Document Number: p1213r1
Date: 2018-11-12
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: N4720 Merging Modules

Global Module Fragment is Unnecessary

Nathan Sidwell

The merged modules draft retains the global module fragment of the TS. Its semantics are defined in terms of legacy header units, but with a few exceptions. Exceptions are awkward, we should be wary of committing to them.

TL;DR; Presented at San Diego'18. Did not gain consensus.

1 Background

Both legacy header units and the global module fragment are part of the modules specification in order to solve the *how do we get there from here* problem. Namely module unit source will need to interact with non-modular header files. The headers cannot be included within a module's purview as their declarations would then be part of the module and, if well-formed at all, usually gain module linkage.

The global module exists to solve this. It is an unnamed module where all legacy code resides. Arbitrary collections of entities can be placed there in any particular compilation. Across compilations the ODR is still applicable (retaining the traditional duck-like type identity semantics).

The TS's global module was a collection of global module fragments, each specific to a particular translation unit. Syntactically the global module fragment can only consist of `#include` files. The leading nameless module-declaration and the terminating module-declaration must be present in the top-level source file.

The ATOM proposal added legacy header units. These are import declarations using `"quoted"` or `<angled>` module names. The intent is the name maps to a header file that is processed in some implementation-defined manner to create a legacy header BMI.[1] All the contained entities are part of the global module. An entity may be defined in multiple legacy header units, without necessarily violating ODR.

In both schemes, an entity may be defined in multiple global module fragments, and/or legacy header units. Should the two definitions become present in a particular compilation, the compilation system

1 An intermediate step of creating a Binary Module Interface matches current compiler implementation strategies. It is not a critical feature of this paper.

must arrange for them to be merged in an appropriate manner (possibly giving a diagnostic, if the declarations or definitions are incompatible).

The semantics of the global module fragment are *almost* as if the textual contents are present in a separate source file (including any controlling macro definitions), which is imported as a legacy header unit in the module preamble. For instance:

```
// foo.mcc
#define ARBITRARY_MACROS …
module;
// global module fragment
#define MORE_MACROS …
#ifdef CONTROLLING_MACRO
#include <header-a.h>
#else
#include <header-b.h>
#endif
export module foo;
// module purview
…
```

is *almost* equivalent to:

```
// foo-global-frag.h
#define ARBITRARY_MACROS …
// global module fragment
#define MORE_MACROS …
#ifdef CONTROLLING_MACRO
#include <header-a.h>
#else
#include <header-b.h>
#endif

// foo.mcc
export module foo;
import "foo-global-frag.h";
…
```

The two differences concern visibility in implementation units and entity pruning, which are addressed below in Sections 1.1 & 1.2.

The above is an extreme example, presuming no commonality between global module fragments. It is very likely that commonality exists within the units of a specific module and between different modules of a project. A number of `#include` files being frequently used in global module fragments. Thus one is unlikely to require per-module bespoke global module fragment headers. A more realistic example might be:

```
// foo.mcc
#define ARBITRARY_MACROS ……
module;
// global module fragment
#define SELECT_MODE_A 1
#include <someheader.h>
#include <os.h>
export module foo;
// module purview
…
```

which is almost equivalent to:

```
// someheader-A.h[2]
#define SELECT_MODE_A 1
#include <someheader.h>

// foo.mcc
export module foo;
import <someheader-A.h>;
import <os.h>;
…
```

Some modules would contain '`import <someheader-A.h>;`', and some contain '`import <someheader-B.h>;`', which would be constructed from a legacy header selecting mode B for '`someheader.h`'.

This exposes two different kinds of header file, which can be termed 'augmentable' and 'modal' respectively. Both kinds may be combined into a well formed executable, from translation units including them in a variety of configurations. However, a single translation unit may only include a particular header in a specific configuration.[3] The degenerate case is a header file with a single mode and no augmentations.

An augmentable header file is one where additional, non-conflicting, functionality may be selected over the default, by defining specific controlling macros before inclusion. A modal header file is one where controlling macros select between mutually exclusive functionality.

An augmentable header file with all augmentations enabled may be used as a single legacy header unit – users that do not require the additional functionality will be able to ignore it. An example is defining `_XOPEN_SOURCE` before '`#include <unistd.h>`' to obtain access to a `readlink` function.[4]

2 With suitable compiler invocation, this intermediate source file need not exist, and the resulting legacy header unit could be perhaps named `<someheader.h/A>`, located at an implementation-defined pathname.

3 The headers are usually idempotent, so the first inclusion determines the configuration. Idempotency is not a critical requirement of this paper.

