Doc. no. P2412r0
Date: 2021-07-12
Project: Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (bs@ms.com)

Minimal module support for the standard library

Bjarne Stroustrup

Abstract

Header files are a major source of complexity, errors caused by dependencies, and slow compilation. Modules address all three problems, but are currently hard to use because the standard library is not offered in a module form. This note presents logical arguments and a few measurements that demonstrates that **import std** of a module **std** presenting all of the standard library can compile many times faster than plain old **#include <iostream>**. Compared to using traditional headers, you can gain significantly in both logical benefits and compile-time performance from using modules.

Today's multitude of headers, their potential for surprises, and slow compilation are barriers to entry to novices and casual users. That's a serious problem because this kind of users are the future of C++. They are the majority of users and our future experts. Standard-library modules can help and I conjecture that the benefits in code hygiene and fast compiles of modules also scale. The standard library should provide a model for the definition and use of modules.

1. We need standard-library modules

C++20 offers modules, but not a standard-module version of the standard library. This seriously hinders the adoption of modules

- A simple program using modules for standard-library features cannot be ported without defining those modules
- Educators can't just use modules as easily as the standard headers
- There is a shortage of realistic examples of good use of modules
- Different sets of "semi standards" will appear for popular platforms

The standard library is supposed to be an example of good style. However, the need to **#include** or **import** a host of standard-library headers is a serious burden for many and an example of how not to present facilities. One problem that can be addressed by modules is the need for users of all experience levels to know which headers need to be included to get a job done. Separate modules obviously make sense for logically unrelated components, but the current mess of 98 standard headers reflecting 50 years of history is an embarrassment for the C++ community.

If experience is to go by, we (WG21) could spend years discussing details of how best to "modularize" the standard library. Most of that discussion will focus on optimal support for specialized groups of advanced users. Key ideas behind this note are

- Postpone those discussions
- Adopt essential modules for "the masses" for C++23 ASAP

I suggest we urgently proceed in the direction outline in P0581R1 "Standard Library Modules." To summarize:

Design principles:

- Leave "C headers" alone
- Present modules as logical sets of functionalities
- Robust support for C++ in diverse environments
- Every standard facility is provided by exactly one module

Suggested modules:

- **std.fundamental**
- **std.core**
- **std.io**
- **std.os**
- **std.concurrency**
- **std.math**
- **std**

The suggested contents of the modules can be found in P0581R1 (which must be updated for C++20). Here, I will focus on module **std** – the module that offers almost all of the standard library – so that we can write:

```
import std;

int main()
{
        std::cout << "Hello, World\n";
}
```

From this classical start, learners, novices, and experts can proceed for quite a while without having to look up where a standard-library facility is defined.

I assume that many would like some fine-grain modules, e.g., to ensure that standard-library facilities not suitable in a context aren't deliberately or accidentally imported. For example:

- **import std.STL;** // containers, algorithms, iterators, ranges, concepts (only)
- **import language_support;** // minimal support for the complete C++ language
- **import educational;** // facilities for teaching programming using C++

I am certain that we could spend years discussing such fine-granularity, restrictive, alternative, and additional modules: Should they exist? Isn't the finest granularity better and necessary? Where do we put **experimental::** features?

Let's not wait for these potentially infinite discussions aiming at perfection. Instead:

1. Get module **std** into C++23
2. Get as many of the other modules from P0581R1 as we can agree on into C++23
3. Invite proposals for more specific modules and accept ones that has broad consensus
4. Keep working on refinements.

My impression is that there are no technical problems left for having a component presented as part of several modules (yet defined in one module only), so by adopting module **std** we do not paint ourselves into a corner vis. a vis. fine-grained modules (see §6).

2. Standard-library components exported from module std

Standard headers, being headers, contain many implementation details that should not be **export**ed. So, **module std** cannot just be an aggregation of headers **export**ing their every declaration. Instead, only declarations of names mentioned in the standard should be **export**ed from **module std**.

Ideally, no **import** should pollute the global namespace. Doing so impedes composition of code from libraries. That's just like for **#includes**. So, **import std** should not pollute the global namespace. Unfortunately, key standard-library features are directly **#includ**ed into the global namespace and it would be very hard to avoid repeating that pre-C++98 mistake in time for C++23. However desirable, such major cleanup will have to wait.

Module **std** cannot and should not **export** macros.

Sadly, **errno** is a macro. Let people **#include<cerrno>** when needing to check for errors in simple math. Unfortunately, every serious use of **<cmath>** will have to do that (unless we finally find an alternative).

