Document Number: P2581R1
Date: 2022-07-26
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Specifying the Interoperability of Built Module Interface Files

Changes

- R1
    - Change “Binary Module Interface” to “Built Module Interface”.
    - Introduce a new section exploring the outcome of having independent parsing context between the translation unit importing a module and the translation unit declaring a module interface.
    - Editorial changes throughout the paper for better readability, including removing redundancy with the new section.

Abstract

This paper specifies a mechanism to allow build systems to identify if a built module interface shipped with a pre-built library can be used directly, or if the build system needs to produce its own version of the built module interface file.

Introduction

Built module interface files are an implementation aspect on how modules are reused across different invocations of the compiler without the need for a new translation. The format of those files is implementation-defined. With the exception of MSVC, implementations are defining them to be only as interoperable as precompiled headers were.

While there is no fundamental problem with that approach, it does create an additional problem for libraries being shipped as prebuilt artifacts. In some cases, built module interface files in those artifacts are going to be reusable in a given compiler invocation, and in other cases they won’t.

The purpose of this paper is to establish a mechanism to allow for the clear identification of whether a given built module interface file will be usable within a given build context without the need to actually read the BMI files and without the need to invoke the compiler for each module.

Document Number: P2581R1
Date: 2022-07-26
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Requirements for a Built Module Interface to be Usable in a Different Translation Unit

The translation of a module interface happens independently of the translation units importing that particular module. While from the perspective of the language itself this sounds like a simple extension of the One Definition Rule, from the perspective of how the tooling sees different translation units it creates new requirements.

Particularly, specific compiler arguments used when translating the module interface need to be used coherently when translating a unit that imports that module. However, other arguments should be allowed to be different. **The ability to have an include directory or compiler definition that is only used when translating the module interface and is not necessary when translating a unit that imports that same module is a significant enhancement in the tooling space**.

The end result is that there needs to be a way to reason about which flags need to be used coherently, and which flags are allowed to be different. This paper introduces the following categories to organize those requirements:

- *Basic Toolchain Configuration Arguments*: The coherency on the toolchain configuration is not a new requirement. This is the same set of arguments that need to be coherent when two independent translation units in the same program include the same header today. This needs to account for the “benign ODR violations” that are common practice today.
- *BMI-Sensitive Arguments*: Current implementations, particularly clang and gcc, currently have a BMI format that is sensitive to changes that would otherwise not cause an incoherency on the basic toolchain configuration. For instance: libstdc++ supports translation units using different language standard versions to communicate within the same program (with documented limitations), however it is not valid to use a BMI translated with a different language standard version than the unit importing it.
- *Local Preprocessor Arguments*: Preprocessor arguments used when translating the module interface, but that would otherwise not be needed in the translation of the unit importing that module, also arguments needed in the translation unit importing a module but that are not needed when translating the module interface being imported.
- *Other Arguments*: Those are arguments that do not affect the preprocessor, and are also not relevant either for ABI coherency or the BMI format. They do not substantially change how the translation is made and therefore this paper will not cover their usage.

For the purpose of illustration, let’s imagine a program that has translation units A, B and C. Where A is a module interface, while B and C import that module. In this scenario, let’s imagine the usage of gcc.

```
// a.cpp
export module A;
#include <string>
#include <some_header.h>
export std::string foo() {
 return DEFINED_IN_SOME_HEADER_H;
}

// b.cpp
import A;
#include <other_header.h>
int bar() {
  return foo() + DEFINED_IN_SOME_OTHER_HEADER_H;
}

// c.cpp
import A;
#include <yet_another_header.h>
int bar() {
  return foo() + DEFINED_IN_YET_ANOTHER_HEADER_H;
}
```

As `a.cpp` uses `std::string` in its interface, this code needs to be compiled coherently in terms of the "dual abi"[1] supported by gcc. The end result is that the argument `-D_GLIBCXX_USE_CXX11_ABI=0` is a part of the Base Toolchain Configuration Arguments in this particular context.

However, as the interface exposed in module A doesn't depend on things that changed between the standard versions from C++11 to C++17, it would be valid to compile each of the three translation units to an object file using a different language version. That means the language standard version, in that case, is not a part of the Base Toolchain Configuration Arguments.

When translating `b.cpp` and `c.cpp`, on the other hand, the compiler needs access to a BMI of module A translated with the same language version. The outcome is that the module interface A needs to be translated more than once. In other words, that means the language standard version being used is a BMI-Sensitive Argument, and that it is valid to have different BMIs of module A in the same program.

