Modules ABI Implications

2017-07-21 – Nathan Sidwell nathan@acm.org

2017-03-01	Original document
2017-03-07	More thoughts on sub-module mangling, squangling and debuggers. Feedback from Gaby dos Reis & Jason Merrill
2017-03-27	Updates following meeting with Richard Smith, March 22nd.
2017-07-21	Module mangling binding, linkage promotion following C++ Toronto '17 meeting.
2017-09-01	Improve & clarify module-subst mangling.

1 Background

C++ modules defines a new kind of symbol linkage – module-linkage. This document discusses what exactly this means in implementation terms.

All objects[1] are owned by a module:

- Objects not in a named module are contained within the global module.
- Objects might be exported from a module, and therefore available once the module has been imported. The exception is the global module, where nothing may be exported.

1.1 Target System

The target object and executable system for discussion is ELF-like. That is, I presume ELF & DWARF features, but don't think any obscure features of ELF are needed, and the discussion could readily translate to other object and executable formats. Further, as a design constraint I am not making any additions to existing ELF or DWARF features.

That is, modules should just slot into an existing system where symbol name mangling is the only mechanism available for distinguishing objects at the object-file level.

2 Motivating Examples

I often find concrete code examples illuminating. Here are some.

1 This document uses 'object' to refer to both variables, types and functions. Generally anything requiring a name visible from other translation units.

2.1 Identically Named Module-Linkage Functions

The proposal is clear that module-linkage names are only visible within the set of translation units comprising the module's interface and implementation. Thus we are free to name such functions and objects without concern to other modules:

```
module A1; // implementation
void Init () {}
```

```
module A2; // implementation
void Init () {}
```

Both `Init` functions can reside in a final executable without error. However both may be (independently) accessed from other translation units of the same module.

I include this example for clarity.

2.2 Two Modules Exporting Same Name

Consider an all-module world where two different modules export identically named functions:

```
module A1 [[interface]];
export void Init ();
```

```
module A2 [[interface]];
export void Init ();
```

Clearly importing these into the same translation unit is going to lead to tears. But consider they are independently imported as internal implementation components of two other modules:

```
module B1; // implementation
import A1;
// exported in B1's interface
void B1::Init () {
  ::Init ();
}
```

```
module B2; // implementation
import A2;
// exported in B2's interface
void B2::Init () {
  ::Init ();
}
```

Here modules A1 and B1 work together and A2 and B2 similarly. Users of B1 and B2 are unaware of the respective dependencies on A1 and A2.

Now consider a user program importing both B1 and B2:

```
import B1;
import B2;
int main () {
  B1::Init ();
  B2::Init ();
  return 0;
}
```

- The proposal is mute on whether this is ill-formed or not.
- Do we want this to work – i.e. can module A1’s `Init` and A2’s `Init` coexist in a final executable?

2.3 Header Files in Modules

The switch to modules is not going to be an atomic operation. There will be a transition period, and if history teaches us anything, that is going to be a very long time. Thus we will need to support code bases containing a mixture of old-world #includes and new-world imports.

Consider a library that is modularized, but is reliant upon a library that is not modularized. For simplicity let’s presume that’s the standard library. The non-modularized libary’s header files will have to be #included in the global-module portion of the modularized translation units:

```
#include <iostream>
module Foo; // implementation (but applicable to interface too)
void Print (char const *text) {
  std::cout << “She said ‘” << text << “’\n”;
}
```

- This example must work. i.e. the references to global module objects from a module-aware compilation must be satisfied by definitions from a module-unware compilation.

2.4 Old-World & New-World Users

A slightly different case is where it is the users of a library that is in a state of transition. Consider the case of `foolib`, a library that provides both old #include headers for legacy users and a new module interface for new users.

Now consider two further libraries, both dependent on foolib. One, oldlib, uses `` `#include <foolib.h>’ `` to get at foolib’s goodness. The other, newlib, has been converted to use `` `import foolib;’ ``

Finally, what if the main executable (or any other component) wants to use both oldlib and newlib?

We cannot have two instances of foolib in the executable – one supporting old-world users and the other new-world users.

- Do we want this to work – i.e. can a single instance of foolib support both old-world and new-world users?

2.4.1 Hidden Import Solution

One way to guarantee functionality is to simply have the body of `foolib.h’ be:

```
      import foolib;
```

However, for the purposes of this discussion, I consider this sleight of hand. As thus, the question needs clarification.

2.4.2 Backwards Binary Compatibility

To refine the question, let us first consider the case of a pre-built archive of object files, and an associated non-module header file describing its interface.

Now presume foolib has been updated to provide a module-interface, but without adding any extra features etc. Compilation of the library produces a new archive, which we can refer to as `foolib[mod].a’.

- Should `foolib[inc].a’ (original non-module library) and `foolib[mod].a’ be binary compatible?

