Document Number: N4720
Date: 2018-01-29
Revises: N4689
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

1 Scope [intro.scope]

1 This Technical Specification describes extensions to the C++ Programming Language (Clause 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 14882:2017, 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 underlining 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) — ISO/IEC *14882:2017, Programming Languages – C++*

ISO/IEC 14882:2017 is hereafter called the *C++ Standard*. The numbering of clauses, subclauses, and paragraphs in this document reflects the numbering in the C++ Standard. References to clauses and subclauses not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

3 Terms and definitions [intro.defs]

No terms and definitions are listed in this document.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at `http://www.electropedia.org`

— ISO Online browsing platform: available at `http://www.iso.org/obp`

4 General [intro]

4.1 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are those defined in subclause 4.1 in the C++ Standard, except that references to the C++ Standard therein shall be taken as referring to the document that is the result of applying the editing instructions. Similarly, all references to the C++ Standard in the resulting document shall be taken as referring to the resulting document itself. [*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]

5.1 Separate translation [lex.separate]

Modify paragraph 5.1/2 as follows

2 [*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.5) by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (6.5). — *end note*]

5.2 Phases of translation [lex.phases]

Modify bullet 7 of paragraph 5.2/1 as follows:

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (17.2). — *end note*] It is implementation-defined whether the source for module interface units for modules on which the current translation unit has an interface dependency (10.7.3) is required to be available. [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]

Add new paragraphs as follows:

2 The result of processing a translation unit from phases 1 through 7 is a directed graph called the *abstract semantics graph* of the translation unit:

— Each vertex, called a *declset*, is a citation (10.7.3), or a collection of non-local declarations and redeclarations (Clause 10) declaring the same entity or other non-local declarations of the same name that do not declare an entity.

— A directed edge (D_1, D_2) exists in the graph if and only if the declarations contained in D_2 declare an entity mentioned in a declaration contained in D_1.

The *abstract semantics graph of a module* is the subgraph of the abstract semantics graph of its module interface unit generated by the declsets the declarations of which are in the purview of that module interface unit. [*Note:* The abstract semantics graphs of modules, as appropriately restricted (10.7.6), are used in the processing of *module-import-declarations* (10.7.3) and module implementation units. — *end note*]

3 An entity is *mentioned* in a declaration D if that entity is a member of the *basis* of D, a set of entities determined as follows:

— If D is a *namespace-definition*, the basis is the union of the bases of the *declarations* in its *namespace-body*.

— If D is a *nodeclspec-function-declaration*,

  — if D declares a contructor, the basis is the union of the type-bases of the parameter types

- — if D declares a conversion function, the basis is the type-basis of the return type
- — otherwise, the basis is empty.

— If D is a *function-definition*, the basis is the type-basis of the function's type

— If D is a *simple-declaration*

- — if D declares a *typedef-name*, the basis is the type-basis of the aliased type
- — if D declares a variable, the basis is the type-basis of the type of that variable
- — if D declares a function, the basis is the type-basis of the type of that function
- — if D defines a class type, the basis is the union of the type-bases of its direct base classes (if any), and the bases of its *member-declaration*s.
- — otherwise, the basis is the empty set.

— If D is a *template-declaration*, the basis is the union of the basis of its *declaration*, the set consisting of the entities (if any) designated by the default template template arguments, the default non-type template arguments, the type-bases of the default type template arguments. Furthermore, if D declares a partial specialization, the basis also includes the primary template.

— If D is an *explicit-instantiation* or an *explicit-specialization*, the basis includes the primary template, and all the entities in the basis of the *declaration* of D.

— If D is a *linkage-specification*, the basis is the union of all the bases of the *declaration*s contained in D.

— If D is a *namespace-alias-definition*, the basis is the singleton consisting of the namespace denoted by the *qualified-namespace-specifier*.

— If D is a *using-declaration*, the basis is the union of the bases of all the declarations introduced by the *using-declarator*.

