Document Number: N4681
Date: 2017-07-14
Revises: N4667
Reply to: Gabriel Dos Reis
Microsoft
gdr@microsoft.com

Working Draft, Extensions to C++ for Modules

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

List of Tables

1 Scope [intro.scope]

1 This Technical Specification describes extensions to the C++ Programming Language (2) that introduce modules, a functionality for designating a set of translation units by symbolic name and ability to express symbolic dependency on modules, and to define interfaces of modules. These extensions include new syntactic forms and modifications to existing language semantics.

2 The International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use this color to represent added text and ~~strikethrough~~ to represent deleted text.

2 Normative references [intro.refs]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — N4660, *Working Draft, Standard for Programming Language C++*

N4660 is hereafter called the *C++ Standard*. The numbering of Clauses, sections, and paragraphs in this document reflects the numbering in the C++ Standard. References to Clauses and sections not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

3 Terms and definitions [intro.defs]

4 General [intro]

4.1 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in 4.1 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Acknowledgments [intro.ack]

1 This specification is based, in part, on the design and implementation described in the paper P0142R0 "*A Module System for C++*".

5 Lexical Conventions [lex]

5.11 Keywords [lex.key]

In 5.11, add these two keywords to Table 3 in paragraph 5.11/1: module and import.

6 Basic concepts [basic]

Modify paragraph 6/3 as follows:

3 An *entity* is a value, object, reference, function, enumerator, type, class member, bit-field, template, template specialization, namespace, module, parameter pack, or `this`.

Modify paragraph 6/4 as follows:

4 A *name* is a use of an *identifier* (5.10), *operator-function-id* (16.5), *literal-operator-id* (16.5.8), *conversion-function-id* (15.3.2), ~~or~~ *template-id* (17.2), or *module-name* (10.7) that denotes an entity or *label* (9.6.4, 9.1).

Add a sixth bullet to paragraph 6/8 as follows:

– they are *module-name*s composed of the same dotted sequence of *identifier*s.

6.1 Declarations and definitions [basic.def]

Append the following phrase to paragraph 6.1/2:

, or a *module-declaration*, or a *module-import-declaration*, or a *proclaimed-ownership-declaration*. [*Example:*

```
import std.io;                // make names from std.io available
export module M;              // declare module M
export struct Point {         // define and export Point
  int x;
  int y;
};
```

— end example]

6.2 One-definition rule [basic.def.odr]

Modify bullet (3.2) of paragraph 6.5/3 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

Add a seventh bullet to 6.2/6 as follows:

– 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 shall appear in the purview of the same module and there can be at most one definition of `D` in the owning module.

The purpose of this requirement is to implement module ownership of declarations.

6.3 Scope [basic.scope]

6.3.2 Point of declaration [basic.scope.pdecl]

Add a new paragraph 6.3.2/13 as follows:

13 The point of declaration of a module is immediately after the *module-name* in a *module-declaration*.

6.3.6 Namespace scope [basic.scope.namespace]

From end-user perspective, there are really no new lookup rules to learn. The "old" rules are the "new" rules, with appropriate adjustment in the definition of "associated entities."

Modify paragraph 6.3.6/1 as follows:

1 The declarative region of a *namespace-definition* is its *namespace-body*. Entities declared in a *namespace-body* are said to be members of the namespace, and names introduced by these declarations into the declarative region of the namespace are said to be *member names* of the namespace. A namespace member name has namespace scope. Its potential scope includes its namespace from the name's point of declaration (6.3.2) onwards; and for each *using-directive* (10.3.4) that nominates the member's namespace, the member's potential scope includes that portion of the potential scope of the *using-directive* that follows the member's point of declaration. 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-definitions* of N in every module unit of M and, if the name X is exported, in every translation unit that imports M. [*Example:*

```
// Translation unit #1
export module M;
export int sq(int i) { return i*i; }

// Translation #2
import M;
int main() { return sq(9); }        // OK: 'sq' from module M
```