I.e. can we link the final executable’s use of oldlib and newlib with either foolib[inc].a or foolib[mod].a?

3 Discussion

The first example shows that non-exported objects must have a mangled name that encodes the module name in some way. Alternatives are unattractive:

- Require module-linkage symbols to have hidden visibility. This would force modules to reside only in shared objects, and there be a 1:1 correspondence between shared objects and modules.
- Invent a new ELF symbol name lookup. Forcing users to update their static and dynamic linker technology will make modules wither on the vine.

What exactly this mangling is, is implementation specific.

Supporting the second example, of identical exported names, would also require mangling of exported names to encode the module name. This is certainly a possibility. However, it is not a new problem and C++ already provides a mechanism to avoid the problem – namespaces, and that is how the user modules B1 & B2 avoid the problem. We should continue to recommend that approach do disambiguating names in a library’s interface.

The third example demands that the mangling of global-module objects have the same mangling as a non-modular compilation.

The fourth example is also part of the transition path from #include to modules. It is impractical to demand that the end user require oldlib be converted to module form, before they can rebuild their code. There may be hundreds of cases of oldlib, and being unable to incrementally convert each one will result in nothing being converted. While the trick of having the include file simply import the module works, let us consider the more involved task of having binary compatibility to an already build archive.

The old-world header file will behave in a module-aware compilation as-if it is a partition of the global module. Thus to support this use, the mangling of exported symbols must match the mangling of symbols from the global module. This is the same requirement as the second example. It resolves the binary compatibility issue, as all symbols that are user-visible have unchanged manglings.

To increase flexibility, it is tempting to introduce an attribute that controls whether the module name is part of the mangling of an exported symbol. While this may provide assistance to implementors of compilation systems, let us eschew it unless it is proved necessary. Such ability is likely to lead to unnecessary complexity.

4 Recommendation

These examples suggest the following approach to symbol mangling:

- Objects in the global module have the same mangling they would have in a non-modular compilation. This satisfies the second example.
- Objects exported from named modules have the same mangling they would have if they were in the global module. This satisfies the fourth example.
- Objects that have module-linkage include the module name in their mangling. This satisfies the first example.

The second example is not supported, its requirements conflict with the fourth example and can be resolved by the use of namespaces.

5 Presentation to Users

The proposal does not give explicit guidance as to how names in different modules may be presented to a user, such that she can determine which modules are involved. As modules are not namespaces it would be confusing to hijack the qualified name syntax to show module specificity too.

An `@module' notation has been used informally, and I suggest using that to suffix the qualified name. For instance:

```
module Foo.Baz;
namespace Foo::Baz {
  void Init (); // void Foo::Baz::Init@Foo.Baz ()
  class Bob {
    void Frob (); // void Foo::Baz::Bob::Frob@Foo.Baz ()
    }
}
```

A suffix, particularly one introduced by '@', seems more natural than a prefix.

6 Additional Optimizations

The module interface TU has some unique properties, that may permit additional code generation techniques. In particular, it is known that all module implementation TUs and all importing TUs will read the internal binary module interface. Alongside that BMI will be an object file, containing conventional output from compiling the interface TU. All importers and implementors of the module will be expected to link with this object file.

Thus it provides a unique location for objects that, in conventional C++, have no unique location. Such as bodies of non-inlined inline functions, virtual tables of classes lacking key-functions (or other ABI-specific location mechanism), template instantiations, debug information for module-specific classes.

Whether emitting all these pieces in the interface TU's object file is a win probably depends on optimization level and type of data being considered. For instance, emitting non-inlined bodies is probably only advantageous at -O0. If other TUs are inlining the functions themselves, emitting and then discarding the bodies is extra work for the interface compilation and final link. Alternatively, debug information is probably always a win, when debugging is enabled.

6.1 Recommendation

It is taken as axiomatic that the BMI is generated by the same compiler that compiles any TUs importing the module. As such, it can conspire to step outside the ABI, where such behavior could not be observed by other TUs.

Until implementation experience is gained, there is no specific recommendation for additional ABI changes or enhancements.

7 Itanium C++ ABI-Specific Mangling

The Itanium ABI is documented at https://github.com/itanium-cxx-abi/cxx-abi/tree/gh-pages. This section discusses changes to that ABI to support the above features. It is desirable that all implementors of the Itanium ABI implement the same set of changes.

As recommended above, changes in mangling only occur for objects with (non-global) module-linkage. Exported objects and those within the global module retain their existing mangling. If an object's mangled name incorporates a module-linkage object, the mangling will be altered, regardless of whether the object itself is exported or not.

7.1 Hierarchy

One immediate choice is whether namespaces are inside modules or modules are inside namespaces. That is, is the compiler's internal representation consider that

- a module is fragmented across the namespaces to which contributes partitions, or
- the namespace partitions of a module are contained within the module that contributes them?