Sadly, **assert()** is a macro and we couldn't so far agree on a non-macro contracts replacement for it. For now, if someone wants to use **assert()**, let them **#include<casset>**.

```
#include<cerrno>
#include<cassert>
import std;

void f(double d1, double d2)
{
        assert(0<=d1);
        double y = sqrt(d2); // OK (unfortunately; I would have preferred to require std::sqrt(2))
        If (errno)
                std::cerr << "negative argument to sqrt()\n";

}
```

Fortunately, **errno** and **assert()** are not packaged together with lots of logically independent facilities.

3. Header units

Why don't we just recommend everybody to use header units?

- Header units do not address the problem of complexity of including the right headers. Many people will still get confused and frustrated about which headers to **import**. Others will **import** every header they think they might need (probably using header guards in case someone **#include**s), thus adding a significant compilation burden (compared to modules).
- Header units contain transitive **#include**s, leaks macros, and suffer the significant compile-time overheads implied by that.
- Headers carry almost 50 years of history, implying many unfortunate header dependencies, so header units don't even address the fundamental needs of logically-defined fine-grained modules.
- Header units still leak macros. This can be useful for keeping old code working, but leaves code vulnerable to unexpected macro substitutions and to changes to the set of macros used in the implementation of the header unit.

Header units can be used as a transition mechanism, but implies a huge risk of users getting stuck with backwards-looking preservation of decades old mistakes.

4. Compile-time performance

Traditionally, we have tried to minimize the time needed to compile a program using headers by minimizing the number and sizes of the headers **#include**d. This has led to much complexity in code and build systems, but fundamentally it has been a losing battle against complexity. Header files have grown from a minor nuisance and a significant convenience in early C to become humongous morass of large files often constituting complex nested dependencies. Cyclic dependencies are handled (or not) by header guards and (non-standard) **#pragma once**.

Large modules – representing significant logically-delimitated facilities – address this complexity for users directly. That's essential for addressing the needs of novices, casual users, and experimenters.

I measure the compile times for two simple programs accessing the standard-library in various ways:

(1) *Traditional use:* **#include** needed headers
(2) *Minimal header unit use:* **import** needed header units
(3) *Simplest use*: **import std;** import a unit containing the whole standard library
(4) *Simplest traditional use*: **#include "all_std.h"**
(5) *Simplest header unit use*: **import "all_std.h";**

where **all_std.h** includes all standard-library headers.

The examples are

(1) "Hello world" needing just **<iostream>**.
(2) "Mix" using a mix of facilities requiring the **#include** of nine popular standard headers (see the appendix).

Both examples illustrate simple and naïve uses of C++, of which there are an immense number. Both would be naively assumed to favor small headers and small modules for compile times.

Compile times vary dependent on compiler, compiler flags, build systems, operating systems, hardware, and more. Unfortunately, there no standard for setting up module creation, so it is non-trivial to time a variety of programs, but since my aim is simply to demonstrate the affordability of large modules the simplest approach a proof of concept is sufficient:

	#include needed headers	**Import** needed headers	**import std**	**#include** all headers	**Import** all headers
"Hello world" (<iostream>)	0.87s	0.32s	0.08s	3.43s	0.62s
"Mix" (9 headers)	2.20s	0.77s	0.44s	3.53s	0.99s

No, that 0.08s is not a typo; it really is 10 times faster to import 10 times more standard-library code.

These are figures from my laptop, a modest Intel i-7 running windows, not so different from what a student or a casual programmer might use. I used the Microsoft compiler. Each run was done 4 times and averaged. There were no dramatic outliers.

The module and the header units were separately pre-compiled into intermediate form before use.

Module **std** is my best approximation of what the complete standard library stripped of implementation details, transitive **#include**s, and macros would be, generated using tools from the Modules TM era.

Generalizing a bit from this limited data and rounding the figures to simplify, I conclude that

- if you can afford **#include <iostream>**, you can afford **import std;**.
- The "Real module" significantly outperforms the header units by a factor of two or more.
- These may be worst case figures for use of **#include**/**import**. For larger programs non-library code often matters more, but conversely, there are large programs where individual translation units consists mainly of massive libraries (included dozens or hundreds of times across a complete build).
- Once the module or the header units are #**include**d or **import**ed, compilation of the user code takes about the same time (the **#include "all_std.h"** numbers are weak on this point, but here the user-code is a very small fraction of the work).
- For "the fair comparisons" (**import**ing and **#include**ing all of the standard-library), **import** beats **#include** between 7 to 60 times. It would not be unreasonable to expect improvements in the 5 to 25 range for realistic examples. See also [DE].