— *end example*]

6.4 Name lookup [basic.lookup]

6.4.2 Argument-dependent name lookup [basic.lookup.argdep]

Modify paragraph 6.4.2/2 as follows:

2 For each argument type `T` in the function call, there is a set of zero or more *associated namespaces* (10.3) and a set of zero or more *associated* ~~*classes*~~ *entities* (other than namespaces) to be considered. The sets of namespaces and ~~classes~~ entities are determined entirely by the types of the function arguments (and the namespace of any template template argument). Typedef names and *using-declarations* used to specify the types do not contribute to this set. The sets of namespaces and ~~classes~~ entities are determined in the following way:

- If `T` is a fundamental type, its associated sets of namespaces and ~~classes~~ entities are both empty.
- If `T` is a class type (including unions), its associated ~~classes~~ entities are the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the innermost enclosing namespaces of its associated ~~classes~~ entities. Furthermore, if `T` is a class template specialization, its associated namespaces and ~~classes~~ entities also include: the namespace and ~~classes~~ entities associated with the types of the template arguments provided for template type parameters (excluding template template arguments); the templates used as template template arguments; the namespaces of which any template template arguments are members; and the classes of which any member template used as template template arguments are members. [*Note:* Non-type template arguments do not contribute to the set of associated namespaces. — *end note*]
- If `T` is an enumeration type, its associated namespace is the innermost enclosing namespace of its declaration, and its associated entities are `T`, and, if~~. If~~ it is a class member, ~~its associated class is~~ the member's class~~; else it has no associated class~~.

— If `T` is a pointer to `U` or an array of `U`, its associated namespaces and ~~classes~~ entities are those associated with `U`.

— If `T` is a function type, its associated namespaces and ~~classes~~ entities are those associated with the function parameter types and those associated with the return type.

— If `T` is a pointer to a data member of class `X`, its associated namespaces and ~~classes~~ entities are those associated with the member type together with those associated with `X`.

If an associated namespace is an inline namespace (10.3.1), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated ~~classes~~ entities and namespaces are the union of those associated with each of the members of the set, i.e., the ~~classes~~ entities and namespaces associated with its parameter types and return type. Additionally, if the aforementioned set of overloaded functions is named with a *template-id*, its associated ~~classes~~ entities and namespaces also include those of its type *template-arguments* and its template *template-arguments*. [*Example:*

```
// Header file X.h
namespace Q {
   struct X { };
}

// Interface unit of M1
#include "H.h"        // global module
namespace Q {
   void g_impl(X, X);
}
export module M1;
export template<typename T>
void g1(T t) {
   g_impl(t, Q::X{ });   // #1
}
export template<typename T>
void g2(T t) {
   using Q::g_impl;
   g_impl(t, Q::X{ });   // #2
}
void j(Q::X x) {
   g1(x);                // OK: g_impl found at #1
   g2(x);                // OK: g_impl found at #2
}

// Interface unit of M2
#include "X.h"
import M1;
export module M2;
void h(Q::X x) {
   g1(x);                // ill-formed: g_impl not found at #1
   g2(x);                // OK: g_impl found at #2
}
```

—*end example*]

Modify paragraph 6.4.2/4 as follows:

4 When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (6.4.3.2) except that:

- — Any *using-directive*s in the associated namespace are ignored.
- — Any namespace-scope friend declaration functions or friend function templates declared in ~~associated~~ classes in the set of associated entities are visible within their respective namespaces even if they are not visible during an ordinary lookup (14.3).
- — All names except those of (possibly overloaded) functions and function templates are ignored.
- — Any function or function template that is owned by a module `M` other than the global module (10.7), that is declared in the module interface unit of `M`, and that has the same innermost enclosing non-inline namespace as some entity owned by `M` in the set of associated entities, is visible within its namespace even if it is not exported.

