What do we talk about when we talk about Modules

Gašper Ažman (gasper.azman@gmail.com)
Andrew Bennieston (a.j.bennieston@gmail.com)

June 24, 2017

1 Aim of this paper

This paper aims to be the definitive guide to arguing about Modules. To do that, it presents all the questions and decisions that need to be taken in order 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 summarized in the summary section for the question.

The hope of the authors of this paper is to untangle the various issues people have with modules, so that constructive argument can happen along the issues, and not around FUD. 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 highly 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 highly convoluted, flexible, and powerful feature of C++. It is no secret that keeping the set of visible names sane in user code while somehow allowing customization points keeps library authors awake at night.

To do this, private (and more and more 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 compiler error messages then include the detail namespace, making names longer than needed.

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 that clash with `mylibrary`.

An example of such a problem is the implementation of the standard library! This is one of the reasons 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 and going "la la la this is real-world code".

If modules specified a strict order of *the only code that "happened" above this module is the code it specifically imports - and that goes NON-transitively!*, this is no longer an issue, 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 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 minute change of any header.

The hope is modules would act as mix-and-match precompiled headers, keeping compilation 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 namespaces, completing the C++-is-a-better-python dream[1].

Unfortunately for this one, it is impossible - modules need to be able to provide overloads to `std::hash`, at least, which necessitates the orthogonality of modules and namespaces.

However, some expect that modules and namespaces, while orthogonal, will be substantially *correlated*. They propose the following ideas to help 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:

```
// 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;
};
```

2.5.2 Module-namespaces: Introduce a scope

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

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

module namespace foo::baz {
  // look ma, two modules per file!
  struct mystruct;
}
// this stuff only visible within the file
// 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, except for the whole macros story, which has a separate section completely.

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 (eg. private and the call is outside the module).

2.6.3 Expose entities

Only expose entities, explicitly. From outside the module, the private entities do not even appear in the overload set.

2.7 Are macros exported?

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

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

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 include 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 whole lot *less* obvious. Again, on the flip side, it means libraries that are `import`ed will *never* leak macros.