Document Number: P1103R1
Date: 2018-10-08
Reply to: Richard Smith Google
richard@metafoo.co.uk

Merging Modules

Contents

Part I

Design and commentary

1 Background [bg]

1.1 Introduction [bg.intro]

At the Jacksonville 2018 committee meeting, P0947R0 ("Another Take On Modules", hereafter referred to as *Atom*) was presented. Two options were polled:

— merging the Atom proposal with the Modules TS, and

— progressing the Atom proposal as a separate TS.

Both options passed, but the first option had stronger support.

At Rapperswill 2018, we presented P1103R0, representing our effort in merging the two proposals and the remaining outstanding questions from the merge. Many open questions were answered, and we voted to adopt the merged proposal into the Modules TS, and scheduled a 2-day ad-hoc meeting to discuss the remaining open questions.

At the Bellevue 2018 ad-hoc Modules meeting, we answered all remaining known open design questions. This paper provides a description for the resulting design, as well as wording (based on the wording in P1103R0, which in turn is based on the wording of the Modules TS) to incorporate this design into the C++20 working draft.

1.2 Acknowledgements [bg.ack]

This paper is based on the work of a great many people. The author would like to thank them all, and specifically Gabriel Dos Reis, on whose Modules TS this paper is primarily based.

2 Summary of merged proposal [merged]

2.1 Basics [merged.basic]

1 A *module unit* is a translation unit that forms part of a module. Such a translation unit begins with[1] a preamble, comprising a module declaration and a sequence of imports:

```
export_opt  module foo;
import a;
export import b;
// ... more imports ...
```

Within a module unit, imports may not appear after the end of the preamble. The `export` keyword indicates that a module unit is a *module interface unit*, which defines the interface for the module. For a module `foo`, there must be exactly one translation unit whose preamble contains `export module foo;`. This is the *primary module interface unit* for `foo` (2.2). (All other module interface units are module interface partitions; see 2.2.)

2 Two related but distinct notions are key to understanding module semantics:

(2.1) — A declaration is *visible* in a context if it can be found by a suitable name lookup.

(2.2) — A declaration is *reachable* in a context if its semantic effects are available for use. (For example, a class type is complete in contexts where a definition of the class is reachable.)

A declaration is reachable wherever it is visible, but the converse is not true in general. Imports control which namespace-scope names are visible to name lookup, and which declarations are reachable semantically. The behavior of an entity is determined by the set of reachable declarations of that entity. For example, class members and enumeration members are visible to name lookup if there is a reachable definition of the class or enumeration.

3 A declaration can be exported by use of the `export` keyword in a module interface unit:

```
export int a;
export {
void f();
}
```

Exported declarations in a module interface unit are visible to name lookup in contexts that import that module interface unit. Non-exported declarations (excluding those with internal linkage) in a module unit are visible to name lookup in contexts within the same module that import the module unit. All declarations in transitively-imported module units are reachable, whether or not they are exported.

4 If a declaration within a namespace is exported, the enclosing namespace is also implicitly exported (but other declarations in the namespace are not implicitly exported). If a namespace is explicitly exported, all declarations within that namespace definition are exported.

2.2 Module partitions [merged.part]

1 A complete module can be defined in a single source file. However, the design, nature, and size of a module may warrant dividing both the implementation and the interface into multiple files. Module partitions provide facilities to support this.

1) A global module fragment (2.3.1) may optionally precede the preamble.

2 The module interface may be split across multiple files, if desired. Such files are called *module interface partition*s, and are introduced by a module declaration containing a colon:

```
export module foo:part;
```

3 Module interface partitions behave logically like distinct modules, except that they share ownership of contained entities with the module that they form part of. This allows an entity to be declared in one partition and defined in another, which may be necessary to resolve dependency cycles. It also permits code to be moved between partitions of a module with no impact on ABI.

4 The primary module interface unit for a module is required to transitively import and re-export all of the interface partitions of the module.

5 When the implementation of a module is split across multiple files, it may be desirable to share declarations between the implementation units without including them in the module interface unit, in order to avoid all consumers of the module having a physical dependency on the implementation details. (Specifically, if the implementation details change, the module interface and its dependencies should not need to be rebuilt.) This is made possible by *module implementation partition*s, which are module partitions that do not form part of the module interface:

```
module foo:part;
```

6 Module implementation partitions cannot contain exported declarations; instead, all declarations within them are visible to other translation units in the same module that import the partition. [*Note:* Exportation only affects which names and declarations are visible outside the module. — *end note*]

7 Module implementation partitions can be imported into the interface of a module, but cannot be exported.

8 Module interface partitions and module implementation partitions are collectively known as *module partition*s. Module partitions are an implementation detail of the module, and cannot be named outside the module. To emphasize this, an import declaration naming a module partition cannot be given a module name, only a partition name:

```
module foo;
import :part;           // imports foo:part
import bar:part;        // syntax error
import foo:part;        // syntax error
```

2.3 Support for non-modular code [merged.nonmodular]

1 This proposal provides several features to support interoperation between modular code and traditional non-modular code.