6.5 Program and linkage [basic.link]

Change the definition of *translation-unit* in paragraph 6.5/1 to:

translation-unit
  toplevel-declaration-seq$_{opt}$

toplevel-declaration-seq
  toplevel-declaration
  toplevel-declaration-seq toplevel-declaration

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* `:` *declaration*

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

module-name-qualifier-seq
  module-name-qualifier `.`
  module-name-qualifier-seq identifier `.`

module-name-qualifier
  identifier

Insert a new bullet between first and second bullet of paragraph 6.5/2:

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

Modify 6.5/6 as follows:

6 The name of a function declared in block scope and the name of a variable declared by a block scope extern declaration have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If that entity was exported by an imported module, the program is ill-formed. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage and is owned by the global module.

Insert a new paragraph before paragraph 6.5/8

> A name declared at namespace scope in the purview of a module that does not have internal linkage by the previous rules and that is introduced by a non-exported declaration (10.7.1) has module linkage. The name of any class member where the enclosing class has a name with module linkage also has module linkage.

10 Declarations [dcl.dcl]

Add a new alternative to *declaration* in paragraph 10/1 as follows

> *declaration* :
> *block-declaration*
> *nodeclspec-function-declaration*
> *function-definition*
> *template-declaration*
> *explicit-instantiation*
> *explicit-specialization*
> *linkage-specification*
> *namespace-definition*
> *empty-declaration*
> *attribute-declaration*
> *export-declaration*
> *module-import-declaration*
>
> *export-declaration* :
> `export` *declaration*
> `export {` *declaration-seq* $_{opt}$ `}`
>
> *module-import-declaration* :
> `import` *module-name attribute-specifier-seq*$_{opt}$ `;`

10.1 Specifiers [dcl.spec]

10.1.2 Function specifiers [dcl.fct.spec]

Add a new paragraph 10.1.2/7 as follows:

7 An exported inline function shall be defined in the same translation unit containing its export declaration. An exported inline function has the same address in each translation unit importing its owning module. [*Note:* There is no restriction on the linkage (or absence thereof) of entities that the function body of an exported inline function can reference. A constexpr function is implicitly inline. —*end note*]

10.3 Namespaces [basic.namespace]

Modify paragraph 10.3/1 as follows:

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 with external linkage is always exported regardless of whether any of its *namespace-definitions* is introduced by `export`. [*Note:* There is no way to define a namespace with module linkage. —*end note*][*Example:*

```
export module M;
namespace N {     // N has external linkage and is exported
}
```

— *end example*]

Add a new section 10.7 titled "**Modules**" as follows:

10.7 Modules [dcl.module]

1 A *module unit* is a translation unit that contains a *module-declaration*. A *named module* is the collection of module units with the same *module-name*. A translation unit may not contain more than one *module-declaration*. A *module-name* has external linkage but cannot be found by name lookup.

2 A *module interface unit* is a module unit whose *module-declaration* contains the `export` keyword; any other module unit is a *module implementation unit*. A named module shall contain exactly one module interface unit.

3 A *module unit purview* starts at the *module-declaration* and extends to the end of the translation unit. The *purview* of a named module `M` is the set of module unit purviews of `M`'s module units.

4 A namespace-scope declaration `D` of an entity (other than a module) in the purview of a module `M` is said to be *owned* by `M`. Equivalently, the module `M` is the *owning module* of `D`.

5 The *global module* is the collection of all declarations not in the purview of any *module-declaration*. By extension, such declarations are said to be in the purview of the global module. [*Note:* The global module has no name, no module interface unit, and is not introduced by any *module-declaration*. — *end note*]

6 A *module* is either a named module or the global module.

10.7.1 Export declaration [dcl.module.interface]

1 An *export-declaration* shall only appear in the purview of a module unit. An *export-declaration* does not establish a scope and shall not contain more than one `export` keyword. The *interface* of a module `M` is the set of all *export-declarations* in its purview. An *export-declaration* of the form

export *declaration*

shall declare at least one entity. The names of all entities in the interface of a module are visible to any translation unit importing that module. 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`. The entity and the declaration introduced by an *export-declaration* are said to be *exported*.

