Document Number: P0775r0
Date: 2017-10-03
To: SC22/WG21 CWG/EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: Working Draft, Extensions to C ++ for Modules, n4681

Module Partitions

Nathan Sidwell

The current wording of n4681 requires a module interface to be a single translation unit. When one considers a module may itself be a collection of many sub-components, this requirement is restricting, and not alleviated by composing a module of sub-modules. An extension to the concept of *module-name* is proposed, which permits partitioning a module interface and implementation.

1 Background

The current modules draft, n4681, defines a module name as a dotted sequence of identifiers. Although such a syntax suggests a modular hierarchy, no such concept exists. Module names reside in an entirely flat namespace, each independent of one another.

Modules are expected to aid construction of complex software, and the natural partitioning of a library into separate sub-modules would be expected to function. However, it does not fully satisfy the need of constructing a module from a set of mutually-dependent components.

1.1 Motivating Example

Consider a hypothetical module implementing a compilation system. Such a module will need objects to represent ‘declarations’ and objects to represent ‘types’. For a C++-like language, tagged-types will need to reference declarations and declarations will need to reference types. A typical implementation using existing C++ technology might consist of 2 header files:[1]

```
// decl.h
class Type; // forward declaration
class Decl {
  Type *type;  // type of the object
  …  // other members
};
```

1 Header protection is not shown.

```
// type.h
class Decl; // forward declaration
class Type {
  Decl name;  // name of the type
   … // other members
};

// compiler.h
#include <decl.h>
#include <type.h>

// decl.cc
#include <decl.h>
#include <type.h>
bool Decl::IsFunction () {
  return type->IsFunction ();
}
```

Converting this to modules requires could be done in two different ways.

1.2 Single Module

Placing the parts into a module interface unit could be done by textual inclusion, or preprocessor inclusion. Textual inclusion is likely to result in a large, possibly unwieldy, source file. Such a source file would have contention in a concurrent development environment.

Alternatively using a `#include` to incorporate them into a module interface leads one to develop them knowing the context they must be included from. These header files will no longer stand alone and be includeable from elsewhere. For one thing, they will probably be decorated with '`export`' keywords to expose the appropriate pieces of interface to the user. But more importantly, module-linkage entities will most likely be link-incompatible with external-linkage entities.[2]

1.3 Multiple Sub-modules

Using sub-modules for the separate parts of an interface would rely on forward-declaration via proclaimed-ownership-declarations. However, it will still have the following drawbacks:

1. Moving entities between sub-modules is a visible breaking change in module ownership.[3] This may be undesirable.

2 The proposed changes to the vendor-neutral C++ ABI insert an additional mangling component for module-linkage symbols that encodes the module name.

3 The amended ODR rules imply that symbols for exported entities could encode module ownership in some manner. The ABI changes used by Clang & GCC do not encode ownership for exports so that backwards compatibility to non-modular external symbols can be maintained.

2. There is no source code guidance that these particular set of sub-modules are only intended to be used together (via a container module), by the user. This is both a documentation problem, and an efficiency problem, as the compilation system would not automatically know to optimize the container module for its own inclusion.

For instance, presume the following module interfaces:[4]

```
// compiler.cc[5]
export module compiler;
export {
  import compiler.decl;
  import compiler.type;
}

// compiler-decl.cc
export module compiler.decl;
import compiler.type : extern class Type;
export class Decl { … };

// compiler-type.cc
export module compiler.type;
import compiler.decl : extern class Decl;
export class Type { … };
```

A user importing '`compiler`' would separately import '`compiler.decl`' and '`compiler.type`'. Unless instructed via implementation-specific command-line switches, it would not merge those two sub-modules by value into a `compiler` module.

2 Proposal

p0529r0 (2016-11-23, R.Smith) presented a scheme to specify partitions of a module interface. It added a new context-sensitive keyword, `partition`, and used an implementation-defined scheme to map *string-literal*s to the file system.

This proposal does not introduce new keywords, and provides a simple way to partition an interface into multiple translation units. This proposal does not suggest a *string-literal* module naming system. P0778r0 makes such a suggestion and the changes suggested here could be mapped onto that proposal.

The fundamental idea is to add structure to the *module-name* construct. This structure informs the compilation system of which parts of the name are to be used to designate module-ownership, and which parts provide partitioning of the module interface.

4 I am using the *proclaimed-ownership-declaration* syntax suggested in p0788r0.
5 I remain agnostic about particular file suffixes. Refer to p0778r0 for further discussion.

Rather than the module-name just being a dotted sequence of identifiers, a trailing `:identifier` may be present. This suffix denotes a *module partition*.

Coupled with this would be the following concepts:

1. The identifiers before the '`:`' specify the *base module name*.
2. The base module name defines the ownership of declarations with non-internal linkage.
3. A module-name lacking a partition component is an unpartitioned module.
4. A module-name with a partition component is a module partition.
5. A module partition may only be imported into modules with the same base module name.
6. The module interface unit (used by users of the module) does not use a partitioned name (this follows from Item 5).

These concepts make module partitions an implementation detail of any particular module. Entity declarations may be moved between partitions of the same module without breaking ABI compatibility. Whether the compiler artifact emitted by compiling an unpartitioned module interface unit includes the module partitions by reference, or value, is an implementation choice. Importing by value is likely to lead to compilation performance improvements as end users importing the module will then only read a single file.

