Document Number: P0731r0
Date: 2017-07-27
To: SC22/WG21 CWG/EWG
Reply to: Nathan Sidwell
nathan@acm.org
Re: Working Draft, Extensions to C ++ for Modules, n4681

Module Interface Imports

Nathan Sidwell

The current wording of n4681 does not reflect the original intent concerning imports. This paper discusses the current wording and suggests edits to clarify the original design.

1 Background

The modules-ts states that module implementation units have visibility to all the non-internal-linkage entities *declared* in the module's interface unit. The specification is not clear on whether module implementations units have visibility of all the non-internal-linkage entities made *visible* in the interface unit. Three implementations of modules, in various states of completeness, all diverge in their behaviour regarding module import transitivity. It is unknown whether this is due to incompleteness of implementation, or ambiguity of specification or both.

2 Example

This example was provided by Boris Kolpakov and shows implementation divergence. It consists of two modules and a user.

```
// module core interface
export module core;
export void f ();
```

```
// module extra interface
export module extra;
import core;
export void g ();
```

```
// module extra implementation
module extra;
#ifdef IMPL_IMPORT
import core;
#endif
```

```
void g () {
  f (); // #1
}

// user
import extra;
#ifdef USER_IMPORT
import core;
#endif
int main () {
  f (); // #2
}
```

Here the end user imports module `extra`, which itself imports (but does not re-export) module `core`. The three compilers tested all vary in whether they require explicit '`import core;`' in the module extra implementation TU and/or the user TU:

Compiler	Version	IMPL_IMPORT	USER_IMPORT
G++	modules branch	Not needed	Needed
Clang	`5.0.0-svn305177-1~exp1`	Needed	Needed
VC	`19.11.25325`	Not needed	Not Needed

VC appears to be transitively importing core into the user's TU and thus the call at #2 is resolved to the function declared in core's interface. Clang does not appear to be making the import visible within the purview of module implementations and so requires an explicit import to resolve the call at #1. GCC is not transitively importing core, but is making the import visible within module implementations.

3 Discussion

Three separate issues are raised:

3.1 User visibility

That VC makes module core's exports visible in the user TU, without an explicit import is an implementation bug, confirmed by GDR[1]. The proposal notes:

> [dcl.module.export,7.7.3]/1 … [Note: A module interface unit (for a module M) containing an import-declaration does not make the imported names transitively visible to translation units importing the module M.]

1 Gaby dos Reis

As notes are non-normative, the note itself implies the non-transitivity is deducible. That can be deduced due to absence of wording specifying that names introduced via an *import-declaration* within the purview of a imported module interface are made visible to importers of that module.

Note this non-transitivity at the user-level is distinct from compiler implementation details that most probably do require the complete graph of module interface units available in compiled form.

3.2 Implementation unit visibility

Whether a module implementation implicitly sees all the imports of its interface is not clear. The proposal states:

> [basic.scope.namespace,6.3.6]/1 … If the name X of a namespace member is declared in a *namespace-definition* of a namespace N in the module interface unit of a module M, the potential scope of X includes the *namespace-definition*s of N in every module unit of M and, if the name X is exported, in every translation unit that imports M.

and

> [dcl.module.interface,10.7.1]/1 … All entities with linkage other than internal linkage declared in a module interface unit of a module M are visible to all module units of M. …

However, here we are considering a *module-import-declaration*, which makes sets of names visible in arbitrary namespace partitions. It does not itself declare any new entities:

> [dcl.module.import,10.7.2]/1 … [*Note:* The entities are not redeclared in the translation unit containing the *module-import-declaration*. — *end note*]

Other text clarifies that an exported *module-import-declaration* makes names visible in imports:

> [dcl.module.export,10.7.3]/1 An exported *module-import-declaration* nominating a module `M′` in the purview of a module `M` makes all exported names of `M′` visible to any translation unit importing `M`.

However, a module implementation does not import, in the normal sense, its own interface:

> [dcl.module.import,10.7.2]/2 A module `M1` has a dependency on a module `M2` if any module unit of `M1` contains a *module-import-declaration* nominating `M2`. A module shall not have a dependency on itself.

There does not appear to be text saying that a module implementation sees the imports of its interface.

