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 fomatting.

Contents

List of Tables

1   General   **[intro]**

1.1   Scope   **[intro.scope]**

1 This Technical Specification describes extensions to the C++ Programming Language (1.2) 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.

1.2   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)   — N4567, *Working Draft, Standard for Programming Language C++*

    N4567 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.

1.3   Implementation compliance   **[intro.compliance]**

1 Conformance requirements for this specification are the same as those defined in 1.3 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.4   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++*".

2 Lexical Conventions [lex]

2.11 Keywords [lex.key]

In 2.11, add these two keywords to Table 3 in paragraph 2.11/1: module and import.

3   Basic concepts   [basic]

Modify paragraph 3/3 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 3/4 as follows:

4   A *name* is a use of an *identifier* (2.10), *operator-function-id* (13.5), *literal-operator-id* (13.5.8), *conversion-function-id* (12.3.2), ~~or~~ *template-id* (14.2), or *module-name* (7.7) that denotes an entity or *label* (6.6.4, 6.1).

Add a sixth bullet to paragraph 3/8 as follows:

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

3.1   Declarations and definitions   [basic.def]

Append the following phrase to paragraph 3.1/2:

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

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

—*end example*]

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

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

Add a seventh bullet to 3.2/6 as follows:

　　– if a declaration of `D` that is not a *proclaimed-ownership-declaration* appears in the purview of a module (7.7), 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.

3.3   Scope   [basic.scope]
3.3.2   Point of declaration   [basic.scope.pdecl]

Add a new paragraph 3.3.2/13 as follows:

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

3.3.6 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 "namespace" which is now clarified as the collection of "namespace partitions".

Modify paragraph 3.3.6/1 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 (3.3.2) onwards; and for each *using-directive* (7.3.4) 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 partition (7.3) of a namespace N in the module interface unit of a module M, the potential scope of X includes the namespace partitions of N in every module unit of M and, if the name X is exported, in every translation unit that imports M. [*Example:*

```
// m-1.ixx
module M;
export int sq(int i) { return i*i; }

// m-2.cxx
import M;
int main() { return sq(9); }        // OK: 'sq' from module M
```

—*end example*]

3.5 Program and linkage [basic.link]

Change the definition of *translation-unit* in paragraph 3.5/1 to:

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

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

toplevel-declaration
    module-declaration
    module-export-declaration
    module-import-declaration
    exported-fragment-group
    proclaimed-ownership-declaration
    declaration

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

module-export-declaration
    export *module-import-declaration*

module-import-declaration
    import *module-name attribute-specifier-seq$_{opt}$* ;

exported-fragment-group
    export { *fragment-seq* }

fragment-seq
> *fragment*
> *fragment-seq fragment*

fragment
> *module-import-declaration*
> *declaration*

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 3.5/2:

> — When a name has *module linkage*, the entity it denotes is owned by a module `M` and can be referred to by names from other scopes of the same module unit (7.7) or from scopes of other module units part of `M`.

Modify 3.5/6 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 3.5/8

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

7 Declarations [dcl.dcl]

Add a new alternative to *declaration* in paragraph 7/1 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*
>
> *export-declaration* :
>   export *declaration*
>   export { *declaration-seq* $_{opt}$ }

7.1 Specifiers [dcl.spec]

7.1.2 Function specifiers [dcl.fct.spec]

Add a new paragraph 7.1.2/7 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*]

7.3 Namespaces [basic.namespace]

Modify paragraph 7.3/1 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-definition* is introduced by export. [*Note:* There is no way to define a namespace with module linkage. — *end note*][*Example:*
>
> ```
> module M;
> namespace N {     // N has external linkage and is exported
> }
> ```
>
> — *end example*]

Add a new section 7.7 titled "**Modules**" as follows:

7.7   Modules [dcl.module]

1   A *translation-unit* shall contain at most one *module-declaration* as a *toplevel-declaration*. A *module unit* is a *translation-unit* that contains a *module-declaration*. Such a translation unit is said to be part of the module designated by the *module-name*. A *module-name* has external linkage.

2   A *module* is a collection of module units, at most one of which contains *export-declaration* s or *exported-fragment-group* s or *module-export-declaration* s. Such a distinguished module unit is called the *module interface unit*. Any other module unit is called a *module implementation unit*.

3   A *module unit purview* starts at the *module-declaration* and extends to the end of the translation unit. The *purview* of a 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 and is not introduced by any *module-declaration*. — *end note*]

Add a new subsection 7.7.1 titled "**Export declaration**":

7.7.1   Export declaration [dcl.module.interface]

1   An *export-declaration* shall appear in the purview of a module other than the global module. It shall not contain more than one export keyword. The *interface* of a module M is the set of all *export-declaration* s in its purview. An *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 a module interface unit of a module M are visible to all module units of M. The entity and the declaration introduced by an *export-declaration* are said to be *exported*.

2   The name introduced by the declaration of 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:*

```
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
```

— *end example*]

3   In a *exported-fragment-group*, each *fragment* is processed as an exported declaration.

4   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.

Add a new subsection 7.7.2 titled "**Import declaration**":

7.7.2   Import declaration [dcl.module.import]

1   An *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 *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*]

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*]

3   A module `M1` *has an interface dependency* on a module `M2` if the module interface 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 M3 -> M1 -> M2 -> M3
```

—*end example*]

Add a new subsection 7.7.3 titled "**Module exportation**":

7.7.3   Module exportation   [dcl.module.export]

1   A *module-export-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 an *import-declaration* does not make the imported names transitively visible to translation units importing the module `M`. —*end note*]

Add a new section 7.7.4 titled "**Proclaimed ownership declaration**":

7.7.4   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   A 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.

14   Templates   [temp]

14.7   Template instantiation and specialization   [temp.spec]

Add new paragraphs to 14.7:

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.