— If D is a *using-directive*, the basis is the singleton consisting of the norminated namespace.

— If D is an *alias-declaration*, the basis is the type-basis of its *defining-type-id*.

— Otherwise, the basis is empty.

The *type-basis* of a type T is

— If T is a fundamental type, the type-basis is the empty set.

— If T is a cv-qualified type, the type-basis is the type-basis of the unqualified type.

— If T is a member of an unknown specialization, the type-basis is the type-basis of that specialization.

— If T is a class template specialization, the type-basis is the union of the set consisting of the primary template and the template template arguments (if any) and the non-dependent non-type template arguments (if any), and the type-bases of the type template arguments (if any).

— If T is a class type or an enumeration type, the type-basis is the singleton $\{T\}$.

— If T is a reference to U, or a pointer to U, or an array of U, the type-basis is the type-basis of U.

— If T is a function type, the type-basis is the union of the type-basis of the return type and the type-bases of the parameter types.

— If T is a pointer to data member of a class X, the type-basis is the union of the type-basis of X and the type-basis of member type.

— If T is a pointer to member function type of a class X, the type-basis is the union of the type-basis of X and the type-basis of the function type.

— Otherwise, the type-basis is the empty set.

4 [*Note:* The basis of a declaration includes neither non-fully evaluated expressions nor entities used in those expressions. [*Example:*

```
const int size = 2;
int ary1[size];                              // size not in ary1's basis
constexpr int identity(int x) { return x; }
int ary2[identity(2)];                       // identity not in ary2's basis

template<typename> struct S;
template<typename, int> struct S2;
constexpr int g(int);

template<typename T, int N>
S<S2<T, g(N)>> f();                          // f's basis: {S, S2}
```

— end example] *— end note*]

5.11 Keywords [lex.key]

In 5.11, add these two keywords to Table 5 in paragraph 5.11/1: module and import.

Modify note in paragraph 5.11/1 as follows:

1 ...
[*Note:* The ~~export and~~ `register` keyword~~s are~~ is unused but ~~are~~ is reserved for future use. *— end note*]

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, or parameter pack.

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]

Modify paragraph 6.1/1 as follows:

1 A declaration (Clause 10) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and ~~attributes~~semantic properties of these names. [...]

Append the following two bullets to paragraph 6.1/2:

2 A declaration is a *definition* unless

— ...

— it is an explicit specialization (17.7.3) whose *declaration* is not definition~~.~~,

— it is a *module-import-declaration*,

— it is a *proclaimed-ownership-declaration*.

[*Example:*

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

— end example]

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

Replace paragraph 6.2/1 with:

1 A variable, function, class type, enumeration type, or template shall not be defined where a prior definition is reachable (6.4).

Modify opening of paragraph 6.2/6 as follows

6 There can be more than one definition of a class type (Clause 12), enumeration type (10.2), inline function with external or module linkage (10.1.6), inline variable with external or module linkage (10.1.6), class template (Clause 17), non-static function template (17.5.6), static data member of a class template (17.5.1.3), member function of a class template (17.5.1.1), or template specialization for which some template parameters are not specified (17.7, 17.5.5) in a program provided that ~~each definition appears in a different translation unit~~ no prior definition is reachable (6.4) at the point where a definition appears, and provided the definitions satisfy the following requirements. For an entity with an exported declaration, there shall be only one definition of that entity; a diagnostic is required only if the abstract semantics graph of the module contains a definition of the entity. [*Note:* If the definition is not in the interface unit, then at most one module unit can have and make use of the definition. —*end note*] Given such an entity named `D` defined in more than one translation unit, then

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 a name X (not having internal linkage) is declared in a namespace N in the purview of the module interface unit of a module M, the potential scope of X includes the portion of the namespace N in the purview of every module implementation unit of M and, if the name X is exported, in every translation unit that imports M after a *module-import-declaration* nominating M. [*Example:*

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

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

—*end example*]

6.4 Name lookup [basic.lookup]

Modify paragraph 6.4/1 as follows:

1 The name lookup rules apply uniformly to all names (including *typedef-names* (10.1.3), *namespace-names* (10.3), and *class-names* (12.1)) wherever the grammar allows such names in the context

discussed by a particular rule. Name lookup associates the use of a name with a set of declarations (6.1) or citations (10.7.3) of that name. For all intent and purposes of further semantic processing requiring declarations, a citation is replaced with the declarations contained in its declset. [...] Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the ~~attributes~~semantic properties introduced by the name's declaration used further in the expression processing (Clause 8).

