# What do we want from a modularized Standard Library?

| | |
|---|---|
| Document #: | P2172R0 |
| Date: | 2020-05-17 |
| Project: | Programming Language C++ |
| Audience: | LEWG |
| Reply-to: | Corentin Jabot <corentin.jabot@gmail.com> |

*Modules will solve everything!*

## Abstract

[P0592][**?**] makes modularizing the standard library a priority for C++23. The idea is that C++20 has modules and we should use them. This paper wonders what would be the benefits of such a time-consuming endeavor.

## Benefits and Constraints

### Compile times

Modules offer primarily 4 benefits: Improvement in compile times, Some protection against ODR violations, isolation of implementation details, and isolation from and of macros.

It is an attractive proposition for libraries that can be modularized. Unfortunately, some of these benefits may not be realized because of constraints specific to the standard library.

It is important to remember that C++20 standard headers are importable, which suffice to offers most of the compile times improvements. On my system, the time it takes to import ALL standard header (about 0.10 seconds) is a 10x decrease in time over including them.

### Isolation from macros

Isolation from macros would allow implementers to get rid of the _Ugly names. This can be supported with headers units as they are not affected by the preprocessor state, even when included, using a level of indirection

```
// vector_impl.hpp
int max = 0

// vector
import "vector_impl.hpp";

// user.cpp
#define max
```

```
#include <vector> // OK
```

Unfortunately, it is customary for implementers to support several C++ versions within the same standard library codebase. Implementers may decide not to do that de-uglification until they drop support for C++ versions without modules

### ODR And ABI

Modules are supposed to protect against a class of ODR issues by mangling the module name in the symbol name. That protection is weakened by the existence of the weak ownership model, but, most importantly doing so would be an ABI break. Until the standard decides to break ABI, existing symbols have to be own by the global module. That benefit is therefore not currently realizable given our current constraints.

ABI stability or assurance that programs are well-formed. Pick one?

### Symbol isolation

It might be possible for implementers to hide implementations details:

```
// vector_impl.hpp
namespace std_detail {
    int foo = 42;
}
export namespace std {
    class vector {};
}

// vector.cppm
export module;
import "vector_impl.hpp";
export module __std.vector;
export std::vector;

// vector
import __std.vector;

// user.cpp
#include <vector> // OK
int x = std_detail::foo ; // error
```

But again, that requires a significant refactor of existing implementations which might run go the desire of implements to support multiple C++ versions.

## Should modules be big or small?

The ideal size of modules depends on many factors that are not yet fully understood:

- The cost of loading/querying modules withing a single TU.

- The cost of compiling the module interfaces which is vaguely linear with the number of module interfaces.
- Bigger modules lead to fewer nodes in the dependency graph reducing parallelism and distribution opportunities.
- The cost of having more module dependencies, aka how many TU need to be recompiled when a module changes.

However, the standard library is in a unique position of being the foundation of everything else, such that rebuilding the standard module interface would only be done once per toolchain (compiling all the headers units takes about 20 seconds of CPU time on my system).

We also need to consider that the standard library has a lot of dependent parts (for example many classes have iostream operators) and as such, small modules may not make a lot of sense.

Of course, modules are not magical. C++ compile times are often driven more by overload resolution than by header parsing. Larger headers may result in larger overload sets.

This is why large modules alone are not a sufficient solution to reduce compile times. Solutions to reduce the size of overload sets and lookup are necessary, one of these solutions being to use hidden friends.

In any case, it would be counter-productive to provide modules that are smaller than the current header granularity.

## Should new features be introduced as modules?

It is tempting to introduce new feature solely as modules. However, because of ABI, new names should most likely not be own by modules as this would prevent WG21 to move declarations between modules or change the order of dependencies between names.

Hopefully, modules should allow LEWG to deprioritize concerns about costs of features in terms of importing more headers. On that note, It is unfortunate that WG21 did not mandate the translation of `#include` to `import` for importable standard headers.

## The teaching argument

People have argued that the library should be modularized to lead by example. However, we see that the standard library has many constraints other projects might not have

- `include` need to keep working
- We decided not to break ABI
- Implementers support multiple language versions

These constraints prevent us to ”just use modules”.

Thee is also an added burden for users if there are several ways to import symbols, especially if modules are small: People will have to remember that pair is in `<utility>` regardless.

## Conclusion

Most of the benefits of modules can be realized while sticking with `#include` and `import <>` as the only way to import standard names. Other benefits can never be realized because of ABI stability. Larger compile times can be gained by focusing on constraining lookup.

It is worth asking ourselves whether modularizing the standard library is the best use of our time.

### Recommendations

- Prioritize work to re-specify existing overloaded operators and customization points in term of hidden friends over a modularization of the standard library.
- Assuming work to reduce the cost of large overload sets, prefer large modules.
- While freestanding and modularization are orthogonal, freestanding can inform how to best organize standard modules. Features that are at least partially freestanding should be in the same module.
- Do not try to replace all headers (`<cassert>`, `<version>`). Features that depend on the conditional compilations should be explicit in their use of the preprocessor. Macros and functions macros that always have the same definition should be replaced.
- Assume standard headers are imported for the purpose of determine the cost of proposals in term of compile times.

## References

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4861