Freestanding Proposal

Document number: D0829R1
Date: 2017-10-14
Reply-to: Ben Craig <ben dot craig at gmail dot com>
Audience: SG14, Library Evolution Working Group

I. Introduction

The current definition of the freestanding implementation is not very useful. Here is the current high level definition from [intro.compliance]:

> 7 Two kinds of implementations are defined: a hosted implementation and a freestanding implementation. For a hosted implementation, this document defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation defined set of libraries that includes certain language-support libraries (20.5.1.3).

The main people served by the current freestanding definition are people writing their own hosted C++ standard library to sit atop the compiler author's freestanding implementation (i.e. the STLport use case). The freestanding portions contain all the functions and types known to the compiler that can't easily be authored in a cross-compiler manner.

The current set of freestanding libraries provides too little to kernel and embedded programmers. Why should a systems programmer need to rewrite `std::sort()` or `std::memcpy()`?

I propose we provide the (nearly) maximal subset of the library that does not require an OS or space overhead. In order to continue supporting the "layered" C++ standard library users, we will continue to provide the (nearly) minimal subset of the library needed to support all the language features, even if these features have space overhead. Freestanding library features requiring OS support will be removed. Language features requiring space overhead or OS support will remain intact.

II. Motivation

Systems programmers want to sort things. They want to use move semantics. They may even want to bundle the arguments of a variadic template function into a `std::tuple`. These are all reasonable things to do on a system with no operating system and kilobytes of storage. The C++ standard even has reasonable specifications for these operations in the context of a tiny, OS-less system. However, systems programmers must currently rely on either hand-rolled code or implementer extensions in order to get these facilities.

Systems programmers don't have a guide as to what C++ library facilities will work without trying them. The standard says `atomic_load` will work; `memcpy` will probably work; but will `stable_sort`? Standardizing the subset of implementable C++ that is usable in a freestanding environment would provide clarity here, and help to educate systems programmers.

III. Current State

There were some presentations at recent CppCons where others made a more full featured C++ work in the Linux and Windows kernels [Quinn2016] [Baker2017]. In both of these cases, C++ was being used in a sandboxed / pseudo-virtualized way. C++ code (with exceptions and RTTI) was being run in the kernel context, but very few calls were made between the C++ code and the operating system kernel. The C++ code was in a guest operating system. This proposal should make it reasonable to have C++ code interact closely with the internals of a host operating system, perhaps in the context of a driver.

The Microsoft Windows kernel and Apple Mac OSX kernel both currently support limited, non-compliant subsets of C++ for driver writers. The Linux kernel does not support C++ officially, though with a fair amount of work on the part of the driver developer, C++ can be made to work. Drivers written in C++ are highly unlikely to be accepted in the upstream Linux source repositories.

IncludeOS [Bratterud2017] is an OS primarily intended for running in VMs, though some bare metal support has been tested. One might expect such a project to use a freestanding implementation as a base, but instead, it starts with a hosted implmentation of C++ and drops support for the impractical parts (threads and filestreams in particular).

Out of libstdc++, libc++, and Microsoft's Visual Studio STL, only libstdc++ has any relevant mention of "freestanding" or "hosted". In practice, users take a hosted implementation of C++ and use it as-is in situations where it was never intended. This means that all the headers tend to be available, but only a few of the headers actually work. Many of the headers that work aren't marked freestanding. Some headers have parts that could work, except they are mixed with other parts that won't work. For example, `iterator_traits` in `<iterator>` is fine, but the implementation details of the stream iterators cause build errors with the `/kernel` flag in Microsoft Visual Studio 2017.

IV. Scope

The current scope of this proposal is limited to the freestanding standard library available to systems and embedded programming.

This paper is currently concerned with the divisions of headers and library functions as they were in C++17. "Standard Library Modules" (P0581) discusses how the library will be split up in a post-modules world. This paper may influence the direction of P0581, but this paper won't make any modules recommendations.

I could see the scope increasing to the standard library availability on GPUs. It also occurs to me that the list of standard library functions suitable for a freestanding implementation is likely a large subset of the list of functions that could be marked `constexpr`, and conditionally `noexcept`. This paper will not attempt to address those items though.

V. Impact on the standard

Rather than list all of the facilities available to a freestanding implementation in one place, as is currently done in [compliance], the standard would tag each header, class, or function that is available in a freestanding implementation. I expect this to be a large number of small edits, but the edits would have easy to understand ramifications throughout the standard.

There is precedent for this kind of tagging in other specification documents. The ECMAScript Language Specification has optional support for ECMA-402 (internationalization). The impact of ECMA-402 is called out explicitly in several places. POSIX tags functions as supported in base, XSI, or in one of many option groups.