Finally, the translations of `b.cpp` or `c.cpp` do not need the include directory for `some_header.h`, since that file is never seen by those translation units, and likewise `a.cpp` doesn't need the include directories for `other_header.h` nor `yet_another_header.h`.

[1] https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html

The arguments setting those include directories are Local Preprocessor Arguments, and the isolation of those arguments across those translation units is a desirable outcome.

In practice, “benign” violations of the One-Definition-Rule are an integral part of reality in the use of C++ today, and those must continue to be viable. Meanwhile, implementations have a much stricter requirement on what is supported when consuming a BMI on a translation unit that imports a module. Therefore, the same module interface may end up with different translations in different objects that end up composing the same program.

This paper proposes:

- **The package manager and build system should use implementation-defined ways to differentiate Basic Toolchain Configuration Arguments and BMI-Sensitive Arguments from Local Preprocessor Arguments for a given translation unit**.
- **The module metadata shipped with a pre-built library should specify only Local Preprocessor Arguments on the section describing how a build system can produce its own BMI file for that module. Those should be used in conjunction with the Basic Toolchain Configuration Arguments and BMI-Sensitive arguments of the translation unit importing a module when producing the BMI**.
- **The coherency of the Basic Toolchain Arguments between the pre-built library and of the code importing the module is the responsibility of the package management and build systems, as it is already the case in the pre-modules world.**

Built Module Interface Format Identifiers

The current consensus on SG15 is that module discovery should be driven by metadata. While there hasn’t been consensus that a single discovery mechanism can be used everywhere, there has been consensus on a viable mechanism that can be used in a large number of environments[2].

That convention should allow a build system to identify where the modules are and if there are candidate built module interface files to be used. However, **we still don’t have a mechanism to identify whether a pre-existing built module interface file can be used in the compiler invocation the build system will have to do**.

The proposal here is that **every built module interface file will have an identifier for the precise format of the file**. The module metadata shipped with a pre-built library will advertise the list of BMI files that were made available with the binaries. That list will include the format identifier of the BMI alongside the location of the file itself. Any consumer of the library will obtain the identifier for their own build context, and if a matching identifier is found, the build system will know that the BMI can be reused.

[2] P2577R2: C++ Modules Discovery in Prebuilt Library Releases. https://wg21.link/p2577r2

Document Number: P2581R1
Date: 2022-07-26
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Defining the Format Identifier

The scope of compatibility for consuming an existing built module interface file is defined by the union of the Basic Toolchain Configuration Arguments and the BMI-Sensitive arguments. And it should explicitly exclude Local Preprocessor Arguments. Those arguments are the ones that should be used to define the format identifier.

The compiler should offer an interface (e.g.: command line option) that will produce to the standard output the identifier as the first line. That command line should accept the regular compiler arguments as if parsing a module interface unit (although without specifying a translation unit). Arguments that are irrelevant to the format of the built module interface output should be ignored.

The semantics are that by combining the Basic Toolchain Configuration Arguments and the BMI-Sensitive Arguments from the translation unit importing a module with the Local Preprocessor Arguments from the module being imported you would be able to produce an equivalent built module interface file as the one shipped with a pre-built library.

A build system would, therefore, only need to invoke the compiler once for each set of Basic Toolchain Configuration Arguments and BMI-Sensitive arguments to get the identifier that would be used for all translation units using the same arguments, and then search the module metadata shipped with the pre-built library for a BMI with that particular identifier.

If a BMI with that identifier is found, the build system doesn't need to emit instructions on how to build that BMI again, however if a BMI is not found, then those arguments are concatenated with the Local Preprocessor Arguments advertised by the module in order to produce a local BMI.

When generating install instructions, the build system would use that same identifier when producing the metadata for the modules that are part of the library being distributed.

Compilers that support multiple input formats

While the author of this paper is not aware of any implementation that supports a number of formats concurrently, this is a significant optimization, and the author considers that it is justifiable to specify it ahead of time.

A compiler that supports multiple input formats for the same context should return the additional identifiers on the same output, following the first line. In other words, the first line returned specifies the format that will be produced, the following lines specify alternative formats that can be consumed.

One hypothetical use case for this would be if clang produces its own format when parsing the interface module units, but is also capable of parsing module files in the IFC format published by Microsoft.