Transitioning from `#include` World to Modules

Gabriel Dos Reis
Microsoft

Abstract

This paper is a report on a C++ implementation strategy to support transitioning existing libraries organized as set of include files to a world where module implementations have teeth and are used to naturally delimit components boundaries and dependencies in source code. The strategy requires no change to existing language rules. It is illustrated in concrete terms with the Standard Library Modules; the technique is scalable and applicable to libraries that can be modularized. The strategy relies on judicious build definitions and **`#include`** translation.

The Problem

Consider the following C++23 program:

```
#include <vector>
import std;
int main()
{
   std::vector<int> vee { 1, 2, 3, 4, 5 };    // #1
   return vee.size();
}
```

The focus here is not whether it is a wise programming practice to **`#include`** the header **`<vector>`** and also **`import`** the std library in the same translation. If you're bothered by that, just imagine that the **`import`** comes from a separate header file the content of which is not under the control of author of the program. This is a valid program, and the question is how can C++ implementations or development tools can ensure that the program compiles correctly.

The implementation problem is this: how can an implementation ensure that the reference to the **`std::vector`** on line #1 is not ambiguous or, for mysterious reasons, does not cause a crash?

You might ask: why is this even a question? Didn't the standard say this program is well-formed? The devil is in the details of C++ implementations.

An obvious implementation strategy (**S1**) of C++ standard headers (the ones that are not inherited from C and friends) is for them to essentially contain only

```
import std;
```

which delegates everything to the Standard Library module `std`. In that case, there is no problem whatsoever.

For various reasons, not all implementations have the desire or the resources to apply this implementation technique. Most want to continue to vend the same contents of their existing headers in pre-C++23 compilation modes. In those universes, the template `std::vector` is attached to the global module. Consequently, an implementation strategy (**S2**) for the Standard Library module `std` is to provide that template attached to the global module (e.g. via constructs like `extern "C++"` declarations) and to meet the letter of the standards. That is good and dandy. A downside of that implementation strategy is that it leaves on the table many advantages of name modules over the global module, in particular guaranteed ODR by construction which is boon for improved compile-time speed up.

So, what can an implementation wanting to attach standard entities to the std module do if they want to enjoy maximum profits of named modules and at the same time continue to vend their headers the way they used to? There are undoubtedly many tricks that implementers can deploy. The rest of the paper is devoted to a technique (**S3**) that scales beyond the Standard Library module to any C++ library considering modularization.

A solution: #include translation + BMI mapping

The core of the strategy **S3** is to have the compiler and the build definition conspire to deploy the obvious implementation technique **S1** (all standard headers are as if importing `std`) through `#include` translation coupled with BMI mapping. That is, the build definition is set up so that:

1. translate `#include` of standard headers to import of header units.
2. Direct the compiler to use the `std` BMI for all standard header units.

Step (1) asks the compiler to replace textual inclusion with imports of header units. That process requires a backing BMI for the corresponding header units. That BMI can be anything as long as the expectations placed on the declarations it makes available are satisfied. Step (2) ensures that all standard declarations are made available, hence satisfying the requirements on the header units it is backing. Note that this step is totally conforming

because any C++ standard header (excluding those inherited from C) is permitted to implicitly #include any other standard headers. Because of step (2), it is important that the build system does not go off trying to automatically build BMIs for header units that might be discovered during dependency scanning. Those BMIs should be built only for the header units that have not been mapped in step (2).

What about macros?

The core strategy works pretty well for most importable C++ standard headers. However, there are a few standard headers that are documented to make certain macros available (like `errno`) and since modules don't export macros, how can the `#include` translation + BMI mapping work?

The solution is to have those macros be "force `#include`d" on the command line. One could imagine one macro-file per standard header that is force `#include`d. Or the build set up can use force include a single macro file that provides all standard macros. We recommend the latter approach for simplicity purposes.

Summary

In summary, the suggested implementation for transition strategy relies purely on build set up and requires no language rule changes:

1. `#include` translation
2. mapping from header units to appropriate named module BMI
3. Forced inclusion of standard macro files.

There is no additional requirement on build systems. A build generator may need to support user-supplied BMI mapping and `#include` translation.

Generalizing beyond `std` module

The implementation strategy S3 just illustrated for the Standard Library module `std` applies to general modules, not just `std`. The description above makes uses of some standard guarantees, such as any standard header can `#include` any other standard headers. What is needed?

A. Ability to describe that certain headers are subsumed by a named module (or even another header)
B. Ability to translate `#include` to `import`
C. Ability to map the BMIs of the header units from (B) to the subsuming module's (or header unit's) BMI.

D. Ability to "force `#include`" macros that would otherwise come from the header files.

It is critical that the build system does not use the output of the dependency scanner after step (B) to automatically generate build of BMI for header units without taking into account the BMI mapping from step (C)

Acknowledgements

The transition strategy **S3** detailed in this paper is based on ideas and techniques published in section 4.2.2 of the 2016 paper "Modules, Componentization, and Transition" by Gabriel Dos Reis and Pavel Curtis [P0141].