With this, the above example could be modularized as follows:

```
// compiler-decl.cc
export module compiler:decl;
export class Type; // forward declaration
export class Decl {
  Type *type;  // type of the object
  …  // other members
};

// compiler-type.cc
export module compiler:type;
export class Decl; // forward declaration
export class Type {
  Decl name;  // name of the type
 … // other members
};

// compiler.cc
export module compiler;
export {
  import :decl;
```

```
  import :type;
}

// compiler-decl-impl.cc
module compiler:decl;
import :type;
bool Decl::IsFunction () {
  return type->IsFunction ();
}
```

Notice it is syntactically impossible to import a partition of a foreign module. There would no longer appear a need for *proclaimed-ownership-declarations*.

3 Changes to Modules-TS Draft

Amend the additional bullet for [basic,6]/8 as:

– they are *module-names* composed of the same dotted sequence of *identifiers* **and *module-partition*, if any**.

Amend [basic.def.odr,6.2]/5:[6]

5 Exactly one definition**, or visible defining import,** of a class is required in a translation unit if the class is used in a way that requires the class type to be complete. …

Change the additional seventh bullet added to [basic.def.odr,6.2]/6:

– if a declaration of `D` ~~that is not a *proclaimed-ownership-declaration*~~ appears in the purview of a module (10.7), all other ~~such~~ declarations **of `D`** shall appear in the purview of the same **base** module and there can be at most one definition of `D` in the **purview of** owning **base** module.

Change the amendments to [basic.scope.namespace,6.3.6]/1 as:

… follows the member's point of declaration. If the name *X* of a namespace member **with non-internal-linkage**[7] is declared in a *namespace-definition* of a namespace *N* in the module interface unit of ~~a~~ module *M***, or a module interface unit partition *MP***, the potential scope of *X* includes the *namespace-definitions* of *N* in every module unit of *M* ***or module partition unit MP respectively and every module or module partition unit that imports MP*** and, if the name *X* is exported, in every translation unit that imports *M*.
[*Example:* …

Modify the bullet added to [basic.lookup.argdep,6.4.2]/4:

6 This appears to be an existing defect, orthogonal to module partitions.
7 The current wording appears to make internal-linkage names visible in module implementation units. This is believed an error.

– Any function or function template that is owned by a module `M` **or base-module `BM`**, other than the global module (10.7), that is declared in the module interface unit of `M` **or an imported partition of `BM`**, and that has the same innermost enclosing non-inline namespace as some entity owned by `M` **or `BM`** in the set of associated entities, is visible within its namespace even if it is not exported.

Change the grammar added to 'program and linkage [basic.link,6.5]/1' as follows:

toplevel-declaration:
module-declaration
~~*proclaimed-ownership-declaration*~~
declaration

module-declaration:
`export`$_{opt}$ `module` *module-name attribute-specifier-seq*$_{opt}$ `;`

~~*proclaimed-owernship-declaration*~~
~~`extern module` *module-name* `;`~~

module-name:
base-module-name ~~*qualifier-seq*$_{opt}$ *identifier*~~ ***module-partition***$_{opt}$

base-module-name:
module-name-qualifier-seq$_{opt}$ ***identifier***

module-name-qualifier-seq:
~~*module-name-qualifier* `.`~~
module-name-qualifier-seq$_{opt}$ *identifier* `.`

~~*module-name-qualifier*~~
~~*identifier*~~

module-partition:
`:` identifier

Amend the new bullet to:

– When a name has *module linkage*, the entity it denotes is owned by a **base** module `M` and can be referred to by name from other scopes of the same **base** module ~~unit~~ (10.7) ~~or from scopes of other module units of `M`~~.

Modify [basic.link,6.5]/6:

… the block scope declaration declares that same entity and receives the linkage of the previous declaration. If that entity was ~~exported by an~~ imported **from a different *base-module***, the program is ill-formed. If there is more …

Modify the grammar changes in [dcl.dcl,10]/1:

...

module-import-declaration:
`import` ***base-****module-name attribute-specifier-seq*$_{opt}$ `;`
`import` ***module-partition attribute-specifier-seq***$_{opt}$ **`;`**

In [dcl.module,10.7] amend the new paragraph 4:

4 **A *module partition* is a module unit whose name contains a *module-partition*.** A *namespace-scope* declaration `D` of an entity ~~(other than a module)~~[8] in the purview of a module **or module partition** `M` is said to be owned by **base module** **`B`**`M`. Equivalently, the **base** module **`B`**`M` is the owning **base-**module of `D`.

In [dcl.module.interface,10.7.1] amend paragraph 1:

… All entities with linkage other than internal linkage declared in the purview of the module interface unit of a module `M` are visible in the purview of all module implementation units of `M` **and all module units of the same base module that directly or indirectly import `M`**. The entity and the declaration introduced by an export-declaration are said to be *exported*.

In [dcl.module.import,10.7.2] amend paragraph 1:

1 A *module-import-declaration* shall appear only at global scope. A *module-import-declaration* makes exported declarations from the interface of the nominated module visible to name lookup in the current translation unit, in the same namespaces and contexts as in the nominated module. **A *module-import-declaration* naming a *module-partition* names a partition of the current base module.** …

Modify [dcl.module.export,10.7.3]/1:

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`. **A module partition shall not export an imported partition.** [*Note:* A module interface unit (for a module `M`) containing a non-exported module-import-declaration does not make the imported names transitively visible to translation units importing the module `M` — *end note*]

Delete [dcl.module.proclaim,10.7.4]

8 Also noted in p0774r0, this appears to be superfluous wording, regardless of the concept of *module-partitions.*