Add new paragraph 6.4/5 as follows:

5 A declaration is *reachable* from a program point if it can be found by unqualified name lookup in its scope.

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-declaration*s 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-argument*s and its template *template-argument*s.

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.

— In resolving dependent names (17.6.4), any function or function template that is owned by a named module `M` (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
  declaration

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

— When a name has *module linkage*, the entity it denotes can be referred to by names from other scopes of the same module unit (10.7.1) or from scopes of other module units of that same module.

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 or module linkage; or

Modify paragraph 6.5/4 as follows:

4 An unnamed namespace or a namespace declared directly or indirectly within an unnamed namespace has internal linkage. All other namespaces have external linkage. A name having namespace scope that has not been given internal linkage above ~~has the same linkage as the enclosing namespace if it~~and that is the name of

— a variable; or

— a function; or

— a named class (Clause 12), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (10.1.3); or

— a named enumeration (10.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (10.1.3); or

— a template~~.~~

has the same linkage as the enclosing namespace if

— said namespace has internal linkage, or

— the name is exported (10.7.2), or is declared in a *proclaimed-ownership-declaration*, or is not being declared in the purview of a named module (10.7.1);

otherwise, the name has module linkage.

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 or if the containing block scope is in the purview of a named 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.

Modify paragrapgh 6.5/9 as follows:

9 Two names that are the same (Clause 9) and that are declared in different scopes shall denote the same variable, function, type, template or namespace if

— both names have external or module linkage and are declared in declarations attached to the same module[1], or else both names have internal linkage and are declared in the same translation unit; and

— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

— when both names denote functions, the parameter-typelists of the functions (11.3.5) are identical; and

— when both names denote function templates, the signatures (17.5.6.1) ar the same.

If two declarations declaring entities (other than namespaces) and attached to different modules introduce two names that are the same and that both have external linkage, the program is ill-formed; no diagnostic required. [*Note: using-declarations*, typedef declarations, and *alias-declarations* do not declare entities, but merely introduce synonyms. Similarly, *using-directives* do not declare entities, either. — *end note*]

6.6 Start and termination [basic.start]

6.6.1 `main` function [basic.start.main]

Modify paragraph 6.6.1/1 as follows:

1 A program shall contain a global function called `main` declared in the purview of the global module.

1) This provision supports implementations where exported entities in different modules have different implementation symbols. Conversely, for other implementations, exported entities have the same implementation symbols regardless of in which module they are declared. Such implementations are supported for the time being by disallowing all situations where the same names with external linkage might appear from different modules.

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*
> *proclaimed-ownership-declaration*

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. [*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 (10.1.5) is implicitly inline. —*end note*]

10.1.6 The `inline` specifier [dcl.inline]

Modify paragraph 10.1.6/6 as follows

6 Some definition for ~~A~~an inline function or variable shall be ~~defined~~reachable in every translation unit in which it is odr-used and the function or variable shall have exactly the same definition in every case (6.5). [*Note:* A call to the inline function or a use of the inline variable may be encountered before its definition appears in the translation unit. —*end note*] If the definition of a function or variable appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function or variable with external or module linkage is ~~declared~~reachable via an inline declaration in one translation unit, it shall be ~~declared~~reachable via an inline declaration in all translation units in which it ~~appears~~is reachable; no diagnostic is required. An inline function or variable with external or module linkage shall have the same address in all translation units. [*Note:* A `static` local variable in an inline function with external or module linkage always refers to the same object. A type defined within the body of an inline function with external or module linkage is the same type in every translation unit. —*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*]

