Reply-To: `gdr@microsoft.com`

Resolved Module TS (N4610) Issues

Gabriel Dos Reis

1. export import M; [Richard Smith; Sep 7, 2016]

Problem

Allow “`export import M;`” and “`export { import M; }`”, with the semantics of “`export module M;`” in the current draft, and remove syntactic support for “`export module M;`”.

Resolution

Change the example in 3.1/2 to read:

```
    export ~~module~~ import std.random;
```

Modify the production *module-export-declaration* in 3.5/1 as follows:

module-export-declaration:
  `export` ~~*module-declaration*~~ *module-import-declaration*

Remove the alternative *module-declaration* from *fragment:*

fragment:
  ~~*module-declaration*~~
  module-import-declaration
  declaration

2. import M; at interface level [Richard Smith; Sep 7, 2016]

Problem

Ban from interface unit of M.

Resolution

Add new paragraph 7.7.2/2:

A module M1 *has a dependency* on a module M2 if any module unit of M1 contains an *import-declaration* nominating M2. A module shall not have a dependency on itself. [*Example*:

```
module M;
import M;             // error: cannot import M in its own unit.
```

--end example]

Add new paragraph 7.7.2/3

> A module M1 *has an interface dependency* on a module M2 if the module interface unit of M1 contains an *import-declaration* nominating M2. A module shall not have a transitive interface dependency on itself. [*Example*:
>
> ```
> // interface unit of M1
> module M1;
> import M2;
> export struct A { };
>
> // interface unit of  M2
> module M2;
> import M3;
>
> // interface unit of M3
> module M3;
> import M1;     // error: cyclic interface dependency M1 -> M2 -> M3 -> M1.
> ```
>
> *--end example*]

3. export const int n = 5; [Richard Smith; Sep 8, 2016]

Problem

Clarify that this is allowed.

Resolution

Modify bullet (3.2) of paragraph 3.5/3 as follows:

> -- a non-inline non-exported variable of non-volatile const-qualified type that is neither explicitly declared extern nor previously declared to have external linkage; or

4. Import declaration and namespace partitions [Lukasz Mendakiewicz; Nov 3, 2016]

Problem

> I was reading N4610 and have a question:
>
> ```
> module M;
> export namespace N
> {
> ```

```
    struct A {};
  }
  namespace N
  {
    struct B {};
  }
```

7.7.1/4 says that all members of **namespace-body** are exported, meaning N::A.

```
   import M;
```

7.7.2/1 says that import declaration adds the **namespace partitions** with external linkage from M to the current TU.
Namespace partition N from M contains both N::A and N::B.

So is N::B visible and can be used in the second TU or not?

Resolution

Only exported declarations from the namespace partitions are meant to be made visible.

Remove the following sentences from 7.3/1

> A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units. ~~A namespace partition is the collection of all the namespace-definition s of the same namespace in a translation unit. A namespace consists of all its namespace partitions.~~ A namespace with external linkage is always exported regardless of whether any of its namespace-definition is introduced by export.

Rewrite paragraph 7.7.2/1 as follows:

> An *import-declaration* makes exported declarations ~~adds the namespace partitions with external linkage~~ from the interface of the nominated module visible to name lookup in ~~to the list of namespace partitions of~~ the current translation unit, ~~thereby making visible,~~ in the same namespaces and contexts as in the nominated module~~, to name lookup~~ ~~the declarations in the interface of the nominated module~~. [*Note*: The entities are not redeclared in the translation unit containing the *import-declaration*. — *end note*] [*Example*:

```
// Interface unit of M
module M;
export namespace N {
     struct A { };
}
namespace N {
    struct B { };
    export struct C{
         friend void f(C) { }       // exported, visible only through
argument-dependent lookup
    };
}
// Translation unit 2
import M;
N::A a { };        // OK.
N::B b { };        // error: ‘B’ not found in N.
void h(N::C c) {
    f(c);          // OK: N::f  found via argument-dependent lookup.
    N::f(c);       // error: ‘f’ not found via qualified lookup in N.
}
```

--*end example*]