Modules: Keep the dot

D1948R0
Reply to: corentin.jabot@gmail.com
Audience: EWG, SG-2, SG-15

Abstract

A number of National Body Comments, and a paper (P1873) call for the removal of dots in modules for C++20.

The author thinks this would have a negative impact on the C++ ecosystem and encourage bad practices in such a way that it would make it significantly harder for any satisfying package management system to ever arise.

Module naming and uniqueness

Modules names identify modules and as such need to be unique within a program.

(Note: A module is a module interface unit, all its implementation units and its partitions, whose partition name is only relevant from within the module they are a partition of)

They also participate in name mangling, which make them part of both the API and ABI of a program (renaming a module entails modifying all translation units importing that module).

Uniqueness of name within the program implies uniqueness of name in the universe.

Most projects will include third party dependencies - which may have dependencies of their own, etc. Note that in this case “third party” would be any codebase not under one’s control, including the team down the hallway. It is also orthogonal to open source - people use proprietary middleware - etc.

Applied transitively, uniqueness of names applies to the universe - which is fundamentally what “ecosystem” implies.

Failing to conserve uniqueness of module names would lead to many libraries being source incompatible with each other and not usable within the same project, defeating the long term hope of dependency management (dependency management implying that dependencies can be had).

Uniqueness of name is a convention

Without a fairly large (and at this date unrealistic) tooling support, module name uniqueness can only be a convention. And while SG-15 so far has not been willing to specify one, such convention will hopefully emerge. As it did for headers names over the past few years (this is not a new problem and the solution are not new either).

Existing practices converged toward making the name of the project and often the name of the entity who created or maintain the project a part of the headers paths
Ex: `#include <boost/asio/ssl.hpp>`

Here the project is Boost.ASIO, under the boost umbrella, and only part of that project is included (presumably we don't need all of boost::asio's declaration in this particular file).

The path separator is used to separate the different components - and the hierarchy on disk reflects that.

Of course, modules are not identified by their paths - which I would argue is a big improvement over headers, so we need some kind of ways to identify and codify these different components.

The only non alpha-numeric at our disposal is _ But of course there is 80 years of existing practice using _ as a space in context in which a space cannot be used.

Consider `boost_asio_ip_address`. Is `address` organized below `ip_address` or is it its own logical and undivisible thing?

Introducing more tokens solves nothing

It was suggested that instead of removing the dot, one or more tokens are added to let users organize their code in a way that would address the concerns expressed above.

Let's pretend we allow another token, for example ⁑ - which has been chosen for its ludicrousness so that we don't get lost in specifics.

A module name is now `boost⁑asio⁑ip⁑address:iterator;`

The intent of the dot removal proponents is for the dot to be added back later.
So now our module looks like:
`boost⁑asio⁑ip.address:iterator;` Where each of [. : ⁑] has a different semantic meaning.

This starts too looks complicated and needlessly increases the cognitive overhead.

But more than that, **it will not be possible to rename modules without incurring an API** break.
In that scenario modules written after c++23 would have a different naming scheme than the one written before.

(Note: A single module is closed while namespaces are not - for that reason, while module names and namespaces should share a common root name it would not be a good idea to use the same component separator for modules and namespaces).

Big modules

Deprived of the ability of organizing their modules into both organizational and logical entities (Generally libraries namespaces are organizational entities, libraries should be logical units and modules names are both), an uninformed developer might be under the impression that modules should be big.

`import  boost`;
`import  qt`;

This poses a number of issues.
From a maintenance standpoint, everything now depends on everything - Maybe not, but who could tell what is used and what is not?
It could be worse though:
`export import  qt`;

Ô Hyrum.

It would be astute to notice that boost cannot be a module as it is a mutable collection of libraries rather than a library, but again, without a dot that is not codifiable.

More concerning still, `import  QtBase`; requires that the binary module interfaces that compose that module would be present at the point of use.

But the issue is mostly that, at link time, all the module **implementation** units defining entities which are ODR-used must have been compiled.

It would be really difficult for a tool to infer what implementation files are required to be provided, which leaves 2 implementation strategies

- Compile and link all implementation units of an imported module, transitively.
- Manually specify a “library” subdivision such that modules could span multiple libraries.

The later is the model that has been used for header/sources files but it present significant disadvantages:

- Cognitive overhead of having to know which library defines a given symbol

- Libraries tend for simplicity to be large organizational units, which present all the same issues as big modules
- Maintaining these build system has a high cost
- Precludes low or zero configuration build systems

In both cases, we know that the number of dependencies grow quadratically and the state of the ecosystem favors big libraries, which makes the cost of having dependencies very high.

Dependency management should follow the “don’t pay for what you don’t use” guidance such that code should

1. Depend on what it needs
2. Not depend on what it doesn’t need

The lack of dots in module names make promoting these practices very difficult.

I should also note that, for all the same reasons, `export import` is a much sharper tool that it appears - and export import -ed modules are part of the API - In general a module should probably export only the thing that module needs.

Submodules

For the reasons stated above, I do not think submodules **as a syntactically enforced language feature** are a particularly good idea short of mandating a file hierarchy.
There are other reasons:

- Some components can never be modules on their own: The existence of an `google.abseil.hash_map` module should not imply the existence of a `google` module.

- More importantly, the set of modules available on a given system is not closed. Hence if a module could implicitly import an arbitrary, mutable collection of submodules the code could be affected by changes in the environment in unpredictable ways.

You will notice that languages that have a notion of submodules baked in the language usually have a filesystem hierarchy matching the module hierarchy such that the set of submodule is bounded (and closed at a given point in time) and under the control of the author of some top level module.

Private modules:

The idea of private modules - where one module could only be imported from some other modules - is a much more convincing idea.

I however fail to see how precluding dots from appearing in module names would favor that use case.

It is easy to find ways to add private modules on top of the existing design:

```
export private module corentin.foo._bar //Modules starting with _
are private

friend module corentin.other_mod.*; // Could also have an explicit
syntax instead of assuming a hierarchy.

export module corentin.foo:private_access.with.dots // as
suggested by Richard Smith
```

I am more concerned with the fact that submodules seems to be a more general, simpler idea than partitions, and by having partitions now and submodules later, we would end up with 2 ways to represent the same ideas.

Recommendations

- I strongly advise the committee to keep the dots in module names in 20.
- The grammar of module names need to be better specified so that a module name is a single token
- I would recommend reserving a single **leading** underscore in a module name component for future use
- I would recommend not allowing modules and partition names names to be keyword or to contain keyword, especially friend, private, public and protected.
- Carefully consider whether partitions are necessary in 20 if there is a strong desire for a generalized submodule mechanism later.
- I encourage SG-15 to consider offering some guidelines pertaining to module naming, module re-exporting and module composition as to encourage a healthy and scalable ecosystem of module-based projects

Acknowledgments

Thanks to Rene Rivera and Colby Pike for reviewing and proof-reading this paper.

References

- P1873R1: [SG2, Evolution] remove.dots.in.module.names (Michael Spencer)
- P1634R0: [SG15] Naming guidelines for modules (Corentin Jabot)

- CppCon 2019: Corentin Jabot “Dependency Management at the End of the Rainbow
- CppCon 2018: Peter Bindels “Build Systems: a Simple Solution to a Complicated Problem”
- P0816R0: [Library Evolution] No More Nested Namespaces in Library Design (Titus Winters)
- P1876R1: [SG2] All The Module Names (Rene Rivera)