2.3.1 Global module fragment [merged.legacy.frag]

1 The merged proposal permits Modules TS-style global module fragments, with the `module;` introducer proposed in P0713R1 and approved by EWG:

```
module;
#include "some-header.h"
export module foo;
// ... use declarations and macros from some-header.h ...
```

2 Prior to preprocessing, only preprocessor directives can appear in the global module fragment, but those directives can include `#include` directives that expand to declarations, as usual. Declarations in the global module fragment are not owned by the module.

3 In order to avoid bloating the interface of a module with declarations included into its global module fragment, declarations in the global module fragment that are not (transitively) referenced by the module unit are

discarded. In particular, such declarations are not reachable from other translation units that import the module unit, and cannot be found by the second phase of two-phase name lookup for a template instantiation whose point of instantiation is outside the module unit. A declaration in a global module fragment is considered to be referenced if it is named within the module unit (after the preamble), or if it is mentioned by a referenced declaration.

4 [*Example:*

```
module;
#include "some-header.h" // defines classes X, Y, Z
export module foo;

export using X = ::X; // export name X; retain all declarations
                      // of X from "some-header.h"
export Y f(); // export name f; retain all declarations
              // of Y from "some-header.h"
// Z is not mentioned, so is discarded
```

— *end example*]

2.3.2 Legacy header units [merged.legacy.import]

1 The merged proposal also permits Atom-style legacy header units, which are introduced by a special `import` syntax that names a header file instead of a module:

```
export module foo;
import "some-header.h";
import <version>;
// ... use declarations and macros from some-header.h and <version> ...
```

2 The named header is processed as if it was a source file, the interface of the header is extracted and made available for import, and any macros defined by preprocessing the header are saved so that they can be made available to importers.

3 Declarations from code in a legacy module header are not owned by any module. In particular, the same entities can be redeclared by another legacy header unit or by non-modular code. Legacy module headers can be re-exported using the regular `export import` syntax:

```
export module foo;
export import "some-header.h";
```

However, when a legacy header unit is re-exported, macros are not exported. Only the legacy header import syntax can import macros.

2.3.3 Reachability of legacy declarations [merged.legacy.reachability]

1 A declaration in a global module fragment or legacy header unit is reachable if it is visible. It is unspecified whether such a declaration is also reachable in contexts where it is not visible but is transitively imported. (Ideally, such a declaration would not be considered reachable in such contexts. However, in practice, making transitively-imported declarations unreachable would impose a severe implementation cost for some implementations, so we leave the extent to which this rule is enforced up to the implementation.)

2.3.4 Module use from non-modular code [merged.nonmodular.use]

1 Modules and legacy header units can be imported into non-modular code. Such imports can appear anywhere, and are not restricted to a preamble. This permits “bottom-up” modularization, whereby a library switches to providing only a modular interface and defining its header interface in terms of the modular interface.

Non-modular code includes translation units other than module units, headers imported as legacy header units, and the global module fragment of a module unit.

[2] When a `#include` appears within non-modular code, if the named header file is known to correspond to a legacy header unit, the implementation treats the `#include` as an import of the corresponding legacy header unit. The mechanism for discovering this correspondence is left implementation-defined; there are multiple viable strategies here (such as explicitly building legacy header modules and providing them as input to downstream compilations, or introducing accompanying files describing the legacy header structure) and we wish to encourage exploration of this space. An implementation is also permitted to not provide any mapping mechanism, and process each legacy header unit independently.

3 Comparison to prior proposals [vs]

3.1 Changes since R0 [vs.r0]

1 This section lists changes to the design of the merged modules proposal since P1103R0.

3.1.1 Namespace export [vs.r0.namespace]

1 In P1103R0 and in the Modules TS, all namespaces (excluding anonymous namespaces and those nested within them) that are declared in a module interface unit have external linkage and are exported. Following strong EWG direction in Rapperswil, in this document such namespace names are only exported if they are either explicitly exported, or if any name within them is exported. [*Note:* The new approach permits implementation-detail namespace names to be hidden from the interface of a module despite being declared in a module interface unit. — *end note*] [*Example:*

```
export module M;

export namespace A {} // exported
namespace B {         // exported
  export int n;
}
namespace C {         // not exported in this proposal, exported in TS / P1103R0
  int n;
}
```

— *end example*]

3.1.2 Reachability in template instantiations [vs.r0.templates]

1 Following discussion and direction from Bellevue, we use a path of instantiation rule to guide visibility and reachability of declarations within a template instantiation, based on the relevant rule from the Atom proposal:

> Within a template instantiation, the *path of instantiation* is a sequence of locations within the program, starting from the ultimate point of instantiation, via each intervening template instantiation, terminating at the instantiation in question. Names are visible and semantic properties are available within template instantiations if they would be visible or available at any point along the path of instantiation, or (for points outside the current translation unit) would be visible or available at the end of the translation unit containing the relevant point of instantiation.
>
> — P0947R1, 7.1 Templates and two-phase name lookup

This rule permits a template to make use of all declarations that were visible or reachable at each point along its path of instantiation, even if those declarations are not visible or reachable in the template definition context nor the template instantiation context.