There were some conversations in the 2017 Albuquerque meeting around adding another class of conforming implementation. I believe that such an action would be a mistake. Maintaining two classifications is difficult enough as is, and freestanding is often neglected. Adding another classification would magnify these problems. I also feel that the freestanding classification should be removed if no action is taken to make it more useful.

Naming alternatives

There was some desire to come up with a new name for "freestanding" in the 2017 Albuquerque meeting. This new name could better express the intended audience of such an implementation. My current recommendation will be to keep the name "freestanding", but I will suggest some alternatives just the same.

- barebones
- basic
- embedded
- freestanding
- kernel
- minimal
- OS-free
- OS-less
- skeleton
- standalone
- stripped
- system-free
- unsupported

VI. Impact on implementations

C++ standard library headers will likely need to add preprocessor feature toggles to portions of headers that would emit warnings or errors in freestanding mode. The precision and timeliness (compile time vs. link time) of errors remains a quality-of-implementation detail.

A minimal freestanding C11 standard library will not be sufficient to provide the C portions of the C++ standard library. `std::char_traits` and many of the function specializations in `<algorithm>` are implemented in terms of non-freestanding C functions. In practice, most C libraries are not minimal freestanding C11 libraries. The optimized versions of the `<cstring>` and `<cwchar>` functions will typically be the same for both hosted and freestanding environments.

VII. Design decisions

Even more so than for a hosted implementation, systems and embedded programmers do not want to pay for what they don't use. As a consequence, I am not adding features that require global storage, even if that storage is immutable.

Note that the following concerns are not revolving around execution time performance. These are generally concerns about space overhead and correctness.

This proposal doesn't remove problematic features from the language, but it does make it so that the bulk of the freestanding standard library doesn't require those features. Users that disable the problematic features (as is existing practice) will still have portable portions of the standard library at their disposal.

Exceptions

Exceptions either require external jump tables or extra bookkeeping instructions. This consumes program storage space.

In the Itanium ABI, throwing an exception requires a heap allocation. In the Microsoft ABI, re-throwing an exception will consume surprisingly large amounts of stack space. Program storage space, heap space, and stack space are typically scarce resources in embedded development.

In environments with threads, exception handling requires the use of thread-local storage.

RTTI

RTTI requires extra data in vtables and extra classes that are difficult to optimize away, consuming program storage space.

Thread-local storage

Thread-local storage requires extra code in the operating system for support. In addition, if one thread uses thread-local storage, that cost is imposed on other threads.

The heap

The heap is a big set of global state. In addition, heap exhaustion is typically expressed via exception. Some embedded systems don't have a heap. In kernel environments, there is typically a heap, but there isn't a reasonable choice of *which* heap to use as the default. In the Windows kernel, the two best candidates for a default heap are the paged pool (plentiful available memory, but unsafe to use in many contexts), and the non-paged pool (safe to use, but limited capacity). The C++ implementation in the Windows kernel forces users to implement their own global `operator new` to make this decision.

Floating point

Many embedded systems don't have floating point hardware. Software emulated floating point can drag in large runtimes that are difficult to optimize away.

Most operating systems speed up system calls by not saving and restoring floating point state. That means that kernel uses of floating point operations require extra care to avoid corrupting user state.

Functions requiring global or thread-local storage

These functions have been omitted or removed from the freestanding library. Examples are the locale aware functions and the C random number functions.

Parallel algorithms

For the `<algorithms>` header, we would only be able to support sequential execution of parallel algorithms. Since this adds little value, the execution policy overloads will be omitted.

VIII. Technical Specifications

Facilities no longer required for freestanding implementations

Usages of the operating system

- `<new> hardware_destructive_interference_size`
- `<new> hardware_constructive_interference_size`

Facilities newly required for freestanding implementations

Portions of `<cstdlib>`

- `div_t`
- `ldiv_t`
- `lldiv_t`
- `EXIT_FAILURE`
- `EXIT_SUCCESS`
- `_Exit`
- `atoi`
- `atol`
- `atoll`
- `bsearch`
- `qsort`
- `abs(int)`
- `abs(long int)`
- `abs(long long int)`
- `labs`
- `div`
- `ldiv`
- `lldiv`

All the error `#defines` in `<cerrno>`, but not `errno`.

The `errc` enum from `<system_error>`.

All of `<utility>` and `<tuple>`.

Portions of `<memory>`.

- `pointer_traits`
- `align`
- 23.10.10, specialized algorithms

Most of `<functional>`. **Omit** the following.

