Document Number: P0923r0
Date: 2018-02-08
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: Working Draft, Extensions to C ++ for Modules, n4720

Modules:Dependent ADL

Nathan Sidwell

The Modules TS extends Argument Dependent Lookup rules for modules. These changes make more things visible from a module interface unit than any other mechanism.

1 Background

The modules-ts adds the following final bullet to [basic.lookup.argdep]/4:

> In resolving dependent names (17.6.4), any function or function template that is owned by a named module M (10.7), that is declared in the module interface unit of M, and that has the same innermost enclosing non-inline namespace as some entity owned by M in the set of associated entities, is visible within its namespace even if it is not exported.

This has the effect of making all internal-linkage functions of the relevant namespaces visible to instantiation-time ADL. Such functions are not visible, even in module implementation units, via regular lookup, or non-dependent ADL. This is surprising.

Other than the surprise at having more things visible, there is a compilation optimization issue. It is now no longer possible, in general, for the compilation of the module interface unit to know the complete set of callers of internal-linkage functions. This calls for implementations to give all such functions an (ABI-specified) externally reachable symbol. Existing diagnostics about such functions being defined, but unused, within a single translation unit, would no longer be possible. An existing optimization is also made more complex. Internal-linkage functions may be interlined into their only, or few, caller(s). The above change inhibits such possibilities (it is usually code-growth limited) – or at least postpones them to Link Time Optimization (at the cost of making LTO more expensive).

In C++98, internal-linkage functions did not participate in dependent ADL. This was changed in C++11, and they became visible. Of course, if an instantiation selects such a static function in one translation unit, it becomes very easy to have undiagnosed undefined behaviour, because instantiations in other translation units cannot select that same static function.

1.1 Example 1

The new lookup rule affects the following example:

```
// module interface unit
namespace X {
  void Foo (int); // extern-linkage #1
}
export module Frob;
namespace X {
  export class Y {...};
  export void Foo (Y &); // external linkage #2
  void Foo (float); // module-linkage #3
  static void Foo (double); // internal linkage #4
}

// importing translation unit
import Frob;
void OneFish (X::Y &p) {
 Foo (p); // #2 found
}
template <typename T> void TwoFish (T &p) {
 Foo (p); // #2, #3, #4 found at instantiation below
}
template void TwoFish (X::Y &);

// module implementation unit
module Frob;
void RedFish (X::Y &p) {
 Foo (p); // #2, #3 found
}
template <typename T> void BlueFish (T &p) {
 Foo (p); // #2, #3, #4 found at instantiation below
}
template void BlueFish (X::Y &);
```

Notice:

- Global module declaration #1 is not visible at any of the calls (it is not owned by `Frob`).
- Module-linkage declaration #3 is visible from the module implementation unit, and at instantiation.
- Internal-linkage declaration #4 is visible during instantiation.

Although the example defines the templates in the same TU as the instantiations, the results are the same, if they were brought in via an independent import.

In a non-module world, if the module interface file were instead a header file and #included appropriately, all four declarations would be visible at all four ADL-applicable calls. (Of course placing a static function in a header file would be very strange.)

The example, I think, shows two surprises:

1. Template instantiation ADL sees more functions than non-template ADL, even when the contexts are the same.
2. Template instantiation ADL sees more functions of an implementation unit's interface unit than may observed by other means.

1.2 Example 2

Lookup of dependent function calls is performed twice, in the current standard:

> Such names are unbound and are looked up at the point of the template instantiation (17.7.4.1) in both the context of the template definition and the context of the point of instantiation. [temp.dep]/1

However, in C++17, the second lookup will always find a superset of the first, and none of the compilers examined (EDG, Clang, GCC)[1] implement the first lookup. This would no longer be the case with modules, as the template definition context might have a different set of visible imports. Again, the question arises as to whether internal-linkage functions are visible at that lookup. Here is an example:

```
// module interface
export module TPL;
namespace X {
  static int Frob (int f) { return f;} // #5
}
export template <typename T> int Foo (T t) {
  return Frob (t); // #6
}

// module user
import TPL;
namespace X {
  class Y {
    operator int () const {return 0;}
  };
}
int Func (X::Y &y) {
  return Foo (y); // #7
}
```

1 Comments by the various compiler developers at CWG in Albuquerque'17.

Should declaration #5 be found during the instantiation of `Foo<X::Y>` at #7 because the definition context is #6?

Making such entities visible from a module interface unit presents the same optimization (and diagnostic) issue as mentioned above, with the new bullet of [basic.lookup.argdep]/4. There may be less programmer confusion in this case, because the context of the lookup is that of the template itself. However, that internal-linkage functions in unrelated namespaces are visible seems somewhat surprising.

1.3 Module-linkage Using-declarations

One aim of adjusting the ADL rules, is to permit a module interface unit to selectively make visible functions from its global module portion at customization points of templates instantiated by importing translation units. This can be achieved with *using-declarations*. For instance the first example above could make declaration #1 visible by adding:

```
export using X::Foo;
```

in the purview of `Frob`'s interface unit. This does not require the semantics of the new bullet though.

Perhaps it worth considering whether module-linkage entities are visible during all ADL outside of the owning module. That might allow an interface unit to make such a global-module entity only visible during ADL, via a module-linkage using-declaration:[2]

```
using X::Foo;
```

Such a mechanism would allow module authors to provide customization points to importers only locatable via ADL.

2 Proposal

I propose two changes:

2.1 Internal-Linkage Entities Cannot Be Found From Outside

Internal linkage entities, declared in a module interface unit purview are never visible to lookups initiated from outside the module interface translation unit.

By that I mean both examples above would not see the internal linkage functions #4 & #5 during template instantiations. Although #5 is visible during template definition #6, the ADL performed in that context is initiated from outside the module unit.

2 The linkage of the *using-declaration* itself need not match that of the entity it brings into scope.

This does not restrict the use of internal-linkage entities in other entities that are made visible, for instance allowing the following:

```
export module Foo;
static int Frob (int f) {return f;}
export inline int Flange (int f) { return Frob (f); }
export template <int I> int Spelunk (int f) {
   return Frob (f + I); // Non-dependent, no ADL
}
```

To be clear, I am not proposing reversion of the C++11 change making internal-linkage entities visible to ADL. Such visibility was only within a single translation unit, and useful for a TU to provide ADL candidates to TU-specific instantiations of templates brought in by header file, or locally declared. Internal-linkage functions visible at the instantiation context remain visible.

2.2 ADL is Consistent

The new bullet should apply to all ADL, regardless of whether the call is dependent or not.

This would have the effect of adding declaration #3 to the overload set of the non-dependent call in the translation unit importing `Frob`, above (the call in `OneFish`).

3 Changes to Modules TS

Modify the new bullet added to [basic.lookup.argdep]/4:

1. The functions and function templates must have with external or module linkage (including using declarations bringing such entities into the module purview)
2. Remove its restriction to dependent name resolution.

Amend [temp.dep]/1:

1. Make internal-linkage functions are not visible during the ADL occurring within the definition context.

Note that this change does not materially affect non-module sources, because the ADL occurring at the instantiation context will still see those internal-linkage functions in the current TU.