10.3.3 The `using` declaration [namespace.udecl]

Modify paragraph 10.3.3/1 as follows:

1 Each *using-declarator* in a *using-declaration* introduces a set of declarations and citations into the declarative region in which the *using-declaration* appears. The set of declarations and citations introduced by the *using-declarator* is found by performing qualified name lookup (6.4.3, 13.2) for the name in the *using-declarator*, excluding functions that are hidden as described below. If the *using-declarator* does not name a constructor, the *unqualified-id* is declared in the declarative region in which the *using-declaration* appears as a synonym for each declaration or citation introduced by the *using-declarator*. [...]

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

10.7 Modules [dcl.module]

10.7.1 Module units and purviews [dcl.module.unit]

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

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

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

module-name-qualifier:
  identifier

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 shall 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 The *global module* is the collection of all declarations not in the purview of any module. 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*]

5 A *module* is either a named module or the global module. A *proclaimed-ownership-declaration* is *attached* to the module it nominates; any other declaration is attached to the module in whose purview it appears.

6 For a namespace-scope declaration D of an entity (other than a namespace), if D is within a *proclaimed-ownership-declaration* for a module X, the entity is said to be *owned* by X. Otherwise, if D is the first declaration of that entity, then that entity is said to be *owned* by the module in whose purview D appears.

7 If a declaration attached to some module matches (according to the redeclaration rules) a reachable declaration from a different module, the program is ill-formed. [*Example:*

```
// module interface of M
int f();              // #1
int g();              // #2, owned by the global module
export module M;
export using ::f;     // OK: does not declare an entity
int g();              // error: matches #2, but appears in the purview of M
export int h();       // #3
export int k();       // #4

// other translation unit
import M;
static int h();       // error: matches #3
int k();              // error: matches #4
```

— *end example*]

8 The subgraph of the abstract semantics graph G of a module M generated by the nodes of G, excluding those introducing names with internal linkage, is available to name lookup in the purview of every module implementation unit of M. The declsets made available by the *module-import-declarations* in the purview of the module interface unit of M are also available to name lookup in the purview of all module implementation units of M.

10.7.2 Export declaration [dcl.module.interface]

export-declaration:
  `export` *declaration*
  `export {` *declaration-seq*$_{opt}$ `}`

1 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. An *exported-declaration* has the declarative effects of its *declaration* or its *declaration-seq* (if any). 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-declaration*s in its purview.

2 In an *export-declaration* of the form

  `export` *declaration*

the *declaration* shall be a *module-import-declaration*, or it shall declare at least one name, and if that declaration declares an entity, the *decl-specifier-seq* (if any) of the *declaration* shall not contain `static`. The *declaration* shall not be an *unnamed-namespace-definition* or a *proclaimed-ownership-declaration*. [*Example:*

```
export int x;                 // error: not in the purview of a module interface unit
export module M;
namespace {
  export int a;             // error: export within unnamed namespace
}
export static int b;        // error: b explicitly declared static.
export int f();             // OK
export namespace N { }      // OK
export using namespace N;   // error: does not declare a name
```

— *end example*]

If the *declaration* is a *using-declaration* (10.3.3), any entity to which the *using-declarator* ultimately refers shall have been introduced with a name having external linkage. [*Example:*

```
int f()                 // f has external linkage
export module M;
export using ::f;       // OK
struct S;
export using ::S;       // error: S has module linkage
namespace N {
  int h();
  static int h(int);    // #1
}
export using N::h;      // error: #1 has internal linkage
```

— *end example*]

[*Note:* Names introduced by `typedef` declarations are not so constrained. [*Example:*

```
export module M;
struct S;
export using T = S;     // OK: exports name T denoting type S
```

— *end example*] — *end note*]

3

An *export-declaration* of the form

`export {` *declaration-seq*$_{opt}$ `}`

is equivalent to a sequence of declarations formed by prefixing each *declaration* of the *declaration-seq* (if any) with `export`.