While this is clearly an internal implementation detail, it can affect the complexity of name compression (squangling) substitutions, as they are built on top of the compiler's internal hierarchy of object nodes.

Both the (in-development) G++ and Clang implementations represent a declaration's module as orthogonal data to the containing scope of the declaration.[2] Richard Smith and I consider this hierarchy to be the most natural representation.

7.2 Module Name

The obvious scheme of extending the `<nested-name>` grammar to include the module name as a component, has some disadvantages:

- It mixes the module name hierarchy with the existing name hierarchy.
- Various special-case names to compress members of the STL would become inaccessible from a modularized STL.

These seem less than ideal.

We propose making the module-name an orthogonal piece of the name mangling. A name, qualified or not, requiring a module-specific mangling would prefix its <name> component with the module-name mangling. This can be achieved by amending the mangling grammar as:

```
<name> ::= [<module-name>] <nested-name>
       ::= [<module-name>] <unscoped-name>
       ::= [<module-name>] <unscoped-template-name> <template-args>
       ::= <local-name> # no change necessary
```

The module name is introduced by a currently unassigned letter. Available upper case letters are B, J, Q, & W. Unless there are compelling reasons not to might I suggest 'W' as an inverted 'M'?

7.2.1 Sub-modules

The proposal includes the concept of sub-modules, using a dotted sequence of identifiers. While this suggests a hierarchy, that is not correct. A sub-module does not have any special visibility into its

2 Early G++ implementation represented it as a hidden namespace, but this became increasingly unwieldy.

containing module's namespace partitions – or vice versa. Sub-module names are purely a user convenience to indicate containment.

Three obvious options for sub-modules are:

- a set of W prefixes, or
- treat W as a W… E encapulation of the components of a module name, or
- flattened to a single identifier, with the '.'s optionally converted to a suitable symbol character.

Using a '.' separator is problematic, even if permitted by the object file encoding, as compilers already decorate mangled names with a '.suffix', when generating optimized clones of functions. Thus their demanglers stop at the first '.' character. Replacing '.' with some other character could lead to ambiguities between a sub-module name and a module name that happens to contain the replacement character.

Thus it is preferable to select one of the first two options. These will have the same encoding length for 2 level names ('foo.bar'). The first will be one character shorter for single names ('foo'), whereas the second will be 1 character shorter per additional level after 2 ('foo.bar.baz' etc). The second also has the advantage that it encapsulates the module as a single entity, which may make user presentation simpler.

Thus the encoding for a module name would be:

```
<module-name> ::= W <unscoped-name>+ E
```

7.3 Compression, aka Squangling

Squangling substitutions are used to reduce the textual size of a mangled name, replacing a repeated use of a name component with a back reference to an earlier encoding within the mangle. The possible substitutions are automatically numbered.

Two questions arise:

- Can (part of) a name within one module be used as a substitution for (part of) a name in another module?
- Are module-name substitutions part of the current substitution space, or an orthogonal space?

7.3.1 Cross-Module Substitutions

The discussion on hierarchy above introduces a natural cross-reference barrier of the innermost namespace-scope object. Whereas the namespaces themselves are not module-specific. Thus two identically-named types within different modules could not use a substitution for the second one, but could substitute its containing namespace. For example:

```
class Foo::Bar::Baz {…}; // module Quux
class Foo::Bar::Baz {…}; // module Fido
```

```
import Frobber; // exports template function Frob
void Frob<Foo::Bar::Baz@Quux, Foo::Bar::Baz@Fido> (T1, T2);
_Z4FrobIW4QuuxEN3Foo3Bar3BazEW4FidoENS1_3BazEEvT_T0_
```

That is, the two instances of `Baz` must be separately encoded, but the second reference of namespace `Foo::Bar` refers to the first one. The two distinct instances of `Baz` are separate substitutions themselves, and used by members of their respective classes.

The module name attaches to the final component of the unscoped-name, nested-name or unscoped-template-name that it prefixes, even if that entity is a non-exported member of an exported entity. Thus if two separate declarations within the same module are within a single mangling, they each need a module component on their first appearance. For example:[3]

```
class Foo::Bar::Baz {…}; // module Quux
class Foo::Bar::Bam {…}; // module Quux
import Frobber; // exports template function Frob
void Frob<Foo::Bar::Baz@Quux, Foo::Bar::Bam@Quux> (T1, T2);
// _Z4FrobIW4QuuxEN3Foo3Bar3BazEW_0ENS1_3BamEEvT_T0_
void Frob<Foo::Bar::Baz@Quux, Foo::Bar::Baz@Quux> (T1, T2);
// _Z4FrobIW4QuuxEN3Foo3Bar3BazES2_EvT_T0_
```

7.3.2 Module Compression

As the `<module-name>` is a separate prefix within a <name>, it would seem any compression scheme should be orthogonal to the current scheme. This can be achieved by allowing back references within the <module-name> component itself. The module name grammar would become:

```
<module-name> ::= W <unscoped-name>+ E
              ::= W <module-subst> <unscoped-name>* E

<module-subst> ::= _ <seq-id>    # single character zero-based seq-id
               ::= W <seq-id-10> _   # otherwise4
```

As with the existing substitution scheme, only a leading sequence of sub-module components would be eligible for substitution. And, similarly, each sub-module component is implicitly added into the module substitution table.

