Document Number: P3470R0
Date: 2024-10-15
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15, EWG

Interface-Unit-Only Module Library Support

Abstract

This paper discusses the limitations of the language and ecosystem on the support for distributing module libraries as “Interface-Unit-Only” (i.e.: the equivalent of “header-only”). It also proposes a mechanism that we could use to get us in that direction by using a new syntax, which may either be an attribute or another use of the inline keyword.

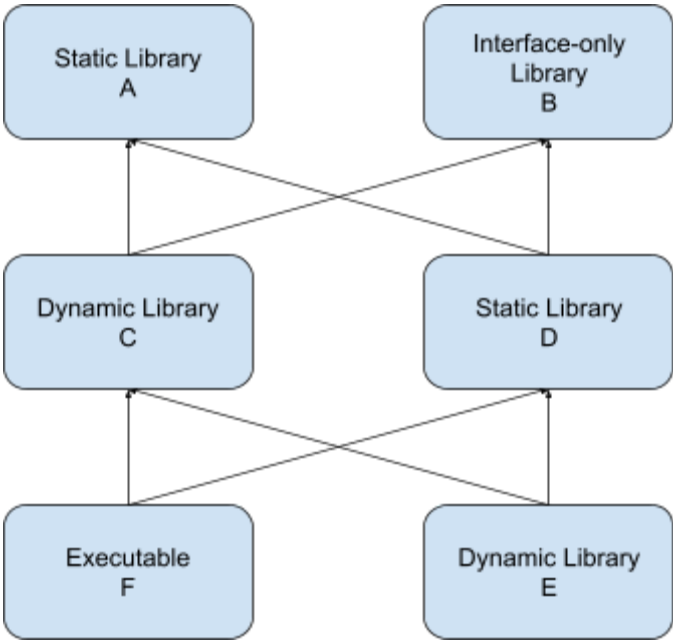
1. Motivation

In the C++ ecosystem, we have a long history of libraries being distributed as “header-only”, meaning that they don’t need to be distributed in an architecture-specific form, and they don’t need to be included in the link line for the final application. With the introduction of modules, we are seeing a significant push from the community to support the equivalent form of distribution with modules. We call that “Interface-Unit-Only”.

With the current implementation of modules, however, there is a consensus, reaffirmed in the 2024 Tokyo meeting, that for every C++ named module importable unit there will be exactly one object that is produced for that module for a single program. This is important because it allows significant optimizations in the code generation for the code that imports a module.

This is a problem for users that want to continue providing only an architecture independent distribution of their libraries, without the expected objects for the named modules. The way that the problem manifests is that it is not clear who is responsible for producing those objects, and there doesn’t seem to be a correct answer to that question.

To illustrate the case, let’s imagine the situation where you have a set of interdependent libraries, some of which may not even be C++, that need to be linked together into a final program or dynamically loadable library.



In that scenario, who would be responsible for generating the objects required for the modules in the interface-unit-only library B?

We can defer this to the final executable to include the object on its link line. However, in that case, how do we communicate to the build of executable F that this is needed? That build of that executable may not even be aware that library B exists.

Furthermore, this would require the build of Dynamic Libraries C and E to leave those symbols undefined. Authors of those libraries frequently will avoid leaving symbols undefined, because that can easily hide mistakes in either the code or the build system.

We could say that the direct user of that library should include that object in its own library. However, in that case those symbols will be present both in Dynamic Library C and the Static Library D, which could result in linker errors. Particularly in architectures where symbol resolution in dynamic loading is done globally, such as Linux.

In the 2024 St Louis meeting, we reached consensus that we should not expect build systems to produce objects for external libraries, because of that issue.

Therefore, the status quo is that Interface-Unit-Only libraries are not supported by the current language specification and tooling ecosystem.

2. Modules ABI

To understand what exactly is happening that precludes the support for Interface-Unit-Only libraries, it's important to understand that modules imply a specific ABI. One example of how that manifests is on the call to the static initializers in a module purview.

