Modules: Contexts of Template Instantiations and Name Lookup

Gabriel Dos Reis
Microsoft

Background

One of the design principles behind "modules" is to ensure that the working programmer does not need to learn new name lookup rules – we have far too many of those. That principle has driven the design and the specification: we defer to "existing name lookup rules" where no new rule is needed. For the most part, that has been well understood in specification, implementation, and usage. This note is to clarify why that principle also holds in the case of template instantiations involving entities from several modules. In general, we don't add new normative paragraphs to state that no new rule is needed.

The Issue

At the Fall 2015 meeting, during a preliminary review by CWG, John Spicer asked whether something needs to be said about name lookup now that we have module boundaries and template instantiations could involve entities from several modules. I replied that the existing rules cover the case, and that no new rule is needed. So, what are the "existing rules"? They are the rules from section 14.6.4 titled "Dependent name resolution", specifically:

> 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 (14.6.4.1) and from the definition context.

Going into the Fall 2016 Issaquah meeting, the term "namespace partition" was used to make precise which contexts dependent names would be looked from as far as exported declarations were concerned. At that same meeting, the resolution of Module Issue #4 removed the term "namespace partition" so the added wording to make precise which contexts are considered was gone. In parallel, Richard Smith, Daveed Vandevoorde, and John Spicer came up with the following example (that we all expect to work) and asked if there is anything that needed to be added to ensure that everything works as expected and that we don't end up in a situation where we are asking to prove a negative.

```
module A;
export template<typename X, typename Y, typename Z>
void f(X x, Y y, Z z) {
      g(x);
      g(y);
      g(z);
}

module B;
import A;
```

```
export struct X {};
export void g(X);
export template<typename Y, typename Z>
void f(Y y, Z z) {
      f(X(), y, z);
}

module C;
import B;
export struct Y {};
export void g(Y);
export template<typename Z>
void f(Z z) {
      f(Y(), z);
}

// Client code:
import C;
struct Z {};
void g(Z);
int main() {
      f(Z());
}
```

Analysis

First, let's walk through the example provided with the "existing name lookup rules".

- In the context of definitions:
    - in module A: there is no declaration for `g()` for the dependent use of `g()`.
    - in module B: we have both `f(#1, #2, #3)`@A and `f(#1, #2)`@B for the dependent use of `f()`.
    - In module C: we have `f(#1, #2)`@B and `f(#1)`@C for the dependent use of `f()`.
- In main():
    - `f(#1)`@C is instantiated with [#1 = Z]
    - The instantiation of `f(#1)`@C triggers the instantiation of `f(#1,#2)`@B with [#1=Y@C, #2=Z]
    - The instantiation of `f(#1,#2)`@B triggers the instantiation of `f(#1,#2,#3)`@A with [#1=X@B, #2=Y@B, #3=Z]
- In the context of instantiation of `f(#1, #2, #3)`@A:
    - ADL finds:
        - g(X@B)@B
        - g(Y@C)@C
        - `g(Z)`

The instantiation of `f()` as called from the `main()` will succeeds by the collective success of the template instantiations requested along the way. This despite the lack of "transitive import". The key here is that ADL acts as the link necessary to stich the appropriate contexts together. In the end, the example works just as expected.

Global Module

At the Spring 2017 meeting in Kona, further analysis revealed that:

- for named modules, the current specification works as expected thanks to ownership
- for global modules, because entities can be defined multiple times (as long as they obey the ODR), there is no unique or canonical place to use to perform ADL. The next examples are meant to help clarify the rules in the proposed resolution. The notion of "namespace partition" is revived.

First, terminology. A namespace partition of a namespace N in a module unit U is the collection of namespace definitions of N appearing in the module unit U. The notion also holds for the global module. The namespace partition of the global module in any translation unit is the declarative region of that translation unit that is not in the purview of any module. So, for a translation unit with no module declaration, it is that entire translation unit. For a module unit, the namespace partition of the global module is the portion of that translation unit preceding the module declaration.

Example 1

Consider a source file "X.h" with the following declarations

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

Next, consider a module F defined as follows

```
module F;
export template<typename T>
void f(T t) { t + t; }
```

Finally, consider this (client) translation unit

```
#include "X.h"
import F;
module M;
void g(X x) {
      f(x);       // instantiates f<X>@F
}
```

This example works as well as John Spicer's original example. The reason is that the translation unit being process contains the declarations of X and operator+ in (thanks to the copy-and-paste semantics of #include), and following similar steps as before.

Example 2

Consider the following three translation units.

Header file X.h

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

Module M1:

```
#include “X.h”    // global module
namespace Q {
      X operator+(X,X);       // private implementation details
}
module M1;
export template<typename T>
void g(T t) { t + Q::X{ }; }
void j(Q::X x) {
      g(x);                   // #1
}
```

Module M2:

```
#include “X.h”
module M2;
import M1;
void h(Q::X x) {
      g(x);                   // #2
}
```

In this example, line marked #1 is OK because the instantiation of g<X> in that translation unit performs ADL on ‘operator+’ and finds the Q::operator+, which was meant to be an implementation detail.