3 This example employs the module compression scheme.
4 We could use the specific uppercase letter after the initial ‘W’ to indicate the length of the `<seq-id>`, but that seems like overkill.

7.4 Linkage Promotion

A module interface unit may export an entity that itself refers to internal linkage entities. When the exported entity is an inline function it is desirable for importers to be able to inline it. When it is a template definition, it is necessary for importers to be able to instantiate it. Both uses require the internal linkage entity to be referenceable from outside the module interface translation unit. For instance:

```
export module Foo;
static int my_counter;  // #1
export inline int count () {
  return my_counter++; // #2
}
```

It is only at the point of use (#2) that we learn of **`my_counter`**'s need to be referenceable outside the translation unit. At that point the compiler might have already emitted assembly code or mangled the name of `my_counter`. A simple solution would be to pre-emptively mangle all such names as-if module-linkage and promote the object-file visibility of the symbol.

However, this is inadequate for a number of reasons:

- It might unduly pessimize optimization opportunities for internal linkage entities that do not need to be made visible.
- It breaks internal-linkage within the module itself – an implementation unit could legitimately make a similar-named entity with module-linkage.
- The internal-linkage entity might be within the global module fragment of the interface unit, where it is impossible to add the module-specific mangling, as the module name is not yet known.

An alias mechanism can be used to retroactively promote internal-linkage symbols to global-linkage at the object file level. Promotion can be determined as early as parsing an exported entity referring to the internal entity, or as late as streaming the exported entity out. The module name will be known at that point.

The name mangling used for the alias cannot simply be that of the promoted entity as-if it has module-linkage (i.e. insert a `<module-name>` component). As mentioned above, that would collide with a module-linkage entity of the same name (and possibly type, if a function). Such an entity cannot reside in the interface unit, but could reside or be referred to in implementation units.

I suggest adding a **`<module-name>`** component as-if the module name consists of an additional sub-module component with an empty name, which is unspellable at the source level. Such a component has the mangling '`0`'. For instance:

```
static int my_counter; // #1
export module Foo;
export … { … my_counter … } // #2 needs promotion
// #1 ~~emitted as ~~local symbol '_Z10my_counter' (implementation-defined)
// #2 promoted to global symbol '_ZW3Foo0E10my_counter
```

7.4.1 Promoted Types, Enumerations etc

A number of entities that formally have internal linkage, but themselves have no object-file symbol, such as class and enumeration members of an anonymous namespace also may require promotion. However their names may be mangled into other symbols before promotion is determined. For example:

```
#include <typeinfo>
export module Foo;
namespace {
  class X { virtual ~X ();};
}
export inline std::type_info &Frob () { return typeid(X);}
```

Class layout, including vtable layout will be determined at the point `X` is complete. This might include mangling for any typeinfo variable. The same promotion-via-alias can be used for such entities.

The internal-linkage name might be used within the mangled name of some other entity that needs promotion. Again the promotion-via-alias can be used for that entity, and the above mangling addition will ensure the symbol name is unique.

7.5 Discussion

The above description introduces a new mangling component <module-name>:

- Prefixed to <name>.
- Innermost-namespace scope objects partition the substitutions across different modules.
- Existing substitution scheme remains unchanged.
- Module-specific substitution scheme added.
- Module-name component of names in existing substitution table applies to final component.
- Internal-linkage promotion scheme uses empty sub-module name component.

8 Debug Encoding

DWARF has the concepts of namespace, class and function (`DW_TAG_namespace`, `DW_TAG_class_type` and `DW_TAG_subprogram` respectively), and it uses those to encode a heirarchy via placing children after their parents and using a sibling reference to list nodes at a single level.

DWARF also has a `DW_TAG_module` entry. I am not sure how well debuggers support it. Looking at GDB's sources all the occurrences to `DW_TAG_namespace` and `DW_TAG_module` occur in dwarf2read.c, and there are slightly more references (24) to the former than the latter (16).

How should modules interact with that encoding?

Discussion led us to conclude that encoding the module-name as an inline namespace was the best path forwards. Perhaps at a later date an additional namespace attribute can be added to indicate it is a module-name. We decided not to use the `DW_TAG_module` alternative.