However, such visibility was intended in the original design. An implementation unit has visibility to all the module-linkage declarations of the interface unit. It is a natural extension of that rule for implementaion units to have visibility of imported entities.

3.3 Transitivity of *module-export-declaration*

There does not appear to be text saying that an exported *module-import-declaration* transitively makes names visible via an indirect exported *module-import-declaration* to final importers:

> [dcl.module.export,10.7.3]/1 … [*Note:* A module interface unit (for a module M) containing an *module-import-declaration* does not make the imported names transitively visible to translation units importing the module M. – *end note*]

The intent of the design is that non-exported *module-import-declaration*s are not made transitively visible.

3.4 Visibility of interface's global module partition

Module implementation units do not have visibility of entities declared in the global module portion of their interface unit's translation unit. This suggests they also do not have visibility of names made visible via import declarations within that portion. It is not explicitly specified what happens if the interface re-imports a module within its purview. Presumably the names are now visible within the interface's global module fragment and within its purview. Thus imports within a single translation unit might not be idempotent with regards to their cross-module unit behaviour.

4 Proposal

I think an ambiguity arises from the specification that it is declarations that make the declared entities visible (and possibly exported), but that imports do not (re)declare entities. Further, editorial changes since the original proposal have added unintended ambiguity and behaviour.

The module proposal's original intent is that:

- A module implementation has visibility to all the non-internal linkage names that are visible in the purview of its interface.
- A module implementation does not have visibility of names made visible in the global module fragment of its interface.
- An exported *module-import-declaration* naming M1 in module M2 makes the names made visible by module M1 visible to importers of module M2 as-if M1 were explicitly imported by any importers of M2.

The following wording edits define and clarify these semantics.

4.1 Changes to dcl.module.interface [10.7.1]

Amend the note in [dcl.module.interface]/1 to:

- State that implementation units see all names made visible in the interface.
- Note that implementation units do not see the interface's global module fragment:

> … The names of all entities in the interface of a module are visible to any translation unit importing that module. The entity and the declaration introduced by an *export-declaration* are said to be *exported*. ~~All entities~~ **Every name of an entity** with linkage other than internal linkage ~~declared~~**made visible** in the purview of the module interface unit of a module M ~~are~~**is** visible in the purview of all module implementation units of M. The entity and the declaration introduced by an *export-declaration* are said to be *exported*.**[*Note:* Names of entities made visible in the global module of an interface unit translation unit are not visible to module implementation units. – *end note*]**

The second and third sentences of the quoted fragment are swapped, as the now-second sentence is related to the first sentence. The now-third sentence discusses visibility, not declarations. The new note makes it clear that the global module is not made visible to implementation units.

4.2 Changes to dcl.module.export [10.7.3]

Amend [dcl.module.export,10.7.3]/1 to:

- Clarify names exported by a module in an exported module-import-declaration are exported.
- State that mixing exported and non-exported module imports is valid.
- Rename M' and M to M1 and M2 respectively:

> 1 An exported *module-import-declaration* nominating a module ~~`M'`~~**`M2`** in the purview of a module **interface unit** `M1` makes all **names** exported **by**~~namesof~~ ~~`M'`~~**`M2`, including those made visible in `M2` via its own exported *module-import-declarations*,** visible to any translation unit importing `M1`. **A module may be named in multiple exported and non-exported *module-import-declarations*.** [*Note:* A ~~module interface unit (for a module `M`) containing a~~ non-exported *module-import-declaration* **nominating M4 in interface unit M3** does not make the ~~imported~~ names **exported by `M4`** ~~of~~ transitively visible to translation units importing the module `M3`. — *end note*]

The first sentence is extended to make it clear that exported imports are made transitively visible. The new second sentence clarifies that mixing exported and non-exported imports is well formed. The note sentence is reordered to match the structure of the first sentence. The renaming of `M'` and `M` matches other parts of the TS where multiple modules are discussed.

5 Acknowledgements

I would like to thank Boris Kolpackov <boris@codesynthesis.com> for bringing this to my attention and experimenting with various implementations. I thank Gaby dos Reis <gdr@microsoft.com> and Gor Nishanov <gorn@microsoft.com> for drafting review.