Document Number:	P2465R3	
Date:	2022-03-11	
Reply To:	Stephan T. Lavavej	<stl@microsoft.com>
	Gabriel Dos Reis	<gdr@microsoft.com>
	Bjarne Stroustrup	<bs@ms.com>
	Jonathan Wakely	<cxx@kayari.org>
Audience:	Library	(LWG)
	Core	(CWG)

Standard Library Modules `std` and `std.compat`

Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup, Jonathan Wakely

Abstract

This paper provides Standardese for two named modules: `std` and `std.compat`.

`import std;` imports everything in namespace `std` from C++ headers (e.g. `std::sort` from `<algorithm>`) and C wrapper headers (e.g. `std::fopen` from `<cstdio>`). It also imports `::operator new` etc. from `<new>`.

`import std.compat;` imports all of the above, plus the global namespace counterparts for the C wrapper headers (e.g. `::fopen`).

Changelog

- R0
    - Reviewed by LEWG on 2021-10-12.
- R1
    - Renamed `std.all` to `std.compat`.
    - Removed lists of alternative names.
    - Removed Library Evolution (LEWG) from the target Audience.
    - Added lines mentioning that alternative designs did not have consensus in LEWG.
    - Updated the Proposed Wording context to depict stable names like [class.spaceship] instead of section numbers like 11.10.3.
- R2
    - Updated after CWG review on 2022-01-27.
    - Added changes to [basic.stc.dynamic.general].
    - Changed "is reachable from" to "precedes", which is a Word of Power in [basic.lookup.general].
    - Removed the Open Wording Questions.
- R3
    - Updated after CWG review on 2022-03-11.
    - Changed [dcl.init.list] back to "is reachable from", adding a citation of [module.reach].
    - Added changes to [dcl.type.auto.deduct].
    - Rebased on Working Draft N4901 (paragraph numbers changed).

Background

See P2412R0 "Minimal module support for the standard library" which explains why `import std;` is ideal for usability and has amazing compiler throughput compared to classic `#include` directives.

Rationale

The biggest concern from the LEWG review of P2412R0 involved the global namespace and can be viewed as Adoption versus Cleanliness.

The Adoption concern is that if Standard Library Modules don't provide `::fopen`, `::size_t`, `::tm`, etc. then it will be difficult to migrate existing codebases - especially because on some platforms, the True Names are always global, so even if users are disciplined about including `<cmeow>` instead of `<meow.h>`, it is extremely easy to depend on the global namespace.

The Cleanliness concern is that the global/`std` namespace issue (where `<meow.h>` definitely provides `::meow` and might provide `std::meow`, and `<cmeow>` definitely provides `std::meow` and might provide `::meow`) is a complexity headache that's annoying to teach and to deal with, and Standard Library Modules are probably the only chance we'll ever have to permanently resolve it.

We believe that providing two named modules will follow LEWG guidance and address both concerns by giving users a choice.

- Users who want Cleanliness can say `import std;` and avoid polluting the global namespace. There's no extra typing, and no throughput impact.
- Users who want to prioritize Adoption in an existing codebase can say `import std.compat;` and avoid having to qualify all mentions of `fopen`, `size_t`, `strlen`, etc.
    - This is a bit more typing, but it's still just 1 line.
    - There's strictly more throughput impact, but it should be minimal (as we've found, importing a named module is extremely fast,

and the `<cmeow>` headers define much less machinery, both in quantity and complexity, than the C++23 Standard Library).

In general, modules are orthogonal to namespaces, but this reflects the existing division between `<cmeow>` and `<meow.h>`. It should be natural for users to learn.

The Sufficient Additional Overloads ([cmath.syn]/2) are always a headache to analyze, but we believe that this avoids disrupting them in any way. Someone who says `import std;` gets all of the Overloads in namespace `std` (and no danger of referring to the global name, an improvement). Someone who says `import std.compat;` gets the same behavior as someone who includes `<cmath>`, `<cstdlib>`, `<math.h>`, and `<stdlib.h>` (or just `<cmath>` and `<cstdlib>` on a platform like MSVC).