4 A namespace-scope or a class-scope declaration lexically contained in an *export-declaration*, as well as the entities and the names it introduces are said to be *exported*. The exported declarations in the interface of a module are reachable from any translation unit importing that module. [*Note:* Exported names have either external linkage or no linkage; see 6.5 — *end note*] [*Example:*

```
// Interface unit of M
export module M;
export struct X {
  void f();
  struct Y { };
};

namespace {
  struct S { };
}
```

```
export void f(S);    // OK
struct T { };
export T id(T);      // OK

export struct A;     // A exported as incomplete

export auto rootFinder(double a) {
  return [=](double x) { return (x + a/x)/2; };
}

export const int n = 5; // OK: n has external linkage

// Implementation unit of M
module M;
struct A {
  int value;
};

// main program
import M;
int main() {
  X{}.f();                  // OK: X and X::f are exported
  X::Y y;                   // OK: X::Y is exported as a complete type
  auto f = rootFinder(2); // OK
  return A{45}.value;       // error: A is incomplete
}
```

— end example]

5 [*Note:* Redeclaring a name in an *export-declaration* cannot change the linkage of the name (10.1.1). [*Example:*

```
// Interface unit of M
export module M;
static int f();             // #1
export int f();             // error: #1 gives internal linkage
struct S;                   // #2
export struct S;            // error: #2 gives module linkage
namespace {
  namespace N {
    extern int x;           // #3
  }
}
export int N::x;            // error: #3 gives internal linkage
```

— end example] *— end note*]

6 Declarations in an exported *namespace-definition* or in an exported *linkage-specification* (10.5) are implicitly exported and subject to the rules of exported declarations. [*Example:*

```
export module M;
export namespace N {
  int x;                    // OK
  static_assert(1 == 1);    // error: does not declare a name
}
```

— *end example*]

10.7.3 Import declaration [dcl.module.import]

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

1 A *module-import-declaration* shall appear only at global scope, and not in a *linkage-specification* or *proclaimed-ownership-declaration*. A *module-import-declaration* nominating a module M makes every citation and every exported declaration from the abstract semantics graph of M available, as a citation, to name lookup in the current translation unit, in the same namespaces and contexts as in M. A *citation* for a declaration attached to a module M is a pair of M and the corresponding declset from the abstract semantics graph of M. [*Note:* The declarations in the declsets and the entities denoted by the declsets 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`, or if there exists a module `M3` such that `M1` has an interface dependency on `M3` and `M3` has an interface dependency on `M2`. A module shall not have an interface dependency on itself. [*Example:*

```
// Interface unit of M1
export module M1;
import M2;

// 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*]

4 A translation unit has an interface dependency on a module `M` if it is a module implementation unit of `M`, or if it contains a *module-import-declaration* nominating `M`, or if it has an interface dependency on a module that has an interface dependency on `M`.

10.7.4 Module exportation [dcl.module.export]

1 An exported *module-import-declaration* nominating a module `M2` in the purview of a module interface unit of a module `M` makes all exported names of `M2` visible to any translation unit importing `M`, as if that translation unit also contains a *module-import-declaration* nominating `M2`. [*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*] In addition to its usual semantics, a *module-import-declaration* nominating a module M with a module interface unit containing one or more exported *module-import-declaration*s also behaves as if it nominates each module nominated by an exported *module-import-declaration* in M; this may in turn lead it to be considered to nominate yet additional modules.

10.7.5 Proclaimed ownership declaration [dcl.module.proclaim]

proclaimed-ownership-declaration:
    `extern module` *module-name* `:` *declaration*

1 A *proclaimed-ownership-declaration* shall only appear at namespace scope. It shall not appear directly or indirectly within an unnamed namespace. A *proclaimed-ownership-declaration* has the declarative effects of its *declaration*. The *declaration* shall declare at least one name, and the *decl-specifier-seq* (if any) of the *declaration* shall not contain `static`. The *declaration* shall not be a *namespace-definition*, an *export-declaration*, or a *proclaimed-ownership-declaration*. The *declaration* shall not be a defining declaration (6.1). A *proclaimed-ownership-declaration* nominating a module M shall not appear in the purview of M.

2 A *proclaimed-ownership-declaration* asserts that the entities introduced by the declaration are exported by the nominated module. [*Note:* A *proclaimed-ownership-declaration* may be used to break circular dependencies between two modules (in possibly too finely designed components.) [*Example:*