2 [*Example:*

```
export module A;
export template<typename T, typename U> void f(T t, U u) {
  t.f();
}

module;
struct S { void f(); };
```

```
export module B;
import A;
export template<typename U> void g(U u) { S s; f(s, u); }

export module C;
import B;
export template<typename U> void h(const U &u) { g(u); }

import C;
int main() { h(0); }
```

The definition of `struct S` and the declaration of its member `f` are not reachable from the point of instantiation of `f<S, int>`, nor from the template definition. But this code is valid under this proposal, because `S` is reachable from module `B`, which is on the path of instantiation. — *end example*]

3 As described above (2.3.3), implementations are permitted to treat additional declarations as reachable even if they would not be reachable on the path of instantiation, if they are transitively imported at the point of instantiation. [*Example:*

```
module M;
struct S;
import C;
// unspecified whether a definition of S is reachable
// here or in the instantiation of h<S>
void q(const S &s) { h(s); }
```

— *end example*]

4 The same rule applies to the set of names found by ADL: names visible along the path of instantiation are visible to ADL. Internal-linkage declarations within the global module are ignored. In addition, exported declarations in the owning module of each associated type are visible to ADL.

3.1.3 Finding the end of the preamble [vs.r0.preamble]

1 In R0 of this proposal, the preprocessor was burdened with finding the end of the preamble, and making macros from legacy header units visible at that point. That was problematic both for implementers (as it is a challenging rule to implement) and for users (as code would silently do something different from what was expected, and imports in a preamble would behave differently from imports in non-modular code). This proposal uses a simpler rule: imported macros become visible immediately after the import declaration.

2 In order to make it practical to determine the set of imports of a module unit without first compiling the imported legacy header units (at least for a module unit with no global module fragment), an additional restriction is imposed: the program is ill-formed if, after reaching the preamble, an imported macro is expanded and then another import declaration is encountered.

3.2 Changes to the Modules TS [vs.ts]

1 This section lists the ways in which valid code under the Modules TS would become invalid or change meaning in this merged proposal.

2 A `module;` introducer is required prior to a global module fragment, as described in P0713R1 and approved by Evolution.

3 When an entity is owned by a module and is never exported, but is referenced by an exported part of the module interface, the Modules TS would export the semantic properties associated with the entity at the point of the export. If multiple such exports give the entity different semantics, the program is ill-formed:

```
export module M;
```

```
struct S;
export S f();  // S incomplete here
struct S {};
export S g();  // S complete here, error
```

Under the Atom proposal, the semantics of such entities are instead determined their the properties at the end of the module interface unit.

In this merged proposal, the semantics of all entities owned by a module are determined by their properties at the end of the module interface unit (regardless of whether they are exported). [*Note:* The order in which declarations appear within a module interface has no bearing on which semantic properties are exported in this merged proposal. — *end note*]

The Modules TS "attendant entities" rule is removed, because there are no longer any cases where it could apply.

4 Entities declared within `extern "C"` and `extern "C++"` within a module are no longer owned by that module. It is unclear whether this is a change from the intent of the Modules TS.

5 Namespace names are exported less often in this proposal, as discussed above.

3.3 Changes relative to the Atom proposal [vs.atom]

1 This section lists the ways in which valid code under the Atom proposal would become invalid or change meaning in this merged proposal.

2 The merged proposal supports global module fragments, which interferes with the Atom proposal's goal of making the preamble easy to identify and process with non-compiler tools. However, the benefits of the Atom approach are still available to those who choose not to put code in the global module fragment.

3 The identifiers `export` and `module` are taken as keywords by the merged proposal, rather than making them context-sensitive as proposed by the Atom proposal. This follows EWG's direction on this question from discussion of P0924R0.

Part II

Wording for applying the merged modules proposal to the C++20 working draft

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 sources for module units on which the current translation unit has an interface dependency (9.11.3) are 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*]

5.4 Preprocessing tokens [lex.pptoken]

Modify bullet 3 of paragraph 5.4/3 as follows:

Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (5.8) is only formed within a `#include` directive (19.2) or when the previous preprocessing token was lexically identical to the *identifier* `import`.

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]

6.1 Declarations and definitions [basic.def]

Modify paragraph 6.1/1 as follows:

1 A declaration (Clause 9) 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. [...]

Add the following two bullets to paragraph 6.1/2:

2 Each entity declared by a *declaration* is also *defined* by that declaration unless:

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

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

Change paragraph 6.2/1 as follows:

1 ~~No translation unit shall contain more than one definition of any~~ A variable, function, class type, enumeration type, or template shall not be defined where a prior definition is reachable (9.11.6).