Note that [module.unit]/1 has reserved `std` and `std.*` as module names.

Alternative Design: Separate Module for the Global Namespace

Another alternative would involve changing the design, where `import std;` would behave as specified here, and `import cstd;` (or some other name) would provide just the global namespace names. (With `<new>` remaining in `import std;`.) This would require users who want both to say both `import` declarations. This alternative design did not have consensus in LEWG.

Alternative Design: Reverse the Modules

Yet another alternative would be to have `import std;` provide namespace `std` and the global namespace, while `import std.strict;` (or some other name) provides only namespace `std`. This alternative design did not have consensus in LEWG.

Frequently Asked Questions

Are any macros provided by the Standard Library modules?
No, and this is intentional.

For feature-test macros, users must `#include <version>` or `import <version>;` (as header units can emit macros).

For `assert`, `errno`, `offsetof`, `va_arg`, etc., users must `#include <meow.h>` or `#include <cmeow>`. (`<assert.h>`/`<cassert>` are extra-special as they can be repeatedly included with varying behavior controlled by `NDEBUG`.)

Are deprecated features provided by the Standard Library modules?
Yes. This is implied by the normative wording.

Are C++ features in C wrapper headers, like `std::byte` in `<cstddef>`, provided by the Standard Library modules?
Yes. `import std;` provides everything in namespace `std`, including the contents of the C wrapper headers, including C++-specific additions.

Are `std::initializer_list` and other machinery with Core Language interactions provided by the Standard Library modules?
Yes. This is implied by the normative wording. However, we do need to update the Core Language wording that requires `<compare>`, `<initializer_list>`, and `<typeinfo>` to be included (classically) or imported (as header units), now that named modules are available.

Is there any implementation experience?
Not yet.

The experimental named modules that MSVC has been shipping for a while are a repackaging of the existing library (with no distinction between `std::sort` and `std::_Internal_helper_functions`) and divide it into subsets (instead of the "only `std`" and "`std` plus global" design here).

Should new proposals target *only* the Standard Library modules?
No. New proposals should expect that many users will continue to use classic includes for many years for various reasons, such as build system issues. (We expect that there will be a few users of C++20 Standard Library header units, although that early adopter population is likely to migrate to C++23 Standard Library modules as soon as possible.)

Therefore, new proposals should continue to carefully consider what headers to modify, and/or what headers to add. The main consideration for new proposals is that if they want to emit macros (other than feature-test macros) for any reason, that would be unavailable to users of Standard Library modules, so non-macro mechanisms should be found if possible.

Proposed Wording

All changes are relative to N4901.

- Add a new feature-test macro `__cpp_lib_modules` to [version.syn], with a value chosen by the editors as usual.
    - Unlike all of the other library feature-test macros, this should **not** have a comment "`// also in <HEADER>`".

- Add a new section that's a child of 16.4.2 "Library contents and organization" [organization], immediately after 16.4.2.3 "Headers" [headers]:

Modules [std.modules]

The C++ standard library provides the following *C++ library modules*.

The named module `std` exports declarations in namespace `std` that are provided by the importable C++ library headers ([tab:headers.cpp] or the subset provided by a freestanding implementation) and the C++ headers for C library facilities ([tab:headers.cpp.c]). It additionally exports declarations in the global namespace for the storage allocation and deallocation functions that are provided by `<new>` ([new.delete]).

The named module `std.compat` exports the same declarations as the named module `std`, and additionally exports declarations in the global namespace corresponding to the declarations in namespace `std` that are provided by the C++ headers for C library facilities ([tab:headers.cpp.c]).

It is unspecified to which module a declaration in the standard library is attached. [*Note:* Implementations are required to ensure that mixing `#include` and `import` does not result in conflicting attachments ([basic.link]). - *end note*]

Recommended practice: Implementations should ensure such attachments do not preclude further evolution or decomposition of the standard library modules.

