Document Number: P2640R0
Date: 2022-09-06
To: SC22/WG21 EWG
Reply to: Nathan Sidwell nathan@acm.org

Modules: Inner-scope Namespace Entities: Exported or Not?

Nathan Sidwell

Certain namespace-scope entities can be introduced without a declaration appearing at namespace scope. In module-interface purview, such entities could have external or module linkage. That linkage must be known immediately, as the entities may be used in code-generation, where the linkage could be significant for symbol generation.

1 Background

Core DR2588[1] raised a design question about the linkage friend declarations that introduce a new entity. The question was forwarded to EWG, and a sequence of events led EWG to answering a modified question, leaving the original DR unanswered. Since then it has been realized that there are other declarations with a similar behaviour to the friend case, and they too need a specific linkage.

The DR noted that the WP is contradictory:

> A function first declared in a friend declaration has the linkage of the namespace of which it is a member (6.6 [basic.link]). [11.8.4]class.friend/4

(There is no similar provision for friend classes first declared in a class.)

> ... otherwise, if the declaration of the name is attached to a named module (10.1 [module.unit]) and is not exported (10.2 [module.interface]), the name has module linkage; [6.6]basic.link/4.8

> A declaration is *exported* if it is declared within an *export-declaration* and inhabits a namespace scope or it is … [10.2]module.interface/2

Depending on the precise example, the above rules lead to conflicting results.

1 https://cplusplus.github.io/CWG/issues/2588.html

1.1 Linkage

An entity's linkage must be known whenever it is involved in code generation. A typical case will involve mangling type information into an object-level symbol, and that mangling may well differ upon external vs module linkage. For instance in a weak-ownership model, the symbol-name of external-linkage entities is blind to module attachment. A strong ownership model may indicate whether symbols are internal to a module or not. (Object formats using mechanism other than mangling to convey such linkage or module ownership may well have a similar requirement.)

1.2 Forward Declarations, Redeclarations and Definitions

Many entities may be redeclared, with subsequent declarations introducing more information (such as default arguments). Also, entities may be forward-declared and then defined at a later point. In all cases the later declaration need not repeat the 'export' keyword, and will infer its presence or absence from the first declaration.

This leads to the question of whether, when the later declaration or definition introduces a new namespace-scope entity, should that entity be affected by the presence or absence of 'export' on the declaration in which it is present, or by the external or module linkage of the entity in which it is being introduced?

1.3 Cases

The known cases are as follows:

1.3.1 Friends

The motivating question was:

```
// TU Friend
export module Friend;
export class X {
  friend int Frob (X *); // #1 linkage?
};

// TU User1
import Friend;
int V = Frob ((X *)nullptr); // #2
```

What is the linkage of #1? It must be known at #2, as that generates code to call Frob – found via ADL of its arguments. Class X could have an exporting forward-declaration and a non-explicitly exported definition. That would make no difference to class X itself, is it significant to such friends?

```
/ TU Friend
export module Friend;
```

```
export class X;
class X { // no export on this redeclaration
  friend int Frob (X *); // linkage?
};
```

1.3.2 Class Member

A class definition can introduce other types into the enclosing scope:

```
export module Struct;
export struct Y {
  struct Z1 *p; // #3 linkage?
};
void Toto (Z1 *) {};  // #4
```

At #4 the linkage of Z1 must be know, because it involved in the symbol name of Toto.

I believe this comes from compatibility with C.

This case is also affected by the possibility of an exported forward declaration and non-explicitly exported class definition.

1.3.3 Function Parameter

Similarly a function declaration can introduce types into the enclosing scope:

```
export module FnParm;
export void FnParm (struct Z2 *); // linkage of Z2?
void Toto (Z2 *) {};  // #5
```

Again, at #5 the linkage of Z2 must be known.

This is not compatible with C, where Z2 would have function-parameter scope, and thus #5 (with the addition of C-required ‘struct’) would declare a function of a different type.

1.3.4 Default Function Parameter

A default function argument value can introduce a new type into the containing scope:

```
export module DfltFnArg;
export void Fn (void * = (struct Z3 *)0);
void Corge (Z3 *) {}
```

As with the class cases, it could be an exportless redeclaration of the function that adds the default argument.