It is important to note that, in the Itanium ABI, even if the module doesn't currently have anything that needs static initialization, the translation unit importing that module will still generate the code to call the initializer, because we don't want the addition of a static initializer to a shared object to result in an ABI break.

At the same time inline functions and debug types can be omitted in the translation unit importing a module, because those are guaranteed to be provided by the object produced for that module interface unit.

Additionally, because we can count on having exactly one object for each module in the final program, we have additional room for future optimization where the compiler may omit the code generation for any template instantiation that is already going to be present in the module's object.

Those are important optimizations, both in terms of the size of the intermediate objects, and in the compile time for those objects.

The desire for maintaining this optimization space has been reaffirmed in the 2024 Tokyo meeting, where we reached consensus on this principle.

3. Interface-Unit-Only Modules as an alternative, but compatible, ABI

If we can't rely on the build system, and we want the Modules ABI to preserve the room for optimization that it currently has, we need to look at an alternative mechanism for solving that problem.

The alternative proposed here is to introduce an alternative, but compatible, ABI for modules that are going to be distributed as an Interface-Unit-Only library.

The difference is that in that alternative ABI any symbol that would have been left as undefined in the object importing a module would instead be generated as a weak symbol in every use.

This is compatible because this is the same technique that we already use for headers, and it's essentially a mechanism to disable the optimization that we want to preserve in modules that are distributed in pre-built libraries.

4. Selecting the alternative ABI

Since this actually requires different behavior from the compiler importing that module, it's important that we're able to clearly communicate which modules being imported by a given translation unit need to use that ABI.

It would definitely be undesirable to disable that optimization for every module being imported by the translation unit. Therefore we need to be able to specify that individually for a given module.

It would be possible to consider that as an argument to the compiler, however, there's currently no mechanism that we could use to propagate that information, and it would also generate significant complexity for the build and packaging ecosystem.

At the same time, the module being imported already has to be processed by the compiler, which already provides a mechanism for that information to be conveyed.

This paper proposes that the best place to document the choice for this alternative ABI is in the source code for the importable unit itself.

This has the lowest cost of integration, since we don't need to introduce new compiler flags just for this purpose and communicate the use of those flags across the ecosystem.

It also is the most explicit and portable mechanism we have available to document that choice, as it would be entirely unambiguous what the intention of the author was.

It also has the benefit of allowing the compiler to validate, at compile time, that all declarations have an inline definition in the module as well as any local linkage entities, preventing the user from creating code that would not be able to work at all.

4.1. Syntax

There are two mechanisms that we could use in syntax to communicate that information to the compiler, using the inline keyword, or using an attribute.

4.1.1. Attribute interface-unit-only

This would be the most lightweight change to the language, as it would be simply a standard attribute attached to the "export module" declaration, instructing the compiler to use that alternative ABI when importing that module.

It doesn't change any of the behavior of the language itself, the only impact it has is instructing the compiler to use that alternative ABI.

It's also possible that this attribute can be used to restrict the usage of the language in a way that prevents declarations in that module's purview without an inline definition in that same translation unit.

4.1.2. inline module

Since the change in ABI is similar to the behavior of inline definitions in the language overall, another option would be to introduce a new syntax by allowing the inline keyword between the export and the module keywords.

This would have the same effect as the attribute. It is a heavier-weight change to the language.

5. Additional issues with Interface-Unit-Only Module Libraries

It is important to note that solving this problem is necessary but not sufficient for the ecosystem to be able to properly support this form of distribution for module code.

Unlike with headers, modules require additional metadata to be distributed with that library. As things stand right now, this form of distribution will require build-system-specific integrations that would allow the identification of what modules are available in that library.

Additional work in the package management space will be required in order to create an interoperable mechanism to distribute libraries in this form.

6. Next steps

At this point, the goal is to build consensus on:

- Do we want to pursue support for libraries distributed as “Interface-Unit-Only”?
- Do we want to pursue implementing that support in syntax?
- Which flavor of that syntax do we prefer?