```
// TU 1
export module Ty;
extern module Sym: struct Symbol;
export struct Type {
  Symbol* decl;
  // ...
};

// TU 2
export module Sym;
extern module Ty: struct Type;
export struct Symbol {
  const char* name;
  const Type* type;
  // ...
```

```
    };
```

— end example] *— end note*]

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

10.7.6 Reachability [dcl.module.reach]

1 When declarations from the abstract semantics graph of a module M are made available to name lookup in another translation unit TU, it is necessary to determine the interpretation of the names they introduce and their semantic properties. Except as noted below, the *reachable semantic properties* of declset D (or of the entity, if any, denoted by that declset) of the abstract semantic graph of M from TU are

— if D contains at least one exported declaration, the semantic properties cumulatively obtained in the context of the exported declaration (10.7.2) members of D in the module interface unit of M. Furthermore, if D denotes an inline function, the property that the inline function has a definition (10.1.2) is a reachable semantic property, even if that definition is not exported. Otherwise,

— the semantic properties cumulatively obtained in the context of all declaration members of D in the module interface unit of M.

[*Note:* These reachable semantic properties include type completeness, type definitions, initializers, default arguments of functions or template declarations, attributes, visibility to normal lookup, entities that are direct targets of edges emanating from D in the abstract semantics graph of M, etc. Since default arguments are evaluated in the context of the call expression, reachable semantic properties of the corresponding parameter types apply in that context. [*Example:*

```
// TU 1
struct F { int f { 42 }; };
export module M;
export using T = F;
export struct A { int i; };
export int f(int, A = { 3 });

export struct B;        // exported as incomplete type
struct B {              // definition not exported
  operator int();
};
export int g(B = B{});
export int h(int = B{}); // #1

export struct S {
  static constexpr int v(int);
};

export S j();          // S attendant entity of j()
constexpr int S::v(int x) { return 2 * x; }


// TU 2
import M;
int main() {
  T t { };               // OK: reachable semantic properties of T include completeness.
  auto x = f(42);        // OK: default argument A{3} evaluated here.
```

```
    auto y = h();          // OK: completeness of B only checked at #1.
    auto z = g();          // error: parameter type incomplete here.
    constexpr auto a = decltype(j())::v(3); // OK: S::v defined
                           // in the abstract semantics graph of M (10.1.2)
  }
```

— end example] *— end note*]

2 Within a module interface unit, it is necessary to determine that the declarations being exported collectively present a coherent view of the semantic properties of the entities they reference. This determination is based on the semantic properties of attendant entities. [*Note:* The reachable semantics properties of an entity, the declarations of which are made available via a *module-import-declaration*, are determined by its owning module and are unaffected by the importing module. [*Example:*

```
// module interface of M1
export module M1;
export struct S { };

// module interface of M2
import M1;
export module M2;
export S f();          // #1
export S* g();         // #2