However, line #2 results in an error (as intended by the design and specification) because in the context of instantiation of g<X> there is no declaration of ‘operator+’ that could be found by ADL. That is to be expected. Also, remember that when module M1 was compiled, Q::operator+ was not referenced in any way (especially by the first phase of lookup) and it wasn’t exported (because it was meant to be an implementation detail), so it is compiled “away”. The only way to make that line not to be an error (not that it is desired by the current design) is for a module interface to always make visible every declaration in the module interface unit to all translation units outside the module boundary. That is expressly contrary to the design.

Example 3

This is a slight variation of Example 2, now using a named function instead of an operator.

Header file X.h

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

Module M1:

```
#include “X.h”          // global module
namespace Q {
      void g_impl(X, X);
}
```

```
module M1;
export template<typename T>
void g(T t) {
      g_impl(t, Q::X{ });           // #1
}
void j(Q::X x) {
      g(x);                         // #2
}
```

Module M2:

```
#include "X.h"
module M2;
import M1;
void h(Q::X x) {
      g(x);                         // #3
}
```

Line #1 relies on g_impl being found by ADL. So in that translation unit, line #2 is OK. However, line #3 is ill-formed for the same reason as for the operator case. Note however that if line #1 is modified to explicitly qualify g_impl as 'Q::g_impl', all both #2 and #3 are equally well formed. The upshot is that if you're exporting a template and you are using an implementation detail as a customization point to be found by ADL, you need to consider two things:

1. Do you have the right structure/design for your code
2. If you still want to do that, then you need to ensure that you express that intent in the code. For example g@M1 could have been written

   ```
   export template<typename T>
   void g(T t) {
         using Q::g_impl;
         g_impl(t, Q::X{ });
   }
   ```
   This is idiomatic C++ since C++98.

Example 4

This example involves four translation units.

Module A:

```
module A;
export template<typename T>
void f(T t) { t + t; }
```

Module B:

```
module B;
import A;
export template<typename T>
void g(T t) { f(t); }
```

Module C:

```
       struct S { };
       S operator+(S,S);
       import B;
       module C;
       export template<typename T>
       void h(T t) {
             g( (t, S{ }) );         // argument is comma expression
       }
```

Main translation unit:

```
       import C;
       void i() {
             h(0);
       }
```

In the main translation unit, h@C is instantiated with #1=int. That in turn triggers the instantiation of g@B with #1=S (context of instantiation is in module C), which triggers instantiation of f@A (context of instantiation is in module B). The instantiation of f@A needs to consider the namespace partition associated with S. Because could be defined in many other translation units, we can't do an unbound search. Rather, the only relevant translation units here are the ones containing the chain of instantiation requests. So, when instantiating f@A in module B with #1=S, the second phase lookup (ADL) considers the global namespace partition from the module unit of C.

Proposed resolution

Define associated namespace partitions, extend the second phase of ADL to consider associated namespace partitions as appropriate.

Add section 3.4.2 as follows.

Paragraph 3.4.2/2:

> For each argument type T in the function call, there is a set of zero or more *associated namespaces* (7.3) and a set of zero or more *associated* ~~*classes*~~ *entities* (other than namespaces) to be considered. The set 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 namespaces 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 (7.3.1), 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.

[Example:

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

              // Interface unit of M1
              #include “X.h”          // global module
              namespace Q {
                     void g_impl(X, X);
              }
```

```
module M1;
export template<typename T>
void g1(T t) {
      g_impl(t, Q::X{ });           // #1
}
export template<typename T>
void g2(T t) {
      using Q::g_impl;
      g_impl(t, Q::X{ });           // #2
}
void j(Q::X x) {
      g1(x);                     // OK: g_impl found at #1
      g2(x);                     // OK: g_impl found at #2
}

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

}
```

--end example]

Modify paragraph 3.4.2/4

When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (3.4.3.2) expect that:

— Any using-directives in the associated namespace are ignored

— Any namespace-scope friend 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 (11.3).

— All names except those of (possible overload) functions and function templates are ignored.

— Any function or function template that is owned by a module M other than the global module (7.7), 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.

Replace 14.6.4.1 paragraph 7 with:

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.

Add a new paragraph to 14.6.4

— Declarations from namespaces associated with the types of the function arguments both from arguments from the instantiation context (14.6.4.1) and from the definition context.

[Example

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

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

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

--end example]

[Note: [Example:

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

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

      // Module interface unit of C
      #include <string>       // not in the purview of C
      import B;
      module C;
      export template<typename T>
      void h(T t) {
            g(std::string{ }, t);
      }
```

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

--end example]

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