Document Number: P0923r1
Date: 2018-05-04
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 are compilation issues. 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. Some existing optimizations are also made more complex:

- Determining if a particular call is the only (remaining) call of a particular internal-linkage function. This is a very strong candidate for inlining, and programmers expect for it to be inlined – and therefore the function abstraction to have *zero* overhead.
- If all call sites are visible with the callee, compilers may choose an ad-hoc parameter and result passing mechanism – for instance use additional registers, or pass by-value. They may also reduce any stack-frame spilling as the live registers at the set of call sites is known.

The above change inhibits such possibilities – 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. The rationale for that was to support exported templates, and avoid the same set of difficulties this paper discusses. This was changed in C++11, and they became visible, as exported templates were removed from the language.

Another change that occurred between C++98 and C++17 was the linkage of anonymous namespace members. In C++98 they were not necessarily internal-linkage – the anonymous namespace was merely a uniquely & unspellably named namespace, (coupled with a using directive). In C++17 members have internal linkage. Because of the new visibility of internal-linkage functions to ADL, change in the above lookup rules, the anonymous namespace change had no affect on ADL behaviour.

National Body comment US 041 raised similar issues, and was discussed at the Albuquerque'17 meeting. Four straw polls were taken, to answer:

> Should $ACTION, be able to find names of entities owned by an imported module which were not explicitly exported but have $LINKAGE (and as otherwise already constrained by 6.4.2p4)

over the following matrix of values.

Should ↓ See →	**(Non-Exported) Module Linkage**	**Any Linkage**
Second Phase ADL	SF:6 | F:9 | N:3 | A:2 | SA:0 Yes	SF:0 | F:0 | N:4 | A:13 | SA:2 No
All ADL	SF:0 | F:0 | N:3 | A:11 | SA:5 No	SF:0 | F:0 | N:0 | A:11 | SA:6 No

Presentation and discussion of this paper at the Jacksonville'18 meeting. There seemed general unease that in Example 1 below the importing translation unit's OneFish & TwoFish functions and the implementation unit's RedFish & BlueFish functions saw different candidates in the overload sets of the the pairs of `Foo` calls. It was resolved that a working group of Gaby dos Reis, Richard Smith, John Spicer and Nathan Sidwell should examine this issue and report.

Unfortunately the group has had insufficient time to reach a conclusion. This updated paper extends the motivating examples to explicitly include unqualified lookup and anonymous namespaces.

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.

2 Discussion

Function calls of the form:

```
foo (arg0, …, argN);
```

incur up to 3 different lookups for `foo`:

- An unqualified lookup of '`foo`' starting at the current scope. If this finds something that is not a set of namespace-scope functions, it is the only lookup performed.
- An ADL lookup of '`foo`', from the containing function's definition context, using the associated classes and namespaces thereof of the argument types. If the call is dependent, this lookup occurs at instantiation time.
- For a dependent call, an ADL lookup of '`foo`', within the instantiation context is also performed.

All three lookups need considering in the context of modules. Dependent calls are more complex, due to their deferred resolution nature. They have the potential of speculatively exposing functions that might never be chosen in any instantiation.

2.1 Goals

In extending dependent-call lookup into modules, it would be good to achieve the following:

- Preserve existing C++17 behaviour when all entities involved are owned by the current translation unit. This means that (non-module) legacy code will not change behaviour, and within a module, the behaviour will be 'familiar'.
- Not expose internal-linkage entities outside of a module-interface unit without direct use of that entity. i.e., a specific entity may be referenced by some exposed context (e.g. body of exported inline function), but including such an entity in a set to be chosen from elsewhere may not be.
- ADL behaviour is not affected by the dependency of the ADL. Resolving a dependent ADL may look in additional places to a non-dependent ADL, but the rules by which it finds things there are the same as non-dependent ADL.

2.2 Unqualified Lookup

If the unqualified lookup of the function name in a dependent call sees internal-linkage functions, it reduces the optimization possibilities for those functions, as described above. It will not be known until instantiation whether the internal-linkage function is selected. (The internal-linkage function will also be subject to linkage-promotion.)

It may be wise to remove such internal-linkage functions from the overload set, within a module (interface) purview. Legacy code would not be affected as that could not be within a module purview. Further, global module contents, which is intended for inclusion of legacy headers, would not be expected to declare internal linkage functions.

2.3 Anonymous-Namespace Functions

Anonymous-namespace functions have internal-linkage. It will be least confusing if the visibility of such functions is the same as an explicitly '`static`' function in a named namespace. Presuming such explicit declarations are not visible to extra-translation-unit ADL, anonymous-namespace functions will also not be available to such ADL. This may seem overly restrictive, preventing a mechanism by which a module may make functions *only* visible to ADL from importers. But modules provide module-linkage, which could be extended to permit such ADL visibility (as one of the above straw polls suggested).

2.4 Anonymous-Namespace Types

Classes defined in an anonymous namespace have internal-linkage. They may be used in a non-anonymous-namespace function signature:

```
namespace {
  class X;
}
void Frob (X *);
```

Although `::Frob` is an external-linkage function, its signature not accessible from outside this translation unit (it cannot be declared in multiple TUs). Both Clang and G++ effectively give it internal linkage. Consider the following use of such a type:

```
export module Foo;
namespace {
  class X { … };
}
export X *Get_X () { … }
export void Frob (X *h) { … } // #1
```

It would be strange for #2 to be a well-formed declaration, but not reachable from elsewhere (either by ADL or regular lookup).

2.4.1 Implementation Wrinkle

Making ‘`::Frob (<anon>::X *)`’ externally reachable requires giving it a unique link-level symbol. Prior to anonymous-namespace entities having internal linkage, this was achieved by giving the anonymous namespace a program-wide unique name. However, that can prove difficult to guarantee in some cases, and G++ (at least) reverted to giving a specific fixed name to the global namespace, safe in the knowledge that it was used to name internal-linkage entities, or any use of it would make the entity effectively internal.

As ‘`class <anon>::X`’ may be in the global module, the module name is not necessarily available at the point we first need to use it in determining a symbol name. However, as with explicit static function declarations, it would be strange for a legacy header file to declare anonymous namespace entities. But as with the explicit case, a behaviour will need specifying, and unless it is defined as ill-formed, compilers will need to implement it.

3 Proposal

No specific proposal is made at this time.

4 Document History

R0, 2018-02-08 Presented Jacksonville’18.

R1, 2018-05-04 summarize Jacksonville discussion, add US041 polls from Albuquerque’17. Remove using-declaration discussion. Expand examples to anonymous namespace.