the decision for whether a header is importable or not depending on which file the compiler found based on the token and the ordered include directories.

There is already precedent for this in MSVC, where you can specify header units either as a tuple of the token and the path to the BMI or as just the path to the header file and the path to the BMI.

Fragility of the order of include directories

Problems caused by changes in the order of the include directories are a frequent experience in the C++ ecosystem. As we are required to explicitly enumerate the header units, there's a chance that something that is listed as an importable header is not actually ever going to be used because of shadowing created by changes in the order of include directories.

Since we already expect to support specifying the token that is expected to be used in the case of some header units, it would be possible to remove that fragility entirely at least for the cases where an intended token is specified.

Note that this means the compiler may need to map a token to both a header file and to a BMI file, such that in cases where the include is not replaced by an import, the same header is used.

This paper recommends that implementations do not perform the search in the ordered include directories for either importation or inclusion whenever the token is listed explicitly as a header unit, instead using the explicitly defined header file or BMI.

Consolidating multiple includes in a single import

Since the cost of importing a pre-translated header is potentially smaller than textually including and then translating that header multiple times, it is a reasonable optimization to consolidate a set of related headers into a single import.

This optimization would need to be entirely opt-in to the authors of the library, since that would be a potential behavior change. But in cases where a set of headers is always used together, it would be likely very advantageous to consolidate the importation of any of the headers into a single import.

And in cases where macros are not part of the intended interface, the authors may decide that it would be equivalent to importing a named module instead.

This paper recommends that implementations offer a way to specify that when a given token is used in include directives, that source inclusion can be replaced by the importation of a different header unit or named module.

Header units in the context of other translation units

There are two different interpretations for how header units should work. One of them is that the header unit should be translated using the same include directories and definitions from the

translation unit making the importation, the second is that the header units should be translated only once and used coherently across the entire project.

While the approach of translating the header unit using the definitions and include directories from the translation unit doing the import provides the most backwards compatibility with the behavior of headers today, it doesn't come without drawbacks.

The first drawback is the cost of translating the same header multiple times, which while it's likely still smaller than the header being included in all the translation units it would have been included, it's still an avoidable cost.

The second drawback is the lost opportunity to increase the coherence of the C++ code in terms of mitigating the risks of One-Definition-Rule violations. With header units we have a path forward where a given project would have a single coherent translation of a given header. Issues created by ODR violations created by source inclusion in different contexts are a common source of pain in C++ development.

Given those two drawbacks, **this paper recommends that implementations go in the direction of using a single translation of a header unit within a project, and that the include directories and compiler definitions used when translating the header unit are not defined by the translation unit doing the import or include.**

The outcome of this recommendation is that switching from inclusion to importation is not necessarily a change that is done without change in behavior, therefore the choice to make a header importable needs to take that into account.

This paper recommends the documentation of this change in behavior to instruct users that headers that depend on specific compiler definitions on the translation unit consuming the header should not be made into importable headers. (e.g.: -D_XOPEN_SOURCE, etc)

This leads us to a workable definition for the distinction between importable headers and non importable headers.

This paper recommends the definition of "importable header" to be: A header where the state of the preprocessor at the point of inclusion is not meant to semantically change how the header should be translated, and where isolation from that state is desirable. (include guards are an exception).

Identifying header units before dependency scanning

Since the compiler may replace header file inclusion by header unit importation whenever the compiler is given the information that a given header is also available as a header unit, the

presence of header units may fundamentally change the direct dependencies of a translation unit.

Therefore the dependency scanning phase needs access to the list of header units before the dependency scanning, particularly because if an include is replaced by an import, the compiler needs to report the dependency on that header unit as well.

While it would be possible for the build system to pre-emptively produce a BMI for every single header unit available, this has the likelihood to create unnecessary work, as not necessarily all header units will be consumed by the translation units in the project.

This paper recommends that implementations offer a way to specify a list of header units without identifying the BMI file for them when performing the dependency scanning step, such that only the header units that are actually used in a project need to have a BMI generated.

Given that the importation may affect the state of the preprocessor for subsequent directives, the compiler may need to simulate the importation of the header unit by performing up to phase 4 of the translation of the header file to identify macros defined at the end of the importation.

This compounds with the recommendation about header units in the context of other translation units, meaning the compiler needs to be able to simulate the importation using the preprocessor arguments that would be given to the actual header unit.

This paper recommends that the build system communicates to the compiler the preprocessor arguments required for performing the translation up to phase 4 of all header units in the dependency scanning step.