Document Number: p1395r0
Date: 2019-01-18
To: SC22/WG21 EWG
Reply to: Nathan Sidwell nathan@acm.org
Re: p1103r1 Merging Modules

Modules: Partitions Are Not a Panacea

Nathan Sidwell,

The merging modules proposal, P1103r1, specifies module partitions as a mechanism to separate pieces of a module implementation without exposing potentially ABI-breaking symbols to the module user. Unfortunately there are cases where this is inevitable.

1 Background

Module partitions are essentially separate modules, except they bestow the same module ownership on any entities they declare. For external- and module-linkage entities this presents no problems, as there can be exactly one definition. Moving a such a declaration or definition from one partition to another does not change the module ownership.

Internal-linkage entities can become accessible in module importers via linkage-promotion, or appliance of `decltype`. Either requires the entity to have a globally-unique symbol name.

Some C++ module implementations do not presume new linker technology, augmenting name mangling to obtain module-unique symbols. This document restricts itself to such a regime.

2 Discussion

The cases that present difficulty are where an internal-linkage entity must be exposed in a globally unique manner. Two examples are:

```
export module Foo:part.a;

static void InternalFn () { … }

export inline void Frobber () {
  InternalFn (); // #1
}
```

and

```
export module Foo:part.b;
```

```
namespace {
  class InternalType { … };
}

export InternalType Badger () { // #2
   return InternalType ();
}
```

In the former case, importers of `Foo` that call `Frobber` will need to generate a body that function that calls `InternalFn@Foo:part.a`. In the latter case, users can get at `<anon>::InternalType@Foo:part.b` and reference its members.

In both case, the entity's linkage is effectively promoted to module linkage.

In these cases the primary module name is insufficiently distinct, as another partition of the same module could expose its own variants of similarly named internal entities. To disambiguate these requires use of the partition name in the externally visible symbol, and thus subsequently moving the entities to a different partition will change this symbol. Alternatively, not incorporating the partition name will restrict the module author to not promoting the linkage of otherwise identically named entities.

Note that the two cases are slightly different. The function promotion can be discovered lazily, and symbol aliasing could be used to generate a globally-accessible symbol. The anonymous namespace-scoped type cannot be processed lazily. Its name is needed in the naming of any template specializations it may participate it (amongst other cases).

2.1 Implementation Partition & Dependent ADL

'Modules: ADL & Internal Linkage', p1347r1 discusses how internal linkage functions might be visible in dependent ADL of extra-module instantiations. Module partitions present a further wrinkle.

Consider an implementation partition:

```
module foo:impl;

namespace details {
  static void Frob (int) { ...} // #1
}
```

and its associated primary module interface:

```
export module foo;

import :impl;  // #3 … or …
import :indirect; // #4 (which itself imports :impl)
```

```
namespace details {
  struct X {…};
}

export X *X_getter ();

template<typename T> void frobber (T *p) {
  Frob (p); // #2
}
```

Presume one of either #3 or #4 is in effect, where #4 indirectly imports `foo:impl`. When instantiating of `frobber` with `T` = `details::X@foo`, is the definition at #1 visible to ADL lookup at #2? Does direct or indirect partition importing make a difference?

For avoidance of doubt, were `foo:impl` an interface partition, I believe the answer to these questions should be however p1347 is resolved – the definition #2 behaves as-if it appeared directly in the primary interface. This includes the case of the function definition being in an anonymous namespace, but in that case the type being instantiated over, `<anon>::X`, and the means by which importers could access it, would need to be declared in the same interface partition as partitions do not share a single anonymous namespace.

2.2 Anonymous Namespace Symbol Names

The internal-linkage type issue mentioned above could be solved by giving anonymous namespaces symbol names that incorporate the partition name (and subsequently using that in the symbol manglings of any namespace members). This clearly makes the symbol name change if the type is moved to a different partition.

An alternative might be to decorate the type's symbol name with the partition name, thus only affecting anonymous-namespace types.

Implementation-wise the former is simpler, and may well be less confusing to users inspecting demangled symbols.

3 Proposal

The tradeoff to be made is between:

1. Unrestricted linkage promotion at the expense of restricting refactoring of module partition source that contains such promotion.
2. Unrestricted module partition refactoring at the expense of linkage promotion collisions.

When considering an entity achieving internal linkage via the ‘static’ keyword, option #2 appears the better alternative. It will future-proof code bases against unplanned refactorings. The cost is that one cannot, in general, define static linkage entities with colliding signatures in different module partitions.

However, entities can have internal linkage via membership of an anonymous namespace. Choosing option #2 means that symbol collisions could occur on such entities too. Including the indirect case of members of a type within the anonymous namespace.

Option #2 has the advantage that it protects against a future unforeseen circumstance, whereas option #1 allows unrestricted code development.

Note that option #2 does not protect against a future change that introduces a *new* linkage promotion colliding with an existing one. However, that can be worked around by changing the name of the newly promoted entity. That rename cannot be an ABI breakage, because the name was heretofore never made visible outside of the translation unit containing it.

The collisions that can occur with #2 require the module author to use the same source-level names for different entities. This is likely to be a source of bugs, regardless of modules.

4 Revision History

R0 First version