Modify opening of paragraph 6.2/6 as follows

6 There can be more than one definition of a class type (Clause 10), enumeration type (9.6), inline function with external or module linkage (9.1.6), inline variable with external or module linkage (9.1.6), class template (Clause 12), non-static function template (12.6.6), static data member of a class template (12.6.1.3), member function of a class template (12.6.1.1), or template specialization for which some template parameters are not specified (12.8, 12.6.5) in a program provided that ~~each definition appears in a different translation unit~~ no prior definition is reachable (9.11.6) at the point where a definition appears, and provided the definitions satisfy the following requirements. There shall not be more than one definition of an entity with external linkage that is attached to a named module; no diagnostic is required unless a prior definition is reachable at a point where a later definition appears. Given such an entity named `D` defined in more than one translation unit, then

6.3 Scope [basic.scope]

6.3.6 Namespace scope [basic.scope.namespace]

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* (9.7.3) 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 translation unit M is imported into a translation unit N, the potential scope of a name X declared with namespace scope in M is extended to include the portion of the corresponding namespace scope in N following the first *module-import-declaration* or *module-declaration* in N that directly or indirectly imports M if

— X does not have internal linkage, and

— X is declared after the *module-declaration* in M, and

— either M and N are part of the same module or X is exported.

[*Note:* A *module-import-declaration* imports both the named translation unit(s) and any modules named by exported *module-import-declaration*s within them, recursively (9.11.3). —*end note*]
[*Example:*

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

// Translation unit #2
export module N;
export import M;

// Translation unit #3
import N;
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-name*s (9.1.3), *namespace-name*s (9.7), and *class-name*s (10.1)) 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) of that name. [...] 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~~ the reachable (9.11.6) redeclarations of that declaration used further in the expression processing (Clause 7).

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* 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 (9.7.1), 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 (10.7.3) 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 (9.7.1.2).

— All names except those of (possibly overloaded) functions and function templates are ignored.

— Any function or function template in the interface of a named module `M` (9.11) that has the same innermost enclosing non-inline namespace as some entity attached to `M` in the set of associated entities is visible within its namespace to a lookup that does not occur within a module unit of `M`, even if it is not visible during an ordinary lookup.

— All declarations that are visible at any point in the instantiation context (9.11.5) of the lookup are visible even if they are not visible during an ordinary lookup, excluding internal linkage declarations attached to the global module.

6.5 Program and linkage [basic.link]

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

translation-unit:
preamble$_{opt}$ *declaration-seq*$_{opt}$

preamble:
global-module-fragment$_{opt}$ *module-declaration*
preamble$_{opt}$ *module-import-declaration*

The *preamble* includes the longest possible sequence of *module-import-declaration*s.

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 (9.11.1) 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 10), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (9.1.3); or
— a named enumeration (9.6), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (9.1.3); or
— a template~~.~~

has its linkage determined as follows:

— if the enclosing namespace has internal linkage, the name has internal linkage;
— otherwise, if the declaration of the name is attached to a named module and is not exported (9.11.2), the name has module linkage;
— otherwise, the name has external 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 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. If the declaration is attached to a named module, the program is ill-formed.

Modify paragraph 6.5/9 and add /10 as follows:

9 Two names that are the same (Clause 6) 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, 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-type-lists of the functions (9.2.3.5) are identical; and
— when both names denote function templates, the signatures (12.6.6.1) are the same.

If multiple declarations of the same name with external linkage would declare the same entity except that they are attached to different modules, the program is ill-formed; no diagnostic is required. [*Note: using-declaration*s, typedef declarations, and *alias-declaration*s do not declare entities, but merely introduce synonyms. Similarly, *using-directive*s do not declare entities. —*end note*]

10 If a declaration would redeclare a reachable declaration attached to a different module, the program is ill-formed. [*Example:*

```
// module interface of M
module;
int f();              // #1
int g();              // #2, attached to the global module
export module M;
export using ::f;     // OK: does not declare an entity
int g();              // error: matches #2, but attached to 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] As a consequence of these rules, all declarations of an entity are attached to the same module; the entity is said to be attached to that module.

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` attached to the global module.

9 Declarations [dcl.dcl]

Add new alternatives to *declaration* in paragraph 9/1 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

9.1 Specifiers [dcl.spec]

9.1.6 The `inline` specifier [dcl.inline]

Modify paragraph 9.1.6/6 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 an inline function or variable appears in a translation unit at a point where no inline declaration of it is visible ~~before its first declaration as inline~~, the program is ill-formed. If a function or variable with external or module linkage is declared inline in one translation unit, there ~~it~~ shall be a visible ~~declared~~ inline declaration in all translation units in which it ~~appears~~is declared; 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*]

Add a new paragraph 9.1.6/7 as follows:

7 An exported inline function or variable shall be defined in the translation unit containing its exported 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 (9.1.5) is implicitly inline. —*end note*]

9.7 Namespaces [basic.namespace]

Add a new paragraph after 9.7/1 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.

2 [*Note:* A namespace name with external linkage is exported if any of its *namespace-definition*s is exported, or if it contains any *export-declaration*s (9.11.2). A namespace is never attached to a module, and never has module linkage even if it is not exported. *—end note*] [*Example:*

