What do we talk about when we talk about Modules

Gaper Aman (gasper.azman@gmail.com)
Andrew Bennieston (a.j.bennieston@gmail.com)

June 25, 2017

1   Aim of this paper

The Modules TS [1] has generated much discussion. This paper aims to capture the questions and decisions that are necessary to arrive at a coherent proposal.

This paper takes no stance on any of the issues highlighted. Where possible, the differing opinions are cited and summarised in the discussion of each section. It is likely that there are issues and discussions of which the authors are not aware. We do not claim to represent every facet of the design space, and expect further iteration for refinement.

The goal of the authors of this paper is to disentangle the various issues people have with modules, so that constructive argument can happen along the issues and requirements. The issues are presented in as orthogonal a manner as possible, so that loops in arguing are avoidable.

2   What Are Modules For?

This section presents all the various answers to the titular question that the authors are aware of. It is probable that not all the answers are mutually compatible, realistic, or even on topic. However, unless stated and subsequently addressed, they will keep surfacing.

2.1   Reducing the need for detail namespaces

Name lookup is a flexible and powerful feature of C++, but the rules are complex and it is no secret that the set of names that are visible in user code is often larger than would be desirable. Customisation points further complicate matters.

To do this, private (and increasingly often) public features of libraries are implemented in a library-specific detail namespace, with externally visible names pulled into the public-facing main namespace with `using` directives.

This keeps implementations sane, but has the unfortunate side-effect of making compiler diagnostic messages longer than would be ideal.

One view is that it would be nice for modules to obviate the need to use detail namespaces to hide names by allowing name lookup outside the module to only find entities that are explicitly exported.

2.2   Strict ordering guarantees for code inclusion

Another issue that keeps library authors awake at night is the introduction of new names into their namespaces due to code like

```
#include "otherlibrary.hpp"
#include "mylibrary.hpp"
```

where `otherlibrary` defines some names (or macros) that clash with `mylibrary`.

An example of such a problem is the implementation of the standard library; all names in it need to use the `__name` and `_Name` conventions, despite already being in the `std` namespace – macros are not allowed to redefine names like that, since they are reserved. Regular libraries deal with this problem by assuming it won't happen.

If modules were specified to guarantee a strict ordering, such that *only the code that "happened"*
above this module is the code it specifically imports – applied *non*-transitively for name visibility
– then this issue disappears, and the standard library (along with every other library) can stop
worrying about macros rewriting its code.

2.3 Replace Precompiled Headers: the *Parse It Once* Argument

C++ code takes a long time to parse, and longer to compile. There are many reasons for this, but
having to read millions of lines of files to just get the declarations and definitions of all the visible,
invisible, and irrelevant names, for *every compilation unit*, is one of them.

If there was a way to parse a header once, expose a data structure that would be able to load
only the needed code at the time it is needed, and share that effort, the hope is that compilation
times would drop.

2.4 Replace Precompiled Headers: the *Highlander* Argument

Precompiled headers are also very inflexible, since *there can only be one*, and therefore encourage
having a pinch-point in the compilation graph that necessitates the recompilation of everything on
every minor change of any header.

The hope is that modules would act as mix-and-match precompiled headers, keeping compila-
tion quick while maintaining strict "as needed" import policies.

2.5 Better Namespaces: Module Namespaces

Some have the wish that modules would be better namespaces – modules would introduce names-
paces, completing the C++-is-a-better-python dream[1].

This goal is complicated by the necessity for code (which may be modularised) to define entities
in other namespaces. Standard library customisation points serve as a prime example; it is desirable
for a module to be able to define a specialisation of e.g. `std::hash`, which suggests that modules
and namespaces may need to be orthogonal.

Although the concepts may be orthogonal, many expect that there will be a strong correlation
between modules and (outer) namespaces. The following ideas have been proposed to help with
the common cases.

2.5.1 Introduce a namespace by default and offer an opt-out.

A module can introduce a namespace by default, and then offer ways of "escaping", for instance:

```cpp
// all names implicitly in the foo::bar namespace
module foo::bar;

struct mystruct; // foo::bar::mystruct;

::std::hash<mystruct> {
  std::size_t operator(mystruct const&)
      const;
};
```

This style of opt-out was specified in a paper by Tristan Bridle. [2]

2.5.2 Module-namespaces: Introduce a scope

A special syntax would enable introducing the module and its namespace together:

```cpp
module namespace foo::bar {
  // all names in the foo::bar namespace
  struct mystruct;
```

---

[1]The authors make no claim as to whether this is a dream worth having.

```
} // end module foo::bar

module namespace foo::baz {
  // look ma, two modules per file!
  struct mystruct;
}
// Entities defined here are only visible within the file and cannot be exported.
namespace std {
struct std::hash<foo::bar::mystruct>
{
   std::size_t operator(
        foo::bar::mystruct const&)
      const;
};
} // std
```

2.6 Better Namespaces: visibility qualifiers on names

Since modules seem to have the ambition of offering tight control of entities, they pose a question about what exactly they expose. The space of options follows. These are mostly mutually incompatible – macros are dealt with in a separate section.

2.6.1 Expose names

In this way, once you've exported a name, it's visible, along with all its overloads (for functions) or specializations (for types).

2.6.2 Same as class specifiers

Have access specifiers `public`, `private` and `protected` work for modules the same way as they work for classes: expose names for the purposes of overload set resolution, but break if the resolved-to entity is not accessible (e.g. the name is private and the call is outside the module).

2.6.3 Expose entities

Only expose entities, explicitly. From outside the module, the private entities are hidden from all name lookup (i.e. non-exported function names do not appear in the overload set).

2.7 Are macros exported?

This is quite possibly the biggest current disagreement. This section summarises the arguments and counter-arguments.

2.7.1 It breaks the compilation model

If macros are exported, then how can the import directive run after the preprocessor?

2.7.2 I want to guarantee my import doesn't rewrite my source code

It is a valid concern, but can be solved independently of macro exports by looking at strict ordering guarantees for inclusion. Presumably you can trust the libraries you import explicitly.

2.7.3 Macros do appear in public interfaces of libraries

From `MAX_SIZE` to `assert` to `BOOST_HANA_STRING`, macros are a part of the public interface of many libraries, and therefore modules should offer a way to expose them, goes the argument.

A possible solution (if your answer is still *no*) is to offer the macros in a separate header meant for the `include` mechanism that imports the non-macro part of the interface it depends on. This makes the way modules would work with the preprocessor far more obvious, but on the flip side

makes the answer to *how do I use this library* a lot *less* obvious. On the other hand, it means libraries that are `import`ed will *never* leak macros.

3   Acknowledgements

We would like to thank Vittorio Romeo for a few suggestions, and Jackie Kay and Louis Dionne for proofreading the paper.

References

[1] Gabriel Dos Reis. *Working Draft, Extensions to C++ for Modules (N4647)* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4647.pdf

[2] Tristan Brindle. *Allowing Class Template Specialisations in Unrelated Namespaces* https://github.com/tcbrindle/specialization_proposal/blob/master/P0665r0.pdf