4 From the GNU C library.

A modal header file cannot do that. It could instead be considered as a set of legacy header units, each member of which selects a particular mode. An example is defining `UNICODE` before '`#include <windows.h>`' to select Unicode (`${FOO}w`) as opposed to ASCII (`${FOO}a`) functions.[5]

In practice it has been found that an augmentable header file may be used with a union of augmentations. Modal header files are often used in more than one mode, within a single set of compilations.[6]

1.1 Visibility in Module Implementations

One difference between an interface's global module fragment, and imports within its purview is that the latter can be visible to its implementation units whereas the former, in general, are not. This might be confusing to users, unappreciative of the subtlety that the global module fragment is not part of the module interface.

The current draft requires such global module fragment declarations reside in header files included from the global module fragment. In migrating to modules, the first step might be to move those includes from the global module fragment to become legacy imports within the purview. Doing so may leave unnecessary legacy imports in an implementation unit (these would be harmless).

The rationale for this difference in visibility is that it permits pruning of the global module fragment entities when writing the BMI.[7]

1.2 Entity Pruning

Entity pruning was recommended at the Bellevue'18 meeting. Only global module fragment entities directly reachable from the interface's purview are made available to module importers for certain uses. Entity pruning permits a significant reduction in the size of a BMI containing a global module fragment. A simple module including `windows.h` fell from a few MB to a few KB.[8]

The entity pruning also reduces the complexity of merging global module fragments across modules transitively imported into some other compilation unit – the sets of entities to merge is smaller.

As entities are not pruned from legacy header units, the merging complexity with them may be greater. However, as mentioned above, different augmentable header files are not expected to often declare the same entities. The closest that may occur is declaration of different members of an overload set, where the merging process merely has to determine declarations are different members of the set.[9]

5 https://en.wikipedia.org/wiki/Windows.h#Macros
6 Conversations with Google and Microsoft engineers respectively.
7 The Bellevue'18 pruning scheme makes the reachable global module declarations available to module implementation units.
8 Gaby dos Reis, Microsoft Visual Studio, Toronto'17.
9 It is not necessary to determine that the declarations are unambiguous, although that might be desirable.

Modal header files may be organized differently. It would be possible for the mode selection to only declare the entities available in that mode. That is commonly not the case, all entities are declared regardless of mode. In either case macros are used to select the set of underlying entities using mode-agnostic names. The latter organization allows users to explicitly select entities of a different mode, which is occasionally required.

The first organization has the same (small) merging complexity of legacy header units of augmentable headers. The second scheme will produce legacy headers requiring merging of all declarations. Even then, the merging complexity will be $O(N_{modes})$ rather than $O(N_{modules})$, which would usually be a much smaller number.

1.3 Global Module Fragment Import Visibility

The non-visibility of the global module means that imports are not idempotent in a surprising way. An import in the global module fragment is private to the interface, whereas an import in the module's purview is visible to implementation units too. The same module could be imported in both places.

The Bellevue'18 pruning scheme does not consider the case of entities provided only by imports within an interface's global module fragment. This has an effect on incremental conversion of source bases.

1.4 ADL

Resolution of dependent expressions at template instantiation time performs additional lookups. Traditionally ADL in the context of the instantiation and in the context of the template's definition. Literal mapping of the latter lookup into modules requires preserving the set of visible entities for a module-defined template. For reasons similar to not making an interface's global module fragment visible in implementation units, it is undesirable to make the global module fragment visible to such ADLs.

However, completely hiding the global module fragment is also problematic, as it provides a module interface no way to make customization points of global module fragment types it exposes available to its users.

Thus the entity pruning scheme of Section 1.2. Reachable global module fragment entities are made available via ADL to extra-module instantiations of templates defined in the interface, or involving the module's types. This satisfies the need to reduce the BMI size, but gives module implementors a mechanism to expose customization points.

2 Discussion

The above-described semantics and the rationale behind them relies on some assumptions, that are not warranted. They give rise to some fragility, which it would be better to avoid. they also give rise to some unexpected consequences and ambiguities.

Sections 2.2 & 2.3 show methods of turning a modal header file into legacy headers, with different user-visible interfaces. Pruning is not the only solution to reducing merging complexity.

2.1 Lazy Loading

The Modules-TS encourages lazy loading of imported entities, as it is not an error to import modules with conflicting declarations. It is only upon use of those entities in a context where the ambiguity is significant that the error becomes diagnosable. If entities are lazily loaded upon lookup, one need only merge entities lazily too. Thus the rationale for reducing the entity merging complexity is suspect.