2 Every name introduced by an *export-declaration* shall have external linkage. If that declaration introduces an entity with a non-dependent type, then that type shall have external linkage or shall involve only types with external linkage. [*Example:*

```
export module M;
export static int n = 43;     // error: n has internal linkage
namespace {
  struct S { };
}
export void f(S);     // error: parameter type has internal linkage
struct T { };
export T id(T);       // OK
```

— *end example*]

3 If an *export-declaration* introduces a *namespace-definition*, then each member of the corresponding *namespace-body* is implicitly exported and subject to the rules of export declarations.

10.7.2 Import declaration [dcl.module.import]

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. [*Note:* The entities are not redeclared in the translation unit containing the *module-import-declaration*. — *end note*] [*Example:*

```
// Interface unit of M
export 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*]

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. [*Example:*

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

— *end example*]

3 A module `M1` *has an interface dependency* on a module `M2` if the module interface of `M1` contains a *module-import-declaration* nominating `M2`. A module shall not have a transitive interface dependency on itself. [*Example:*

```
// Interface unit of M1
export module M1;
import M2;
export struct A { };

// Interface unit of M2
export module M2;
import M3;

// Interface unit of M3
export module M3;
import M1;           // error: cyclic interface dependency M3 -> M1 -> M2 -> M3
```

— *end example*]

10.7.3 Module exportation [dcl.module.export]

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`. [*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*]

10.7.4 Proclaimed ownership declaration [dcl.module.proclaim]

1 A *proclaimed-ownership-declaration* asserts that the entities introduced by the declaration are exported by the nominated module. It shall not be a defining declaration.

2 The program is ill-formed, no diagnostic required, if the owning module in the *proclaimed-ownership-declaration* does not export the entities introduced by the declaration.

17 Templates [temp]

17.6 Name resolution [temp.res]

17.6.4 Dependent name resolution [temp.dep.res]

Add a new paragraphs to 17.6.4:

2 [*Example:*

```
// Header file X.h
struct X { /* ... */ };
X operator+(X, X);

// Module interface unit of F
module F;
export template<typename T>
void f(T t) {
    t + t;
}

// Module interface unit of M
#include "X.h"
import F;
module M;
void g(X x) {
   f(x);            // OK: instantiates f from F
}
```

— end example]

3 [*Note:* [*Example:*

```
// Module interface unit of A
module A;
export template<typename T>
void f(T t) {
   t + t;           // #1
}

// Module interface unit of B
module B;
import A;
export template<typename T, typename U>
void g(T t, U u) {
   f(t);
}

// Module interface unit of C
#include <string>           // not in the purview of C
import B;
module C;
```

```
export template<typename T>
void h(T t) {
   g(std::string{ }, t);
}

// Translation unit of main()
import C;
void i() {
   h(0);                    // ill-formed: '+' not found at #1
}
```

— *end example*]

This example is currently ill-formed by the current specification. It is an open question as to how often the scenario occurs in practice, and whether to make the example well-formed or whether additional syntax will be introduced that does not involve modifying the header. — *end note*]

17.6.4.1 Point of instantiation [temp.point]

Replace paragraph 17.6.4.1/7 as follows:

7 ~~The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.~~The instantiation context of an expression that depends on template arguments is the context of a lookup at the point of instantiation of the enclosing template.

17.7 Template instantiation and specialization [temp.spec]

Add new paragraphs to 17.7:

7 If the template argument list of the specialization of an exported template involves a non-exported entity, then the resulting specialization has module linkage and is owned by the module that contains the point of instantiation.

8 If all entities involved in the template-argument list of the specialization of an exported template are exported, then the resulting specialization has external linkage and is owned by the owning module of the template.