```
export module M;
namespace N1 {}                     // N1 is not exported
export namespace N2 {}              // N2 is exported
namespace N3 { export int n; }      // N3 is exported
```

—end example]

Add a new subclause 9.11 titled "**Modules**" as follows:

9.11 Modules [dcl.module]

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

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

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

module-partition:
    `:` *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*. Two *module-name*s are the same if they are composed of the same dotted sequence of *identifier*s.

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 with no *module-partition*, known as the *primary module interface unit* of the module.

3 A *module partition* is a module unit whose *module-declaration* contains a *module-partition*. A named module shall not contain multiple module partitions with the same dotted sequence of *identifier*s in their *module-partition*. All module partitions of a module that are module interface units shall be directly or indirectly exported by the primary module interface unit (9.11.3). No diagnostic is required for a violation of these rules. [*Note:* Module partitions can only be imported by other module units in the same module. The division of a module into module units is not visible outside the module. *—end note*]

4 [*Example:*

```
// TU 1
export module A;
export import :Foo;
export int baz();
```

```
// TU 2
export module A:Foo;
import :Internals;
export int foo() { return 2 * (bar() + 1); }

// TU 3
module A:Internals;
int bar();

// TU 4
module A;
import :Internals;
int bar() { return baz() - 10; }
int baz() { return 30; }
```

Module `A` contains four translation units:

- a primary module interface unit,
- a module partition `A:Foo`, which is a module interface unit forming part of the interface of module `A`,
- a module partition `A:Internals`, which does not contribute to the external interface of module `A`, and
- an implementation module unit providing a definition of `bar` and `baz`, which cannot be imported because it does not have a partition name.

—end example]

5 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.

6 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*]

7 A *module* is either a named module or the global module. A declaration is *attached* to a module determined as follows:

- If the declaration is
  - a replaceable global allocation or deallocation function (21.6.2.1, 21.6.2.2), or
  - a *namespace-declaration* with external linkage, or
  - within a *linkage-specification*,

  it is attached to the global module.
- Otherwise, the declaration is attached to the module in whose purview it appears.

8 A *module-declaration* that contains neither `export` nor a *module-partition* implicitly imports the primary module interface unit of the module as if by a *module-import-declaration*. [*Example:*

```
// TU 1
export module B;
import :Y;                      // OK, does not create interface dependency cycle
int n = y();

// TU 2
module B:X;                     // does not implicitly import B
int &a = n;                     // error: n not visible here
import B;
int &b = n;                     // OK
```

```
// TU 3
module B:Y;                     // does not implicitly import B
int y();

// TU 4
module B;                       // implicitly imports B
int &c = n;                     // OK
```

—end example]

9.11.2 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 *export-declaration* has the declarative effects of its *declaration* or its *declaration-seq* (if any). An *export-declaration* does not establish a scope and its *declaration* or *declaration-seq* shall not contain an *export-declaration*.

2 A declaration is *exported* if it is

- — a namespace-scope declaration declared within an *export-declaration*, or
- — a *module-import-declaration* declared with the `export` keyword (9.11.3), or
- — a *namespace-definition* that contains an *export-declaration*, or
- — a declaration within a legacy header unit (9.11.3) that satisfies the rules for an exported declaration below.

The *interface* of a module `M` is the set of all exported declarations within its purview. [*Example:*

```
export module M;
namespace A {           // exported
  export int f();       // exported
  int g();              // not exported
}
```

The interface of `M` comprises `A` and `A::f`. *—end example*]

3 An exported declaration shall declare at least one name. If the declaration is not within a legacy header unit, it shall not declare a name with internal linkage.

4 [*Example:*

```
module;
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]

5 If the *declaration* is a *using-declaration* (9.8 [namespace.udecl]), 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*]

6 An exported declaration shall not redeclare a non-exported declaration. [*Example:*

```
export module M;
export struct S;
struct S { int n; };
export typedef S S;     // OK, not a redeclaration of struct S
export struct S;        // error: exported declaration follows non-exported definition
```

—end example]

7 A name is *exported* by a module if it is introduced or redeclared by an exported declaration in the purview of that module. [*Note:* Exported names have either external linkage or no linkage; see 6.5. Namespace-scope names exported by a module are visible to name lookup in any translation unit importing that module; see 6.3.6. Class and enumeration member names are visible to name lookup in any context in which a definition of the type is reachable. *—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]

8 [*Note:* Redeclaring a name in an *export-declaration* cannot change the linkage of the name (6.5). [*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*]

9 [*Note:* Declarations in an exported *namespace-definition* or in an exported *linkage-specification* (10.5) are exported and subject to the rules of exported declarations. *—end note*] [*Example:*

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

—end example]

9.11.3 Import declaration [dcl.module.import]

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

1 A *module-import-declaration* shall appear only at global scope, and not in a *linkage-specification*. In a module unit, a *module-import-declaration* shall appear only within the *preamble*.

2 A *module-import-declaration imports* a set of translation units determined as described below. [*Note:* Namespace-scope names exported by the imported translation units become visible in the importing translation unit (6.3.6) and declarations within the imported translation units become reachable in the importing translation unit (9.11.6) after the import declaration. —*end note*]

3 A *module-import-declaration* that specifies a *module-name* M imports all module interface units of M.

4 A *module-import-declaration* that specifies a *module-partition* shall only appear after the *module-declaration* in a module unit in some module M. Such a declaration imports the so-named *module-partition* of M.