Three compilers implementing modules (Microsoft Visual Studio, Clang & GCC) all implement lazy module loading. In the GCC case, import processing is very cheap, marking namespace module bindings as requiring loading. Loading occurs upon lookup finding that module's namespace binding of a name is needed.[10] Microsoft Visual Studio is similar,[11] and I understand Clang is not too different. The rationale is to improve the compile-time performance of using modules even in the absence of merging complexity.

In Section 1.2, modal legacy module unit merge complexity was described as $O(N_{modes})$. With lazy loading this is further reduced according to the fraction of names looked up. For a large & complex importing translation unit this presumably tends to 1, which is not helpful.

2.2 Synthetic Legacy Header

Should lazy entity merging of modal legacy imports prove to be a significant cost, it can be reduced to zero, by mechanical processing of the modal header to generate a common legacy header, and wrapper legacy headers for each mode.

1. Several compilers have a preprocessing mode that extracts the set of macro `#define` and `#undef` directives.[12] Use this feature to preprocess the modal header file in each required mode.
2. Preprocess the header file normally in each mode. The results should be the same – otherwise there are likely ODR problems. Any one of these can be compiled as a master legacy header.

10 Such lookup might occur for inter-module references too. GCC uses such a technique, Clang does not.
11 Conversation with Jonathan Caves.
12 GCC's & Clang's '`-dD`' or '`-dM`' or Visual Studio's '`/d1PP`' preprocessing options.

3. Each modal legacy header may be created by importing the master legacy header (or including and relying on include translation) into a modal header also containing that mode's macro directives.

Some compilers have preprocessing mode that process only conditional and include directives, leaving macro directives alone (other than that necessary to determine the conditional inclusion), thus generating partially preprocessed source.[13] It should be possible to use this mode instead in step 2, which may result in better diagnostic fidelity within the master legacy header.

The result of this transformation will be a set of N_{modes}+1 legacy headers, requiring no intra-modal-header merging upon import.

2.3 Retaining a #include Interface

Perhaps, user familiarity with define and include directives makes:

```
module;
#define UNICODE
#include <windows.h>
export module foo;
…
```

a more acceptable transitional step than:

```
export module foo;
import <windows.h/UNICODE>;
…
```

With a suitable `<windows.h>` or header translation (Sections 2.3.1 & 2.3.2) it would be possible to write:

```
export module foo;
#define UNICODE
#include <windows.h>
…
```

2.3.1 Rewritten Header

The included `<windows.h>` file could contain:

```
// new windows.h
#ifdef UNICODE
import <windows.h/UNICODE>;
#else
```

13 This is GCC's '`-E -fdirectives-only`' functionality. Clang has a similar mode. Visual Studio does not.

```
    import <windows.h>;
    #endif
```

That this is found rather than the non-module `windows.h` file could be achieved through search path manipulation.

Alternatively, the non-module `windows.h` file could be augmented to contain the above snippet, suitably protected via a module-specific feature-test macro, `__cpp_modules`.

The algorithms of Section 2.2 could be employed to construct the legacy header units. This could be done by the library vendor, rather than the end user.

2.3.2 Header Translation

Header translation could be required to translate the inclusion into an import of a suitable legacy header unit. Such translation would also have to consider the macro set at the point of the import.

Include translation is encouraged outside of legacy header compilations themselves, as one decision at Bellevue'18 was to require no post-preamble legacy imports. The rationale for that was to make generating preprocessed output simpler,[14] but it does so by making compilations themselves more expensive – unless header translation occurs.

2.4 Reachability

The new *referenced-entities-are-visible* rule is possibly fragile. The set of such entities is affected by:

- The definitions of interface-defined or global module fragment functions.
- Instantiations occurring in the interface (of non-interface templates)
- Dependent calls containing an unresolved overload sets from qualified or unqualified names.
- Whether the declarations are directly in the files included from the global module fragment, or obtained from an import declaration therein.

If such items are reachable, directly or indirectly, from the set of exported or module-linkage entities of the interface, they become part of the reachability graph. This graph could easily extend into template definitions, or instantiations thereof, from within the global module fragment itself. Changing a function's inlinedness would be sufficient to change the graph (inline function bodies are part of the graph, non-inlined ones are not).

This has similarities to the fragility of the reachable semantics concept of the TS, which has been replaced by the simpler cumulative semantics rule of the ATOM proposal.

14 I have not found this to be an impediment in generating preprocessed output from sources that import legacy header units.

