Single-file modules with the Atom semantic properties rule

Document #:	P1242R0
Date:	2018-10-08
Project:	Programming Language C++
Audience:	Evolution
Reply-to:	Jeff Snyder <jeff-isocpp@caffeinated.me.uk>

1 TL;DR

Make module interface units 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 each module interface unit. If a 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 existing term *module interface partition* refers only to the interface section of a module interface unit, i.e. the section preceeding the inline module implementation partition.

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

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

Furthermore, the module interface section of the primary module interface unit should be known as the *primary module interface partition*.

When a module interface partition is imported (including via implicit import of the primary module interface partition), its corresponding inline module implementation partition, if present, is also imported.

Inline module implementation units are never re-exported.

4 Examples

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{};
```

Whilst the primary use case for inline module implementation partitions is for writing single source file modules, they can also be used in module interface units other than the primary module interface unit, e.g.

Module interface unit for partition `m:impl`:

```
export module m:impl;
struct s;
s* f();

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

Primary module interface unit for `m`:

```
export module m;
export import :impl;
export void g()
{
    s var;                  // OK, s is complete
}
```

Translation unit using `m`:

```
import m;
s var;                  // ill-formed, s is incomplete
s* s_ptr = f();         // OK
auto sz = sizeof(*f()); // ill-formed, s is incomplete
```

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;`

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.