The function itself need not be a member of the namespace:

```
export module DfltMemArg;
export struct S2 {
  void Fn (void *);
};
void S::Fn (void * = (struct Z4 *)0) {}
void Beans (Z4 *) {}

namespace B {
  export void Fn (void *);
}
void B::Fn (void * = (struct Z5)0);
void Beans2 (Z5 *) {};
```

The type introduced by the default argument is injected into the lexical scope containing the function redeclaration.

1.3.5 Non-Type Template Parameter

A non-type template parameter can declare a new type:

```
export module NTTP;
export template<struct NTTP *> class T1;
void TUse1 (NTTP *) {}
```

1.3.6 Default Non-Type Template Parameter

As with default function arguments, default non-type template parameters can introduce a new type:

```
export module DNTTP;
export template <void * = (struct Z6 *)0> class T2;
void TUse2 (Z6 *) {}
```

As with default function arguments, the default NTTP can be introduced in a redeclaration, which can reside in a different lexical scope to the template itself.

1.3.7 Default Type Template Parameter

A default template type parameter can indroduce a new type:

```
export module DTTP;
export template <typename T = struct Z7> class T3;
void Tuse3 (Z7 *) {}
```

And as with other default parameters, this can be added in a redeclaration.

1.3.8 Unscoped Enumerations

Unscoped enumerations are an example that was not noticed until the above examples came to light. Usually unscoped enumerations are not forward declared, and their enumerators have the same linkage as the enumeration itself (there being no other way to export the enumerators):

```
export module E1;
export enum E1 { A };
```

However, they can be forward-declared by specifying an underlying type:

```
export module E2;
export enum E2 : int;
enum E2 : int { B };
```

This is another case of an implicitly exported definition, and how that affects the linkage of any namespace-scope entities it introduces.

1.3.9 Variable Initializers

Variable initializers can introduce new types:

```
export module V1;
export auto v = (struct V *)0;
V *vp;
```

This is the explicitly-named version of:

```
export module V2;
export inline auto frobber = [] (int i) { return i + 1;};
```

which I have encountered in the ranges library.[2]

Type Using Declaration

A syntactically similar case is for a type using declaration:

```
export module V3;
export using T1 = struct T2 {};
T2 t2;
```

This is semantically identical to an unsurprising typedef declaration:

```
export module V3;
export struct T2 {} typedef T1;
T2 t2;
```

2 Modules introduced a new ODR case to handle this use of lambdas – the lambda is keyed to the variable it initializes and follows that variable's ODR behaviour.

and it is expected that T2 is named in the current scope.

1.3.10 Decltype

One can introduce types inside a decltype:

```
export module DT;
export decltype ((struct DT *)0) dt;
DT *dtp;
```

1.4 Name Lookup

The above examples can be reframed in terms of namespace-scope name lookup. Consider:

```
// TU NS
export module NS;
export void Fn (struct A *);
export struct Y;
struct Y {
 struct B *p;
 friend struct C;
 friend int D ();
};
class C; // Make C visible to name lookup
void D (); // Likewise for D
export template<struct E, typename, void *> class Z;
template<struct E, typename = struct F, void * = (struct G)0> class Z;
export enum X : int;
enum X : int { H };
export auto v = (struct I *)0;
export decltype ((struct J *)0) dt;
```

An implementation unit of NS can name entities A to J, as both module-linkage or external-linkage entities are visible to it:

```
// TU NS-impl
module NS;
A *ai; // Well-formed
B *bi; // Well-formed
C *ci; // Well-formed
int di = D (); // Well-formed
E *ei; // Well-formed
F *fi; // Well-formed
G *gi; // Well-formed
decltype (H) *hi; // Well-formed
I *ii; // Well-formed
J *ji; // Well-formed
```

Is the same true for an importer of NS?

```
// TU User2
import NS;
A *a; // #6
B *b; // #7
C *c; // #8
int d = D (); // #9
E *ei; // #10
F *fi; // #11
G *gi; // #12
decltype (H) *hi; // #13
I *ii; // 14
J *ji; // 15
```

Which, if any, of #6..#15 are well formed? Where applicable, does explicitly exporting the redeclaration make a difference?

1.5 Language-linkage

Language-linkage declarations allow attachment to the global module from within named-module purview. Does this attachment apply to the above cases?

```
export module ATT;
extern “C++” {
  export void Fn (struct A);
  export class Y {
    class B;
    friend class C;
  };
}
```

What is the attachment of A, B & C? (And similarly for the other cases not enumerated here.)

