Document Number: p1347r0
Date: 2018-11-21
To: SC22/WG21 EWG
Reply to:, Davis Herring, Nathan Sidwell
herring@lanl.gov, nathan@acm.org
Re: p1103r1 Merging Modules

Modules: ADL & Internal Linkage

Nathan Sidwell, Davis Herring

The merging modules proposal, P1103r1, specifies that internal linkage functions declared in a module interface unit are available to ADL from outside the interface. This breaks a long-understood meaning of internal linkage, and significantly affects compiler inlining heuristics. This paper advocates for permitting them to be visible, but not callable.

1 Background

At the Jacksonville'18 meeting p0923r0 was presented concerning the dependent ADL rules of the Modules-Ts. P1103r0 revises the ADL rules, addressing many of the issues raised by p0923r0. However, one issue remains, that of internal linkage functions from ADL initiated outside of the module interface.

2 Discussion

Consider:

```
export module foo;
namespace impl {
  class Foo {/*details*/};
  static int frobber (int) {/*details*/} // #1
}
export impl::Foo getafoo ();

// elsewhere
import foo;
template<typename T> int frobbit (T const &t) {
  return frobber (t); // #2
}
void quux () {
  frobbit (getafoo ()); // #3
}
```

The instantiation of `frobbit<impl::Foo@foo const &>` has an ADL lookup at #2. One of the locations to search is `impl@foo`, thus the declaration at #1 is found, and may be selected if a suitable conversion sequence to `int` can be found.

The implementation difficulty is that the compiler of module foo's interface no longer knows the call graph that `impl::frobber@foo` participates in. It has to be pessimistic and emit a definition of that function that is globally reachable. This interferes with its inlining and warning heuristics.

Typically compilers will warn about defined-but-unused internal-linkage entities. These warnings would no longer be available form module-interface internal-linkage functions.

Inlining heuristics usually avoid code-duplication without good cause (for instance having profile data). Thus, functions with unknown call graphs are only inlined if (very) small. Module interface internal linkage functions would rarely be inlined, if a template definition in the interface *might* refer to it even indirectly..

The claim was made that making internal linkage functions callable by such ADL would not affect the compiler's ability to inline them within the module interface. That is correct, but misses the point. It affects the compilers decision as to *whether* inlining would be profitable. The comment was made that:

> Users expect a local function with a single call location to have *no* overhead. Not small, or insignificant, but zero. [Not a direct quote, Chandler Carruth]

Without resorting to Link Time Optimization, the compiler of the interface has insufficient information to do that.

The issue was discussed briefly at the San Diego'18 meeting, with the focus on anonymous namespace members. A comment was made that although such members have internal linkage, users generally use anonymous namespaces to provide a globally unique name component. The internalness of the contained entities is merely happenstance that the compiler may then take advantage of.

We find this argument lacking for two reasons:

1. Implementors have changed how anonymous namespaces are implemented since C++98
2. Modules provide a new way of obtaining globally-unique names

In C++98 anonymous namespace members had external linkage[1]. Implementors had to provide some unique mangling in the symbol names of such entities. Generally choosing a known-unique definition, hash from source tokens, file-name or randomization. All have deficiencies, particularly those that break repeatable builds. In C++11, the members were given internal linkage. Implementors no longer

1 This made their members visible to exported template instantiations. Compilers that did not support export could give them local symbol linkage and optimize on that basis (at the expense of cross-compiler ABI incompatibility). Explicitly 'static' declarations were not visible to such exported template ADL.

needed to give uniqueness to the symbol name, instead relying on underlying local linkage semantics. Thus anonymous namespaces no longer provide unique symbol names.

With modules, we now have module-linkage. This provides the guarantee that functions in different modules, with the same signature and namespace scope will nevertheless be distinct. Implementations may introduce new object-file technology or leverage existing external-linkage by adding additional mangling to the symbol names.

If internal-linkage functions are accessible to extra-interface ADL we take away the user's ability to create truly interface-private functions. We do not gain anything in the tradeoff – module-linkage is still available to provide extra-interface ADL customization points. And as already mentioned, provides the desired uniqueness.

3 Proposal

We propose altering the p1103 semantics such that internal linkage functions are still visible to extra-interface (dependent) ADL, but if selected via overload resolution the program is ill-formed. (If they make overload resolution ambiguous, the program is also ill-formed.)

Making them visible has three purposes:

1. It is harder for specializations within the interface to be different from extra-interface specializations
2. Users will not be confused by expecting a local linkage function to be selected as a customization point – they'll get a diagnostic rather than selection of some other function.
3. We can relax this rule in the future, if it is found problematic, knowing we cannot be breaking any currently-valid source.

Making the program ill-formed, if selected, allows the interface-unit compilation to determine the call graph of internal-linkage functions. Thus not perturbing its existing inlining heuristics.

3.1 Non-dependent ADL

Non-dependent ADL is not changed. This means that existing, non-modular, C++17 code is unchanged. One new situation does arise, and would prevent a compiler determining the call graph of module interface local functions. Consider:

```
export module M;
namespace N {
  struct S {};
  static void foo(S,int) {}  // #1a
}
```

```
export template<class T>
void bar(T t) {
  foo(N::S(), 0);    // #4a: non-dependent
}
```

The call #2 is non-dependent, and resolved at template definition time. Its presence makes the call graph of #1 incomplete. This is exactly the same as if an inline function with external or module linkage called #1.

3.2 Dependent ADL

Dependent ADL may see a superset of functions that a non-dependent call might see (because it considers more contexts), but it will not see a subset. Consider:

```
// TU 1
export module M;
namespace N {
  struct S {};
  static void foo(S,int) {}  // #1b
}

template<class T>
void foo(N::S,T) {}  // #2b

export template<class T>
void bar(T t) {     // #3b: template definition
  foo(N::S(),0);    // #4b: non-dependent, 1b well-formed
  foo(N::S(),t);    // #5b: dependent
}

// TU 2
import M;
void baz() {
  bar(0);  // #6b: binds to #1b (ill-formed)!
  bar(nullptr);  // #7b: binds to #2
}
```

This builds on the previous example, and the non-dependent call at #4b is the same as #4a, and calls the internal-linkage function. The dependent call #5b is resolved in the two instantiations of bar at #6b and #7b. Both those instantiation see #1b and #2b in the overload set. The instantiation caused by #6b resolved to #1b, and is ill-formed. That caused by #7b resolves to an instantiation of #2b and is well-formed.

That `bar` contains a well-formed call of #1b could lead to user confusion, the diagnostic for the failed call should make it clear why the call is ill-formed.

Notice that if the `baz` was defined later in the interface of `M`, both the calls of bar would be well formed. Thus we may still have differing instantiations, but only one of those instantiations succeeds, the others fail with a diagnostic. If internal linkage functions were not visible, it is easier to have differing well-formed instantiations (violating the ODR, no diagnostic required).

4 Wording

These wording changes are relative to p1103r1.

[lex.separate]/2: Note: already omits internal linkage.

[basic.lookup.argdep]/4.5: Append:

> If the function or function template selected by overload resolution has internal linkage and is declared in a translation unit other than the one containing the function call, the program is ill-formed.

[basic.link]/2.2: Append:

> The entity may also be referred to by names subject to argument-dependent lookup where the instantiation context includes a point in the translation unit.

[temp.dep.candidate]/1: Remove

> with external or module linkage

Consider rephrasing "introduced in those namespaces" in light of inline namespaces.

5 Revision History

R0 First version