2.5 Global Module Fragment Imports

Import declarations may appear (indirectly via header inclusion) in a global module fragment. Both named-module and legacy header units may appear. Entities introduced by such imports are not subject to the reachability analysis, and therefore will not be visible to importers of the module containing the global module fragment.

This has consequences for incremental modularization of source code. One cannot replace an include directive with a legacy module import without affecting the reachable entity graph. This includes the replacement done implicitly by include translation. Unfortunately, include header translation is *required* should the included header be a legacy header unit. Without such translation global module multiple definition semantics are easily violated.[15]

As legacy module units are also part of the global module, perhaps a case could be made for extending the reachability analysis into them. But then we must ask where this stops? We will also be in the strange situation of either writing out in a BMI the contents of another BMI, or writing data to make another BMI partially visible. These do not seem attractive requirements.

Placing import statements at any top-level location gives great flexibility in how code bases may be converted to modules. This feature was retained from the published TS, rather than mandate ATOM's preamble. However, this effect increases the brittleness of such conversions that elide the step of moving to legacy header units.

2.6 Eliding Template Instantiations

Consider an implementation that elides instantiating the body of a template in a module interface unit, because the instantiation is already known available via some transitive import (the instantiation definition may not be in the BMI, but simply known to have been emitted in an object file). That is a fine compile-time optimization. Except that [temp.inst] repeats the mantra:

> ... template specialization is implicitly instantiated when ... or if the existence of the definition affects the semantics of the program.

Instantiation affects the entity pruning, which will affect the set of ADL-visible functions for importers of the module. As that is a visible semantic effect, it appears that template instantiations cannot be elided.

15 Some discussion in p1218r0: Redefinitions in Legacy Imports

2.7 Non-Dependent Calls

Some compiler implementations keep the overload set of non-dependent calls until instantiation time, either for pragmatic purposes, or for diagnostics.[16] Are all the members of the overload set referenced for the purposes of entity visibility?

Are conversion functions and deduced template arguments of the resolved call referenced entities? These are not immediately available in some implementations.

2.8 Compiler Speed Optimization Is Difficult

The global module fragment can only consist of preprocessor directives. Of course some of those will be include directives that contain other declarations. However it is not possible for the compilation system to treat each of those includes as-if a legacy import (via include translation), because:

1. Textual include semantics remain. Macro definitions from the including file and previous includes are active within the include.
2. Textual include semantics remain. Declarations previous to the include are visible to name lookup.
3. The pruning semantics of Section 1.2 remain. Thus the set of entities made available to ADL is not that of a legacy header import with the same contents.

Were it not for these features, a compilation system could perform include translation. Case 1 could be accommodated by examining the set of macros at the point of inclusion and selecting a suitably built legacy header.[17] Case 2 could be achieved similarly by pre-including the preceding headers when converting an include to a legacy header.[18] Case 3 is much harder to achieve, and may well be unachievable practically.

These concerns could be addressed by relaxing the global module fragment semantics such that:

A) Only macro definitions from the including file are significant to the semantics of the included file. Macros from preceding includes are not. This would simplify case 1.

B) Header files must include their dependencies. They cannot rely on declarations from a previous include. This resolves case 2.

C) The entity pruning is optional. This resolves case 3.

Cases A & B would be no diagnostic required.

16 Of course they must be resolved to a non-dependent type during parsing.
17 Header translation would be a function of the header name, include path *and* macro set.
18 This could easily lead to a combinatorial explosion of legacy header variants.

3 Proposal

The global module fragment was introduced as a transition mechanism. In order to ameliorate certain implementation drawbacks, additional restrictions were added to its behaviour. As described above, those restrictions appear to have unintended, possible implementation-dependent, consequences. They also restrict implementation optimization opportunities.

The legacy header unit transition mechanism is now available, and some of the semantics of the global module fragment redefined in terms of the new mechanism.

It is possible to rewrite global module fragments in terms of legacy header units, including an automatable scheme described in Section 2.2. Section 2.3 further shows how to wrap those legacy headers in an include file whose interface is exactly that of the original non-modular header. One need not rely on entity pruning to reducing merging complexity.

Because the impact of the global module fragment on other features, it is prudent to not include the fragment in the initial requirements for modules that may go forward to the Working Paper.

The proposal did not gain consensus at the San Diego’18 meeting:

> p1213r0 Should C++ 20 modules have the global module fragment (module; #include … \
> n module foo;)
>
> SF:8 F:7 N2 A:5 SA:5

The proposals were not adopted into p1103.

4 Revision History

R0 First version

R1 Added alternative include mechanism (2.3). Added build optimization description (2.8). Include results of San Diego ballot.