A declaration in the standard library denotes the same entity regardless of whether it was made reachable through including a header, importing a header unit, or importing a C++ library module.

Recommended practice: Implementations should avoid exporting any other declarations from the C++ library modules.

[*Note:* Like all named modules, the C++ library modules do not make macros visible ([module.import]), such as `assert` ([cassert.syn]), `errno` ([cerrno.syn]), `offsetof` ([cstddef.syn]), and `va_arg` ([cstdarg.syn]). - *end note*]

- Modify 6.7.5.5.1 "General" [basic.stc.dynamic.general] as depicted:

"[*Note 2:* The implicit declarations do not introduce the names `std`, `std::size_t`, `std::align_val_t`, or any other names that the library uses to declare these names. Thus, a *new-expression*, *delete-expression*, or function call that refers to one of these functions without importing or including the header `<new>` ([new.syn]) **<ins>**or importing a C++ library module ([std.modules])**</ins>** is well-formed. However, referring to `std` or `std::size_t` or `std::align_val_t` is ill-formed unless ~~**<del>**the name has been declared by importing or including the appropriate header**</del>**~~**<ins>**a standard library declaration ([cstddef.syn], [new.syn], [std.modules]) of that name precedes ([basic.lookup.general]) the use of that name**</ins>**. — end note]"

- Modify 7.6.1.8 "Type identification" [expr.typeid]/7 as depicted:

"**<del>**~~If the header `<typeinfo>` ([type.info]) is not imported or included prior to a use of `typeid`~~**</del>****<ins>**The type `std::type_info` ([type.info]) is not predefined; if a standard library declaration ([typeinfo.syn], [std.modules]) of `std::type_info` does not precede ([basic.lookup.general]) a `typeid` expression**</ins>**, the program is ill-formed."

- Modify 7.6.8 "Three-way comparison operator" [expr.spaceship]/8 as depicted:

"The three comparison category types ([cmp.categories]) (the types `std::strong_ordering`, `std::weak_ordering`, and `std::partial_ordering`) are not predefined; **<del>**~~if the header `<compare>` ([compare.syn]) is not imported or included prior to~~**</del>****<ins>**if a standard library declaration ([compare.syn], [std.modules]) of such a class type does not precede ([basic.lookup.general])**</ins>** a use of **<del>**~~such a class~~**</del>****<ins>**that**</ins>** type – even an implicit use in which the type is not named (e.g., via the `auto` specifier ([dcl.spec.auto]) in a defaulted three-way comparison ([class.spaceship]) or use of the built-in operator) – the program is ill-formed."

- Modify 9.2.9.6.2 "Placeholder type deduction" [dcl.type.auto.deduct]/3 as depicted:

"If the *placeholder-type-specifier* is of the form *type-constraint*$_{opt}$ `auto`, the deduced type `T'` replacing `T` is determined using the rules for template argument deduction. **<ins>**If the initialization is copy-list-initialization, a declaration of `std::initializer_list` shall precede ([basic.lookup.general]) the *placeholder-type-specifier*.**</ins>** Obtain `P` from `T` by replacing the occurrences of *type-constraint*$_{opt}$ `auto` either with a new invented type template parameter `U` or, if the initialization is copy-list-initialization, with `std::initializer_list<U>`."

- Modify 9.4.5 "List-initialization" [dcl.init.list]/2 as depicted:

"The template `std::initializer_list` is not predefined; ~~**<del>**if the header `<initializer_list>` is not imported or included prior to**</del>**~~**<ins>**if a standard library declaration ([initializer.list.syn], [std.modules]) of `std::initializer_list` is not reachable from ([module.reach])**</ins>** a use of `std::initializer_list` — even an implicit use in which the type is not named ([dcl.spec.auto]) — the program is ill-formed."

Acknowledgements

Thanks to Cameron DaCamara, Jens Maurer, and Richard Smith for reviewing this wording and providing suggestions.