Note that, with language-linkage declarations, entities in module purview with external-linkage need not be findable by name-lookup from importers:

```
// TU GMF
export module GMF;
extern “C++” {
  export int Importable (); // #10
  int Internal (); // #11
}

// TU GMF-IMPL
module GMF;
```

```
int a = Importable (); // ok
int b = Internal (); // ok

// TU GMF-USER
import GMF;
int c = Importable (); // ok
int d = Internal (); // ERROR
```

Both Importable and Internal have external linkage, but only the former is nameable by imports (they may explicitly declare Internal and then name it).

Existing behaviour of C language-linkage is to give introduced friends C linkage too:

```
extern "C" struct X {
 friend void F ();
};
void F () {} // Has “C” linkage.
```

Functions, variables and function types have language linkage:

> All functions and variables whose names have external linkage and all function types have a language linkage. [9.11]dcl.link/1

By omission, other entities presumably do not.

Language linkage explicitly applies to all applicable entities within the declaration:

> In a linkage-specification, the specified language linkage applies to the function types of all function declarators and to all functions and variables. ... [9.11]dcl.link/5

with some exceptions:

> A C language linkage is ignored in determining the language linkage of class members, friend functions with a trailing requires-clause, and the function type of class member functions. [9.11]dcl.link/5

2 Discussion

DR 2588 suggested several possible solutions:

- Should the friend's linkage be affected by the linkage of the befriending class?
- Or should the friend's linkage be affected by the presence or absence of `export` on the class definition itself?
- Or should the friend's linkage be determined ignoring any enclosing `export` and ignoring whether the enclosing class is exported, per 11.8.4 [class.friend] paragraph 4 (alone)?

- Or should the friend's linkage be as-if the declaration inhabited its nearest enclosing namespace scope, without the `friend`?

The EWG meeting of 2022-06-09[3] failed to reach consensus on any of these options. Consensus was reached on a 5th alternative:

- If the introducing-friend declaration is a definition, it has the linkage of the befriending class.

This leaves unanswered the linkage of an introducing-friend declaration that is not a definition.

2.1 Make them Ill-formed?

Is it harmful to make any of the problematic cases ill-formed?

The non-friend structure-member case arises out of C compatibility. Such code cannot have module-purview, thus this would seem an opportunity to clean up the language.

The origin of the function-parameter case is unknown.[4]

The non-defining friend case does have use in existing C++ code. The declaration must match some entity defined somewhere in the program. Is there difficulty with requiring this friend declaration to not be an introducing declaration?

The default function argument case is surprising, and I suspect extremely rare.

The template parameter, variable initialization (except for initializing with an unnameable lambda) and decltype instances are all variations on the same theme. The declaration either does not create a new scope, or does not create it until a later lexical position. Thus the newly created type is placed in the same scope. I suspect they are all rare.

2.2 Lexical or Semantic Exportedness?

For cases that do not become forbidden, should the linkage be determined by the lexical presence of ‘export’ on the containing declaration, or by the semantic exportedness of that declaration (where they can differ)?

Lexical behaviour removes an action-at-a-distance interaction with a previous declaration. It may also be less confusing for the default argument cases where the redeclaration is in a different lexical scope to the scope containing the declaration itself.

2.2.1 Parsing

The cases that add a default function or template parameter value to an already-declared entity will be problematic, if the semantic exportedness of the entity to which they attach is significant.

3 https://wiki.edg.com/bin/view/Wg21telecons2022/EWG-2022-06-09

4 One can speculate it is because the alternative, of giving the type parameter-scope, results in unusable functions.

In these cases, the default expression (or type) is parsed before the entity to which they apply is determined:[5]

```
export void Foo (void *);
export template <typename, void *> class TPL;

void Foo (void * = (struct X *)0);
template <typename = struct Y, void * = (struct Z *)0> class TPL;
```

This suggests that these cases should either:

- become ill-formed, or
- be module-linkage, or
- be affected by the presence or absence of a syntactic ‘export’ keyword on the redeclaration.

2.3 Ship Vehicle

A DR against the current C++ std.

3 Revision History

R0 First version

5 The redeclarations are never deferred-parsing regions, as they are lexically at namespace-scope.