5 A *module-import-declaration* that specifies a *header-name* H imports a synthesized *legacy header unit*, which is a translation unit formed by applying phases 1 to 7 of translation (5.2) to the source file nominated by H, which shall not contain a *module-declaration*. [*Note:* All declarations within a legacy header unit are implicitly exported, and are attached to the global module. —*end note*] Two *module-import-declaration*s import the same legacy header unit if and only if their *header-name*s identify the same header or source file (14.2). [*Note:* A *module-import-declaration* nominating a *header-name* is also recognized by the preprocessor, and results in macros defined at the end of phase 4 of translation of the legacy header unit being made visible as described in 14.4. —*end note*] A declaration of a name with internal linkage is permitted within a legacy header unit despite all declarations being implicitly exported. If such a name is odr-used by a translation unit outside the legacy header unit, or by an instantiation unit for a template instantiation whose point of instantiation is outside the legacy header unit, the program is ill-formed.

6 When a *module-import-declaration* imports a translation unit T, it also imports all translation units imported by exported *module-import-declaration*s in T; such translation units are said to be exported by T. When a *module-import-declaration* in a module unit imports another module unit of the same module, it also imports all translation units imported by all *module-import-declaration*s in that module unit. These rules may in turn lead to the importation of yet more translation units.

7 A module implementation unit shall not be exported. [*Example:*

```
// Translation unit #1
module M:Part;

// Translation unit #2
export module M;
export import :Part;     // error: exported partition :Part is an implementation unit
```

—*end example*]

8 A module implementation unit of a module `M` that is not a module partition shall not contain a *module-import-declaration* nominating `M`. [*Example:*

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

—end example]

9 A translation unit *has an interface dependency* on a module unit `U` if it contains a *module-declaration* or *module-import-declaration* that imports `U` or if it has an interface dependency on a module unit that has an interface dependency on `U`. A translation unit 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]

9.11.4 Global module fragment [dcl.module.global]

global-module-fragment:
`module` `;` *declaration-seq*

1 A *global-module-fragment* specifies the contents of the *global module fragment* for a module unit. The global module fragment can be used to provide declarations that are attached to the global module and usable within the module unit. [*Note:* Before preprocessing, only preprocessing directives can appear in the global module fragment (14.3). *—end note*]

2 Declarations in the global module fragment are *discarded* if they are not referenced by the module unit. A discarded declaration is only available to name lookup and reachable within the module unit and template instantiations whose points of instantiation (12.7.4.1) are within the module, even when the instantiation context (9.11.5) includes the module unit. [*Example:*

```
// header "foo.h"
namespace N {
  struct X {};
  int f(X);
  int g(X);
  int h(X);
}

// module M interface
module;
#include "foo.h"
export module M;
// N::f is reachable via argument-dependent name lookup result
// in context of template definition
template<typename T> int use_f() { N::X x; f(x); }
// N::g is not reachable because g is a dependent name
// in context of template definition
template<typename T> int use_g() { N::X x; g((T(), x)); }
// N::h is reachable because use_h<int> has a point of
// instantiation in the module unit M
template<typename T> int use_h() { N::X x; h((T(), x)); }
int k = use_h<int>();
```

```
// module M implementation
module M;
int a = use_f<int>(); // ok
int b = use_g<int>(); // error: no viable function for call to g
int c = use_h<int>(); // ok
```

—*end example*]

3 The *basis* of a declaration D is a set of entites determined as follows:

- If D declares a *typedef-name*, the basis is the type-basis of the aliased type.
- If D declares a variable or function, the basis is the type-basis of the type of that variable or function and the innermost enclosing namespace, class, or 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, and the innermost enclosing namespace, class, or function.
- 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 and the default non-type template arguments, the type-bases of the default type template arguments, and the innermost enclosing namespace, class, or function. 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-definition*, the basis comprises the innermost enclosing namespace, if any.
- 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 A declaration from a global module fragment is *referenced* by the enclosing module unit M if it is

— the unique result of name lookup for an *unqualified-id* or *identifier* in M or in a template instantiation whose point of instantiation is in M, or
— a function named by an expression (6.2) in M or in a template instantiation whose point of instantiation is in M, or
— a lookup result for a dependent name in M (12.7.4) that is visible at the point of definition of the enclosing template, or
— a declaration of an entity that is in the basis of a declaration referenced by M, recursively.

9.11.5 Instantiation context [dcl.module.context]

1 The *instantiation context* is a set of locations within the program that determines which names are visible to argument-dependent name lookup (6.4.2) and which declarations are reachable (9.11.6) in the context of a particular declaration or template instantiation. The instantiation context depends on how the declaration was formed, or where the template instantiation was referenced.

2 During the implicit definition of a defaulted special member function (10.2.3), the instantiation context is the union of the instantiation context of the definition of the class and the instantiation context of the program construct that resulted in the implicit definition of the special member function.

3 During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (12.7.4.1), the instantiation context is the union of the instantiation context of the enclosing specialization and, if the template is defined in a module interface unit of a module M and the point of instantiation is not in a module interface unit of M, the point at the end of the primary module interface unit of M.

