Name Mangling for Modules

2017-03-07 - 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

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 – except 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.

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:

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

```
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 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.

6.1 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 squangling substitutions, as they are built on top of the compiler's internal hierarchy of object nodes.

The (in-development) G++ implementation represents a module's namespace partitions as within the namespace they are partitions of. Thus the module appears multiple times as-if an innermost namespace.

6.2 Module Name

It seems natural to extend the `<nested-name>` grammar to include the module name as a component, introduced via a special marker. However, there are some non-trivial issues with doing so, which will require thought.

It appears that 'M' is available as a prefix character.

The grammar would be:

```
<module-name> ::= M <unqualified-name>
```

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

Two obvious options for sub-modules are:

- a set of M prefixes, 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.

It seems best to mangle as a sequence of prefixes, as that avoid the need to find a different suitable separator character.

Altering the above mangling grammar as follows will allow this:

```
<module-subname> ::= M <unqualified-name>
<module-name> := [<module-name>] <module-subname>
```

Demanglers will need to use a '.' prefix on all-but-the-first M encoding of an object name.

Where implementations use a flattened representation of sub-modules internally, they could pre-mangle the [sub]module name so the mangler itself need not check for sub-modules.

6.3 <nested-name> component

If inserted as a `<nested-name>` component, it should appear just before or just after all namespace components. Which is best is affected by the hierarchy issue mentioned above. It is not really a human-readability issue, because mangled names are intended for machine consumption. As it makes my life easier, I shall place it just after the namespace components.

Unfortunately the existing <prefix> grammar is not sophisticated enough to place the module-name in exactly the correct position. We would need to split it to allow that. Placing the module-name first is simple, add:

```
<prefix> ::= <module-name> <prefix>
```

Placing it after the inner-most namespace requires something like the following surgery:

```
<namespace-seq> ::= [<namespace-seq>] <unqualified-name>

<class-seq> := [<class-seq>] <unqualified-name>

<container> := [<namespace-seq>] [<module-name>] [<class-seq>]

<prefix> := <prefix> <container>
         … other reductions, remove <unqualified-name> variants
```

6.3.1 Unscoped names

Both unscoped names, and the special encoding of the `::std` namespace with an otherwise unscoped name needs addressing:

```
<unscoped-name> ::= [<module-name>] <unqualified-name>
                ::= St [<module-name>] <unqualified-name>
```

or

```
<unscoped-name> ::= [<module-name>] <unqualified-name>
                ::= [<module-name>] St <unqualified-name>
```

Or perhaps partitions of `::` and `::std` with module-linkage should be mangled as a `<nested-name>` disregarding `::std`'s special encoding?

6.4 Orthogonal Mangling

Perhaps it is easier to treat the <module-name> as a separate component to a `<nested-name>?` That is, replace the `<name>` reductions with:

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

6.5 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.

If `<module-name>` is part of a `<nested-name>`, then the substitution scheme that naturally falls out of that depends both on how the module/namespace hierarchy is internally represented, and whether the `<module-name>` is placed consistently with that relative to a `<namespace-seq>`. This might be an unacceptable implementation-dependence.

If the `<module-name>` mangling is orthogonal to `<nested-name>` that difficulty is avoided, however there is still a question of whether a non-namespace object in one module's namespace partition can be substituted for an identically named object in other module's namespace partition. For instance, given:

```
class Foo::Baz@M1;
class Foo::Baz@M2;
void Quux<Foo::Baz@M1, Foo::Baz@M2> (…);
```

Can the second template argument's non-module name encoding back reference the first template argument's non-module name encoding? (I think such template instantiations will inevitably involve discovery via ADL as both class types must be non-exported names to require a `<module-name>` component in their mangling.)

Another choice is whether the separately prefixed parts of a sub-module mangling are independently substitutable or not. Consistency within the mangling scheme suggests that should occur, but internal representation suggests not.

If the `<module-name>` is not part of the regular back-reference scheme, should it use a separate substitution scheme? It would seem advantageous to do so, and a separate scheme would avoid answering the previous question concerning sub-module substitution within the current scheme.

As the `<module-name>` is a single contiguous block regardless of sub-moduleness, and has a unique prefix letter always followed by an `<unqualified-name>` mangling (which will start with a digit), we could indicate substitutions by adding:

```
<module-name> ::= M_ <seq-id>      # <seq-id> is a single character
              ::= MM <seq-id> _    # otherwise[2]
```

Each complete module-name contributes to the module-name substitution cache. Thus sub-module 'Foo.Bar' would not back-reference a previous 'Foo' module or other 'Foo.Whatever' sub-module.

6.6 Discussion

The above description delivers some choices that need to be made. At this time I do not have a good basis to make a decision, so will defer to a later time. The decisions that need to be made are:

- Is the `<module-name>` part of a `<nested-name>` (or an independent entity?)
- If it is part, where in the ordering should it appear?
- Should the `<module-name>` participate in the existing substitution scheme?
- How do the special case compression of `::std` and some STL items within `::std` interact with `<module-name>` substitution?

2 We could use a letter after the initial 'M' to indicate the length of the `<seq-id>`, but that seems like overkill.

7 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?

It may be acceptable to simply ignore the module information. Debuggers already have to deal with ambiguities across multiple source files, so perhaps that will be sufficient. At least to begin with.

If that proves unacceptable, investigation of current debugger's handling of `DW_TAG_module` should be done, to determine if that is a drop-in possibility.

Failing that, extending `DW_TAG_namespace` in some manner might be the best path forwards.

As with mangling, I shall defer a decision to a later point.