These numbers a from just one compiler, one computer, and just two trivial (and therefore comprehensible) code examples. I hope people will add to the data with examples of other styles of uses, performance on other systems, and using other compilers.

I was asked to point out that the numbers I present are from a yet-unoptimized implementation. I hope and expect to see even better performance in the future. Knowing the work on program representation by Gaby Dos Reis and me [IPR], I am not very surprised by these spectacular improvements. I also hope and expect to see similar spectacular results from the other module implementations.

5. A view of the future

The dramatic performance advantages to "large modules" should lead us to re-evaluate our views of how best to present library facilities. Traditionally, we have minimized the interfaces provided by the interface to a library unit. I think that much larger units representing a user's view of a library, rather than an implementer's, will lead to significantly simpler code. I expect to see much application code structured like this :

```
import std;
import my_DB;
import my_GUI;
import my_application_foundation;
// … my code …
```

Or for established projects even just:

```
import my_project_base;
// … my code …
```

Composing modules out of other modules will become an essential design skill.

6. Ownership

If a library component is presented in two modules, say **ostream** in **std** and **std.io**, which module owns it? Ideally, one module is "bigger" than the other and "higher level", just as is the case for **std** and **std.io** here. That way, the higher-level module can simply aggregate (import and re-export) the features from the lower-level module:

```
export module std;
export import std.io;     // ostream owned by std.io
// …
```

Such a hierarchical organization is ideal for comprehension and maintenance. It is easily achieved when (as here) the lower-level module exists when the higher-level module is defined. P0581R1 provides a good start on such a set of lower-level modules.

Note that this has nothing to do with module partitions or "sub-modules." My use of the dot is simply a helpful naming convention.

What happens if we – long after we have shipped **std** – decide to produce a module, e.g., **std.STL2**, that offers a different subset of **std**? We simply **import** and re-**export** the desired parts in the new module.

References

- [P0581] Marshall Clow, Beman Dawes, Gabriel Dos Reis, Stephan T. Lavavej, Billy O'Neal, Bjarne Stroustrup, and Jonathan Wakely: *Standard Library Modules*. P0581R1 2018-02-11.
- [P1453] Bryce Adelstein Lelbach: *Modularizing the Standard Library is a Reorganization Opportunity*. P1453R0 2019-01-21. Pointing out urgency.
- [IPR] Gabriel Dos Reis and Bjarne Stroustrup: A Principled, Complete, and Efficient Representation of C++. Journal of Mathematics in Computer Science Volume 5, Issue 3 (2011).
- [Git] IPR on GitHub .
- [DE] Daniela Engert Modules the beginner's guide Meeting C++ 2019

Acknowledgements

Thanks to Gabriel Dos Reis, Cameron DaCamara, and Stephan Lavavej for making sure that I used their modules implementation according to the letter and spirit of the standard.

Suggested starting point for wording

In 16.5.1 add a section

> *16.15.1.4 Standard-library modules*
>
> > *Module std – exports all non-macro names specified in namespace std.*

The modules suggested in [P0581R1] make a good start for going beyond this minimal suggestion.

Appendix

Here is the simple "Mix" program I used to exercise a few standard-library components. It obviously doesn't do anything interesting. The important point is that I poke into each header. For the experiments, I replaced the nine **#include**s with equivalent **imports**, or **import std;**

```
#include <iostream>
#include  <map>
#include  <vector>
#include  <algorithm>
#include  <chrono>
#include  <random>
#include  <memory>
#include  <cmath>
#include  <thread>

using namespace std;

void test1()
{
        using namespace chrono;
        using namespace literals;
        auto t0 = system_clock::now();
```

```
        double s2 = std::sqrt(2);
        auto t1 = system_clock::now();
        cout << "sqrt2: " << s2 << " " << (t1 - t0) << "\n";
}

void test2()
{
        vector v = { 1,2,3 };
        map<string, int> m = { {"foo",1}, {"bar",2 } };
        for (auto& [name, val] : m) v.push_back(val);
        for (int x : v) cout << x << " ";
        cout << '\n';
}

void test3()
{
        auto up = make_unique<int>(7);
        auto sp = make_shared<int>(8);
        *up = *sp;
        *sp = 9;
        cout << "up: " << *up << " sp: " << *sp << '\n';
}

void test4(int max)
{
        default_random_engine eng;
        uniform_int_distribution<int> dist(1, max);
        cout << "x: " << dist(eng) << " " << dist(eng) << " " << dist(eng) << '\n';
}

void test5()
{
        jthread t1{ test1 };
        jthread t2{ test2 };
}

int main()
{
        std::cout << "Hello, world\n";
        test1();
        test2();
        test3();
        test4(17);
        test5();
}
```