4 During the implicit instantiation of a template that is implicitly instantiated because it is referenced from within the implicit definition of a defaulted special member function, the instantiation context is the instantiation context of the defaulted special member function.

5 During the instantiation of any other template specialization, the instantiation context comprises the point of instantiation of the template.

6 In any other case, the instantiation context at a program point comprises that program point.

7 [*Example:*

```
// translation unit #1
export module stuff;
export template<typename T, typename U> void foo(T, U u) { auto v = u; }
export template<typename T, typename U> void bar(T, U u) { auto v = *u; }

// translation unit #2
export module M1;
import "defn.h";         // provides struct X {};
import stuff;
export template<typename T> void f(T t) {
 X x;
 foo(t, x);
}

// translation unit #3
export module M2;
import "decl.h";         // provides struct X; (not a definition)
import stuff;
export template<typename T> void g(T t) {
 X *x;
 bar(t, x);
}

// translation unit #4
import M1;
import M2;
void test() {
  // OK: the instantiation context of foo<int, X> comprises
  // the point at the end of translation unit #1,
  // the point at the end of translation unit #2, and
  // the point of the call to f(0) below, so
  // the definition of X is reachable (9.11.6)
  f(0);

  // the instantiation context of foo<int, X> comprises
  // the point at the end of translation unit #1,
  // the point at the end of translation unit #3, and
  // the point of the call to g(0) below, so
  // the definition of X is not necessarily reachable
  g(0);
}
```

— end example]

9.11.6 Reachability [dcl.module.reach]

1 A translation unit is *reachable* from a program point if it is a module interface unit on which the translation unit containing the program point has an interface dependency, or it is a translation unit that the translation unit containing the program point directly or indirectly imports, prior to that program point (9.11.3). [*Note:* While module interface units are reachable even when they are only transitively imported via a non-exported import declaration, namespace-scope names from such module interface units are not visible to name lookup (6.3.6). *— end note*]

2 It is unspecified whether additional translation units on which the program point has an inter-

face dependency are considered reachable, and under what circumstances.[3] Programs intended to be portable should avoid depending on the reachability of any additional translation units.

3 A declaration is *reachable* if, for any program point in the instantiation context (9.11.5),

— it appears prior to that program point in the same translation unit, or

— it is not discarded (9.11.4) and appears in a translation unit that is reachable from that program point.

[*Note:* Whether a declaration is exported has no bearing on whether it is reachable. —*end note*]

4 The *reachable semantic properties* of an entity within a context are the accumulated properties of all reachable declarations of that entity, and determine the behavior of the entity within that context. [*Note:* These reachable semantic properties include type completeness, type definitions, initializers, default arguments of functions or template declarations, attributes, visibility of class or enumeration member names to ordinary lookup, etc. Since default arguments are evaluated in the context of the call expression, the reachable semantic properties of the corresponding parameter types apply in that context. [*Example:*

```
// translation unit #1
export module M:A;
export struct B;

// translation unit #2
module M:B;
struct B {
  operator int();
};

// translation unit #3
module M:C;
import :A;
B b1;                       // error: no reachable definition of struct B

// translation unit #4
export module M;
export import :A;
import :B;
B b2;

// translation unit #5
module X;
import M;
B b3;                       // error: no reachable definition of struct B
```

—*end example*] —*end note*]

5 [*Note:* The reachable semantic properties for an entity attached to a module M are the same for all contexts outside that module in which the entity can be referenced, irrespective of whether M is directly or indirectly imported. —*end note*]

6 [*Note:* An entity can have reachable declarations and therefore reachable semantic properties even if it is not visible to name lookup. —*end note*] [*Example:*

```
export module A;
struct X {};
export using Y = X;
```

3) Implementations are not required to prevent the semantic effects of additional translation units involved in the compilation from being observed.

```
module B;
import A;
Y y;              // OK, definition of X is reachable
X x;              // ill-formed: X not visible to unqualified lookup
```

—end example]

10 Classes [class]

10.2 Class members [class.mem]

10.2.10 Bit-fields [class.bit]

Modify paragraph 10.2.10/1 as follows:

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

11 Overloading [over]

11.5 Overloaded operators [over.oper]

11.5.8 User-defined literals [over.literal]

Modify paragraph 11.5.8/7 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 can have internal, module, or external linkage, they can be called explicitly, their addresses can be taken, etc. —*end note*]

12 Templates [temp]

Modify paragraph 12/4 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*. In a function template declaration, the last component of the *declarator-id* shall not be a *template-id*. [...]

12.7 Name resolution [temp.res]

12.7.4 Dependent name resolution [temp.dep.res]

Change in 12.7.4/1:

1 In resolving dependent names, names from the following sources are considered:

— Declarations that are visible at the point of definition of the template.

— Declarations from namespaces associated with the types of the function arguments both from the instantiation context ~~(12.7.4.1)~~ (9.11.5) and from the definition context.

