Standard Library Modules

Marshall Clow Beman Dawes Gabriel Dos Reis
Stephan T. Lavavej Billy O'Neal Bjarne Stroustrup
Jonathan Wakely

Abstract

With recent progress made on the 'module' language feature, it is long overdue that we start a conversation around uses of modules in the standard library. More specifically, how should the standard library be logically presented as set of modules to C++ programmers? This is a proposal to answer that question.

Design Principles

The presentation of the standard library as set of modules follows a few 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

Leave "C headers" alone

We strongly recommend leaving "C headers" alone because only confusion and chaos have ever resulted from past attempts at legislating what C standard headers should contain. The reality is that in practice, the contents of those headers are outside the reach of most C++ implementers. One could wish that wasn't the case, but this is the world we live in.

Present modules as logical sets of functionalities

In the past, we've structured standard headers to minimize their contents in the hope of aiding compiler throughput. The module system is designed to attain the ideal of zero abstraction overhead. That is, a module does not impose any measurable overhead if none of the names it provides is referenced in a translation unit. Consequently, we propose to structure standard library modules around logical set of functionalities, shifting the focus to uses as opposed to implementations.

Robust support for C++ in diverse environments

The decomposition of the standard library into set of logical modules should not preclude uses of C++ in more diverse environments. Rather, the logical presentation should enable more uses – not less – in environments such as embedded systems, operating system kernel modes, etc. The traditional dichotomy of "freestanding" vs. "hosted" implementations is not a good guide. For example, `std::unique_ptr` can be used in embedded systems; similarly the vast majority of the standard algorithms can be used in kernel mode. Let's not forget that while the module core language constructs affect declarations at the source program level, they also imply a certain decomposition and layering of runtime support.

Every standard facility is provided by exactly one module

As noted earlier, the existing header files structure is not guided by any discernable logical principle. That leads to the unfortunate situation where a given standard facility may be provided by more than one standard header. For example, which header "owns" `size_t`, or `abs`? The architecture presented here aims to avoid that confusion. Ultimately, there is only one module that owns an entity or is responsible to providing a functionality. Other, larger, modules may re-export a module that provides a certain functionality. That overall architecture is therefore hierarchical.

Proposed Structures

Naming

All standard library modules should be named according to the pattern '`std`.*xyz*'. However, we do not recommend a reflexive one-to-one mapping of headers <*xzy*> to module '`std`.*xzy*'. At this stage of the discussion, we suggest to focus primarily on the logical decomposition of the facilities rather than the names of the modules themselves, however important they are.

Modules

As a first approximation, we suggest the following: `std.fundamental`, `std.core`, `std.math`, `std.io`, `std.concurrency`, `std.os`, and `std`.

1. Module `std.fundamental` provides the declarations of the following facilities:
   - Content of `<cstddef>`
   - Content of `<limits>`
   - Content of `<cfloat>`
   - Content of `<cstdint>`
   - Content of `<new>`
   - Content of `<typeinfo>`
   - Content of `<exception>`
   - Content of `<initializer_list>`
   - Content of `<csignal>`
   - Content of `<csetjump>`
   - Content of `<cstdlib>`
   - Content of `<stdexcept>`
   - Content of `<system_error>`
   - Content of `<utility>`
   - Content of `<tuple>`
   - Content of `<optional>`
   - Content of `<variant>`
   - Content of `<any>`
   - Content of `<bitset> without the IO formatting declarations`
   - Content of `<type_traits>`
   - Content of `<ratio>`
   - Content of `<chrono>`
   - Content of `<ctime>`

- Content of `<atomic>`
- Content of `<typeindex>`
- `std::unique_ptr` and associated classes and functions from `<memory>`
- `std::shared_ptr` and associated classes and function from `<memory>`

2. Module `std.core` provides the declarations of the following facilities:
    - Re-export of module `std.fundamental`
    - Content of `<array>`
    - Content of `<list>`
    - Content of `<forward_list>`
    - Content of `<vector>`
    - Content of `<deque>`
    - Content of `<queue>`
    - Content of `<stack>`
    - Content of `<map>`
    - Content of `<set>`
    - Content of `<unordered_map>`
    - Content of `<unordered_set>`
    - Content of `<iterator>`
    - Content of `<algorithm>`
    - Content of `<execution>`
    - Content of `<functional>`
    - Content of `<string_view>`
    - Content of `<string>` without IO declarations
    - Content of `<memory>` except `std::unique_ptr` and associated classes and functions, `std::shared_ptr` and associated classes and functions
    - Content of `<memory_resource>`
    - Content of `<scoped_allocator>`
    - Content of `<regex>`
3. Module `std.io` provides the declarations from the following headers:
    - `<cctype>`
    - `<cwctype>`
    - `<cwchar>`
    - `<cstdlib>`
    - `<cuchar>`
    - `<locale>`
    - `<codecvt>`
    - `<clocale>`
    - `<iosfwd>`
    - `<iostream>`
    - `<ios>`
    - `<streambuf>`
    - `<istream>`
    - `<ostream>`

   - `<iomanip>`
   - `<sstream>`
   - `<fstream>`
   - `<cstdio>`
   - Declarations for IO formatting from `<complex>`
   - Declarations for IO formatting from `<string>`
   - Declarations for IO formatting from `<bitset>`
4. Module `std.os` provides the declarations from the following header:
   - `<filesystem>`
5. Module `std.concurrency` provides the declarations for the following facilities:
   - Content of `<mutex>`
   - Content of `<thread>`
   - Content of `<condition_variable>`
   - Content of `<shared_mutex>`
   - Content of `<future>`
6. Module `std.math` provides the declarations from the following headers:
   - Content of `<complex>` without the IO formatting declarations
   - `<numeric>`
   - `<valarray>`
   - `<random>`
   - `<cmath>`
7. Module std provides all the standard facilities. It can be thought of as an aggregation of all preceding modules.

Clearly, this list is incomplete, but it is suggested as a starting point of the conversation. We are not aiming for a magical number, e.g. 5. Rather, we are aiming for a set of principles to guide presentation of existing standard library facilities as logically coherent sets that also vigorously support uses of C++ in diverse environments. The number of standard library module isn't expected to remain fixed in its evolution; consequently the principles will play key roles in the evolution of the standard library structure. For example, care is being exercised to facilitate uses of 'std.core' in diverse environments, including embedded systems and those environments where default global allocation functions 'new' aren't appropriate. Similarly, the IO formatting parts are separated from core functionalities such as `complex`, `bitset`, and `string`.

Freestanding implementations are required to provide at least the module `std.fundamental`.

Concepts

If we had standard library concepts, as in the Range TS, they would go into `std.core` for the most part.

Acknowledgment

Thanks to Gor Nishanov for his feedback on early draft of the R1 revision.

Changes from R0

- Expansion of the general principles behind the overall architecture
- New section on formal specification

- State the place of standard concepts
- Rename `std.threading` to `std.concurrency`
- Removal of `std.memory`
- Rename `std.filesystem` to `std.os`
- Removal of `std.regex`, contents folded into `std.core`
- Rename `std.numeric` to `std.math`.
- New module `std`.