// elsewhere
import M2;
auto x = f();          // OK: completeness of S obtained at #1
auto y = *g();         // OK: completeness of S obtained at #2
```

— end example] *— end note*]

For each declaration D exported from the module interface unit of a module M, there is a set of zero or more *attendant entities* defined as follows:

- — If D is a type alias declaration, then the attendant entities of D are those determined by the aliased type at the point of the declaration D.
- — If D is a *using-declaration*, the set of attendant entities is the union of the sets of attendant entities of the declarations introduced by D at the point of the declaration.
- — If D is a template declaration, the set of attendant entities is the union of the set of attendant entities of the declaration being parameterized, the set of attendant entities determined by the default type template arguments (if any), and the set consisting of the entities (if any) designated by the default template template argument, the default non-type template arguments (if any).
- — if D has a type T, the set of attendant entities is the set of attendant entities determined by T at the point of declaration.
- — Otherwise, the set of attendant entities is empty.

The *set of attendant entities determined by* a type T is defined as follows (exactly one of these cases matches):

- — If T is a fundamental type, then the set of attendant entities is empty.
- — If T is a member of an unknown specialization, the set of attendant entities is the set of attendant entities determined by that unknown specialization.

— If T is a class type owned by M, the set of attendant entities includes T itself, the union of the sets of the attendant entities determined by its direct base classes owned by M, the sets of the attendant entities of its data members, static data member templates, member functions, member function templates, the function parameters of its constructors and constructor templates. Furthermore, if T is a class template specialization, the set of attendant entities also includes: the class template if it is owned by M, the union of the sets of attendant entities determined by the type template-arguments, the sets of the attendant entities of the templates used as template template-arguments, the sets of the attendant entities determined by the types of the non-type template-arguments.

— If T is an enumeration type owned by M, the set of attendant entities is the singleton $\{T\}$.

— If T is a reference to U, or a pointer to U, or an array of U, the set of attendant entities is the set of attendant entities determined by U.

— If T is a function type, the set of attendant entities is the union of the set of attendant entities determined by the function parameter types and the return type.

— if T is a pointer to data member of class X, the set of attendant entities is the union of the set of attendant entities of the member type and the set of attendant entities determined by X.

— If T is a pointer to member function type of a class X, the set of attendant entities is the union of the set of attendant entities determined by X and the set of attendant entities determined by the function type.

— Otherwise, the set of attendant entities is empty.

If a class template X is an attendant entity, then its reachable semantic properties include all the declarations of the primary class template, its partial specializations, and its explicit specializations in the containing module interface unit. If a complete class type X is an attendant entity, then its reachable semantic properties include the declarations of its nested types but not the definitions of the types denoted by those members unless those definitions are exported. Furthermore, if X is an attendant entity of an exported declaration D, then its reachable semantic properties are restricted to those defined by the exported declarations of X (if X is introduced by an exported declaration), or by the semantic properties of X available at the point of the declaration D. [*Note:* If X is a complete class type that is an attendant entity, its nested types (including nested enumerations and associated enumerators) and member class templates are not considered attendant entities unless they are determined attendant entities by one of the rules above. Attendant entities allow type checking of direct member selection of an object even if that object's type isn't exported. Declarations, such as *asm-declaration* or *alias-declaration* or *static_assert-declaration*, that do not declare entities do not contribute to the set of attendant entities. *— end note*] [*Example:*

```
export module M;
export struct Foo;            // Foo exported as incomplete type
struct Foo { };
export using ::Foo;           // OK: exports complete type Foo

struct C { };
struct S {
  struct B { };
  using C = ::C;
  int i : 8;
  double d { };
};

export S f();         // S attendant entity of f().

// translation unit 2
import M;
int main() {
  int x = sizeof(decltype(f())::B);    // error: incomplete B
  int y = sizeof(decltype(f())::C);    // error: incomplete C
  decltype(f()) s { };
  s.d = 3.14;                          // OK
  return &s.i != nullptr;              // error: cannot take address of bitfield
}
```

— *end example*]

3 If *X* is an attendant entity of two exported declarations designating two distinct entities, its reachable semantic properties shall be the same at the points where the declarations occur. [*Example:*