[*Example:*

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

// header file "G.h"
namespace Q {
  void g_impl(X, X);
}

// interface unit of M1
module;
#include "X.h"
#include "G.h"
export module M1;
export template<typename T>
void g(T t) {
  g_impl(t, Q::X{ });    // ADL in definition context finds Q::g_impl, g_impl not discarded
}

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

— end example]

Add new paragraphs to 12.7.4:

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
module;
import Std;
export module M;

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

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

—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
module;
#include "X.h"
import F;
export module M;
void g(X x) {
  f(x);              // OK: instantiates f from F,
                     // operator+ is visible in instantiation context
}
```

—end example]

4 [*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 C1
module;
#include <string>    // operator+ not referenced, discarded
export module C1;
import B;
export template<typename T>
void h(T t) {
  g(std::string{ }, t);
}

// translation unit
import C1;
void i() {
   h(0);         // ill-formed: '+' not found at #1
}

// module interface unit of C2
export module C1;
import B;
import <string>;
export template<typename T>
void h(T t) {
  g(std::string{ }, t);
}

// translation unit
import C1;
void i() {
   h(0);         // OK, '+' found in instantiation context:
                 // visible at end of module interface unit of C2
}
```

—end example]

12.7.4.1 Point of instantiation [temp.point]

Delete paragraph 12.7.4.1/7:

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.~~

12.7.4.2 Candidate functions [temp.dep.candidate]

Modify paragraph 12.7.4.2/1 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.

14 Preprocessing directives [cpp]

Modify paragraph 14/5 as follows:

5 The implementation can process and skip sections of source files conditionally, include other source files, import macros from legacy header units, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

14.2 Source file inclusion [cpp.include]

Add a new paragraph after 14.2/6 as follows:

7 If the header identified by the *header-name* denotes a legacy header unit, the preprocessing directive is instead replaced by the *preprocessing-token*s

`import` *header-name* `;`

How the set of headers denoting legacy header units is specified is implementation-defined.

Add a new subclause 14.3 titled "**Global module fragment**" as follows:

14.3 Global module fragment [cpp.glob.frag]

pp-global-module-fragment:
`module ;` *pp-bracketed-tokens* `module`

1 If the first two preprocessing tokens at the start of phase 4 of translation are `module ;`, the result of preprocessing shall begin with a *pp-global-module-fragment* for which all *preprocessing-token*s in the *pp-bracketed-tokens* were produced directly or indirectly by source file inclusion (14.2), and for which the second `module` *preprocessing-token* was not produced by source file inclusion or macro replacement (14.3). Otherwise, the first two preprocessing tokens at the end of phase 4 of translation shall not be `module ;`.

Add a new subclause 14.4 titled "**Legacy header units**" as follows:

14.4 Legacy header units [cpp.module]

pp-import:
`import`$_{opt}$ *header-name* *pp-decl-suffix*$_{opt}$ `;`

pp-decl-suffix:
pp-decl-suffix$_{opt}$ *pp-decl-suffix-token*
pp-decl-suffix$_{opt}$ `[` *pp-bracketed-tokens* `]`

pp-decl-suffix-token:
any *preprocessing-token* other than `[`, `]`, or `;`

pp-bracketed-tokens:
pp-bracketed-tokens$_{opt}$ *pp-bracketed-token*
pp-bracketed-tokens$_{opt}$ `[` *pp-bracketed-tokens* `]`

pp-bracketed-token:
any *preprocessing-token* other than `[` or `]`

1 A sequence of *preprocessing-token*s matching the form of a *pp-import* instructs the preprocessor to import macros from the legacy header unit (9.11.3) denoted by the *header-name*. The `;` *preprocessing-token* shall not be produced by macro replacement (14.3). The *point of macro import* for a *pp-import* is immediately after the `;` terminating the *pp-import*.

2 A *macro directive* for a macro name is a `#define` or `#undef` directive naming that macro name. An *exported macro directive* is a macro directive occuring in a legacy header unit whose macro name is not lexically identical to a keyword. A macro directive is *visible* at a source location if it precedes that source location in the same translation unit, or if it is an exported macro directive whose legacy header unit, or a legacy header unit that transitively imports it, is imported into the current translation unit by a *pp-import* whose point of macro import precedes that source location.

3 Multiple macro directives for a macro name may be visible at the same source location. The interpretation of a macro name is determined as follows:

- A macro directive *overrides* all macro directives for the same name that are visible at the point of the directive.
- A macro directive is *active* if it is visible and no visible macro directive overrides it.
- A set of macro directives is *consistent* if it consists of only `#undef` directives or if all `#define` directives in the set are valid as redefinitions of the same macro.

When a *preprocessing-token* matching the macro name of a visible macro directive is encountered, the set of active macro directives for that macro name shall be consistent, and semantics of the active macro directives determine whether the macro name is defined and the behavior of macro replacement. [*Note:* The relative order of *pp-import*s has no bearing on whether a particular macro definition is active. —*end note*]

4 If an imported macro is replaced (14.3) or is named by a *defined-macro-expression* outside of a *pp-global-module-fragment*, there shall be no *pp-import*s with a later point of macro import.