Single-file modules with the Atom semantic properties rule

Document #:	P1242R1
Date:	2018-12-01
Project:	Programming Language C++
Audience:	Evolution

Reply-to:	Jeff Snyder <jeff-isocpp@caffeinated.me.uk>
	Richard Smith <richard@metafoo.co.uk>

1 TL;DR

Make the primary module interface unit consist of two parts: an interface part, and an implementation part. These parts would be separated by some syntactic marker, e.g. "`module :private;`"

2 Introduction

While merging the Atom proposal[P0971R1] with the Modules TS[N4720], a contention was discovered between the Modules TS's goal of allowing modules to be defined in a single file and Atom's rule for determining what semantic properties are exported.

The Atom rule for semantic properties of exported declarations[P0986R0, §3.1] is simpler than the rule in the Modules TS, and makes module interface units more robust to refactoring. However, it removes the ability to do certain things, such as defining a type but exporting it as incomplete, without using multiple files. The current proposal for merging modules into C++20[P1103R1] uses the Atom semantic properties rule.

The Atom comparison paper suggests[P0986R0, §3.2] using a module implementation partition to acheive this, e.g.

Implementation partition:

```
module m:impl;
struct s {};
```

Module interface:

```
export module m;
import :impl;
export using s_ptr = s*;
```

This paper proposes an alternative way to reconcile these aims: by allowing a module interface and a module implementation partition to co-exist in the same file.

3 Proposal

This paper proposes allowing a single *inline module implementation partition* to be included in the primary module interface unit. If the primary module interface unit includes an inline module implementation partition, it will appear after the interface itself and be separated from the interface by some syntatctic divider.

Even if an inline module implemenation partition is present, the term *primary module interface partition* refers only to the interface section of the primary module interface unit, i.e. the section preceeding the inline module implementation partition.

The structure of the primary module interface unit will then be as follows:

1. Module interface partition
2. Optional inline module implementation partition

When a the primary module interface partition is imported (either via import into a module implementation partition, or via implicit import into a module implementaiton unit), its corresponding inline module implementation partition, if present, is also imported.

4 Example

As a placeholder syntax, this paper uses "`module :private;`" as the marker that divides the module interface from the inline implementation partition.

Using inline module implementation partitions, the example from the introduction would be written as follows:

```
export module m;
struct s;
export using s_ptr = s*;

module :private;
struct s{};
```

5 Marker Syntax

The syntax "`module :private;`" was chosen as the palceholder syntax because it resembeles the start of a module implementation partition (whilst omitting the module name becuase re-specifying in the same file would be redundant), uses a keyword for the partition name to avoid conflicts with other module paritions, and can be identified by tools without fully parsing the module unit.

Other candidates for the syntax of the marker include:

- `module M:private;`
- `module M:;`
- `module M;`
- `module :;`
- `module :private;`
- `module private;`
- `module;`
- `export module;`
- `export;`
- `do export module;`
- `private module;`
- `not module;`
- `import module;`
- `inline module :private;`
- `inline module;`

6 Wording

Change the [**basic.link**] grammar as follows:

translation-unit:
preamble$_{opt}$ *declaration-seq*$_{opt}$ *private-module-fragment*$_{opt}$

private-module-fragment:
`module : private ;` *declaration-seq*$_{opt}$

Add a paragraph in [**basic.link**]:

A *private-module-fragment* shall only appear in a primary module interface unit ([dcl.module.unit]).

Change paragraph **[dcl.inline]p7** as follows:

An exported inline function or variable shall be defined in the translation unit containing its exported declaration, outside the *private-module-fragment* (if any).

Change paragraph **[dcl.module.interface]p1** as follows:

An *export-declaration* shall only appear at namespace scope and only in the purview of a module interface unit. An *export-declaration* shall not appear directly or indirectly within an unnamed namespace or a *private-module-fragment*.

Change paragraph **[dcl.module.global]p5** bullets 1-3 as follows:

- [...] whose point of instantiation is in M and is not within a *private-module-fragment*
- [...] whose point of instantiation is in M and is not within a *private-module-fragment*
- [...] lookup result for a dependent name that appears in M and not within a *private-module-fragment* (12.7.4)

Change paragraph **[dcl.module.context]p3** as follows:

if the template is defined in a module interface unit of a module M and the point of instantiation is not in a module interface unit of M, the point at the end of the *declaration-seq* of the primary module interface unit of M (prior to the *private-module-fragment*, if one is present).

Change paragraph **[dcl.module.reach]p3** bullet 2 as follows:

[...] it is not discarded (9.11.4), either does not appear within a *private-module-fragment* or appears in a *private-module-fragment* of the module containing the program point, and appears in a translation unit that is reachable from that program point.

Change paragraph **[temp.point]p8** as follows:

in addition to the points of instantiation described above, for any such specialization that has a point of instantiation within the *declaration-seq* of the translation unit, prior to the *private-module-fragment* (if any), the point after the *declaration-seq* of the *translation-unit* is also considered a point of instantiation, and for any such specialization that has a point of instantiation within the *private-module-fragment*, the end of the translation unit is also considered a point of instantiation.

References

[P0971R1] Richard Smith. Another take on modules. Proposal P0971R1, ISO/IEC JTC1/SC22/WG21, March 2018.

[N4720] Gabriel Dos Reis. Working Draft, Extensions to C++ for Modules. Working Draft N4720, ISO/IEC JTC1/SC22/WG21, January 2018.

[P0986R0] Richard Smith & David Jones. Comparison of modules proposals. Proposal P0986R0, ISO/IEC JTC1/SC22/WG21, March 2018.

[P1103R1] Richard Smith. Merging Modules. Proposal P1103R1, ISO/IEC JTC1/SC22/WG21, October 2018.