```
export module M;
struct S;
export S f();      // #1
struct S { };
export S g();      // error: class type S has different properties from #1
```

— *end example*]

12 Classes [class]

12.2 Class members [class.mem]

12.2.4 Bit-fields [class.bit]

Modify paragraph 12.2.4/1 as follows:

1 [...] The bit-field ~~attribute~~semantic property is not part of the type of the class member. [...]

16 Overloading [over]

16.5 Overloaded operators [over.oper]

16.5.8 User-defined literals [over.literal]

Modify paragraph 16.5.8/7 as follows:

7 [*Note:* Literal operators and literal operator templates are usually invoked implicitly through user-defined literals (5.13.8). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates and they follow the same overload resolution rules. Also, they can be declared `inline` or `constexpr`, they may have internal, module, or external linkage, they can be called explicitly, their addresses can be taken, etc. —*end note*]

17 Templates [temp]

Modify paragraph 17/2 as follows:

2 A *template-declaration* can appear only as a namespace scope or class scope declaration. Its *declaration* shall not be an *export-declaration* or a *proclaimed-ownership-declaration*. In a function template declaration, the last component of the *declarator-id* shall not be a *template-id*.

17.6 Name resolution [temp.res]

17.6.4 Dependent name resolution [temp.dep.res]

Add new example to paragraph 17.6.4/1:

[*Example:*

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

// Interface unit of M1
#include "X.h"        // global module
namespace Q {
   void g_impl(X, X);
}
export module M1;
export template<typename T>
void g(T t) {
   g_impl(t, Q::X{ });   // #1: ADL in definition context finds Q::g_impl
}

// Interface unit of M2
#include "X.h"
import M1;
export module M2;
void h(Q::X x) {
   g(x);                 // OK
}
```

— end example]

Add new paragraphs to 17.6.4:

2 [*Example:*

```
// Interface unit of Std
export module Std;
export template<typename Iter>
void indirect_swap(Iter lhs, Iter rhs)
{
   swap(*lhs, *rhs);     // swap can be found only via ADL
}

// Interface unit of M
import Std;
export module M;

struct S { /* ...*/ };
void swap(S&, S&);      // #1;

void f(S* p, S* q)
{
   indirect_swap(p, q);    // instantiation finds #1 via ADL
}
```

— *end example*]

3 [*Example:*

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

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

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

— *end example*]

4 [*Note:* [*Example:*

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

// Module interface unit of B
export 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;
export 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
                  // point of instantiation of h<int>: just before 'i()'
                  // point of instantiation of g<std::string, int>: same as h<int>'s
                  // point of instantiation of f<std::string>: same as g<std::string, int>'s
}
```

— *end example*]

This example is 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*]

5 [*Note:* [*Example:*

```
// Module interface unit of M1
#include <algorithm>

export module M1;
export template<typename T, typename U>
void f(T& t, U& u) {
  min(t, u);                          // #1
}

// Module interface unit of M2
#include <locale>
struct Aux : std::ctype_base {
  operator int() const;
};
void min(Aux&, Aux&);   // #2

export module M2;
import M1;
export template<typename T>
void g(T t) {
  Aux aux;
  f(aux, aux);
}

// Elsewhere, translation unit of global module
import M2;
void h() {
  g(0);
```

```
    }
```

In the body of the function `h`, the call to `g` triggers a request for (implicit) instantiation of `g<int>`. The point of instantiation of that specialization is right before the definition of `h`. That instantiation, in turn, requests the implicit instantiation of `f<Aux,Aux>`. The point of instantiation of that specialization immediately preceeds that of `g<int>`. In that context, the invocation of `min`: (a) selects `std::min`; and (b) invokes the implicit conversion. In particular, the declaration at #2 is not used because it is neither available in the context of definition, nor in the context of instantiation of `f<Aux,Aux>`. However, paragraph 17.6.4.2/1 of the C++ Standard formally renders the behavior of the program undefined because the better match wasn't considered. This is a case where it is unclear if that paragraph is too broad and needs further restrictions, or if there ought to be a mechanism to consider all such functions. *— end example*] *— 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.

17.6.4.2 Candidate functions [temp.dep.candidate]

Modify paragraph 17.6.4.2/1 as follows

1 . . .

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external or module linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.