- 23.14.13, polymorphic function wrappers (i.e. `std::function` and friends).
- 23.14.14.2, `boyer_moore_searcher`
- 23.14.14.3, `boyer_moore_horspool_searcher`

All of `<ratio>`.

Most of `<chrono>`. **Omit** 23.17.7, Clocks.

Portions of `<charconv>`.

- `to_chars_result`
- `from_chars_result`
- `to_chars`(integral)

- `from_chars`(integral)

The `char_traits` class from `<string>`.

Portions of `<cstring>`.

- `memcpy`
- `memmove`
- `strcpy`
- `strncpy`
- `strcat`
- `strncat`
- `memcmp`
- `strcmp`
- `strncmp`
- `memchr`
- `strchr`
- `strcspn`
- `strpbrk`
- `strrchr`
- `strspn`
- `strstr`
- `memset`
- `strlen`

Portions of `<cwchar>`.

- `wcscpy`
- `wcsncpy`
- `wmemcpy`
- `wmemmove`
- `wcscat`
- `wcsncat`
- `wcscmp`
- `wcsncmp`
- `wmemcmp`
- `wcschr`
- `wcscspn`
- `wcxpbrk`
- `wcsrchr`
- `wcsspn`
- `wcsstr`
- `wcstok`
- `wmemchr`
- `wcslen`
- `wmemset`

All of `<iterator>` except for the stream iterators and the insert iterators.

Most of `<algorithm>` and `<numeric>`. The ExecutionPolicy overloads will not be included. The following functions will be **omitted** due to the usage of temporary buffers:

- `stable_sort`
- `stable_partition`
- `inplace_merge`

Portions of `<random>`. The following portions will be **omitted**:

- `random_device`
- `uniform_real_distribution`
- `exponential_distribution`
- `gamma_distribution`
- `weibull_distribution`

- `extreme_value_distribution`
- `normal_distribution`
- `lognormal_distribution`
- `chi_squared_distribution`
- `cauchy_distribution`
- `fisher_f_distribution`
- `student_t_distribution`
- `piecewise_constant_distribution`
- `piecewise_linear_distribution`

A small portion of `<cmath>` will be present.

- `abs(int)`
- `abs(long int)`
- `abs(long long int)`

A portion of `<cinttypes>` will be present.

- `imaxabs`
- `imaxdiv`
- `abs(intmax_t)`
- `div(intmax_t, intmax_t)`

Notable omissions

`bitset` is not included because many of its functions throw as part of a range check.

`errno` is not included as it is global state. In addition, errno is best implemented as a thread-local variable.

`error_code`, `error_condition`, and `error_category` all have `string` in the interface.

Many string functions (`strtol` and family) rely on `errno`.

`string_view` and `array` have methods that can throw exceptions.

`assert` is not included as it requires a stderror stream.

`<variant>` and `<optional>` could work in a freestanding environment if their interfaces didn't rely on exceptions.

`<cctype>` and `<cwctype>` rely heavily on global locale data.

`unique_ptr` is generally used for heap management, but is occasionally used as a makeshift RAII object for other resources. The use of the `default_delete` class in the template parameters is what dooms this class. In the future, `unique_resource and scope_exit` may satisfy the non-deleting use cases that `unique_ptr` currently fills.

Potential removals

Here are some things that I am currently requiring, but could be convinced to remove.

- `<cwchar>`

The `<cwchar>` functions are implementable for freestanding environments, but infrequently used for systems programming. They do not exist in freestanding C11 implementations.

- `<random>`

I am not confident that the remaining distributions in `<random>` are implementable without using floating point. These classes also carry a high implementer burden.

Potential additions

Here are some things that I am not currently requiring, but could be convinced to add.

- `complex<integer>`

I currently omit the `complex` class entirely, but in theory, the integer version of `complex` are fine. I am unsure how many of the associated functions are implementable without floating point access though. I do not believe that `complex` for integer types is a widely used class.

- Floating point support

Perhaps we don't worry about library portability in all cases. Just because kernel modes can't easily use floating point doesn't mean that we should deny floating point to the embedded space. Do note that most of `<cmath>` has a dependency on `errno`.

- `errno` and string functions like `strtol`

While `errno` is global data, it isn't much global data. Thread safety is a concern for those platforms that have threading, but don't have thread-local storage.

- `unique_ptr`

There are uses for `unique_ptr` that don't require the heap, and can be made to work. Since we still require `new` and `delete` to exist, `default_delete` shouldn't have any problems compiling.

IX. Acknowledgements

Thanks to Brandon Streiff, Joshua Cannon, Phil Hindman, and Irwan Djajadi for reviewing this proposal.

Similar work was done in the C++11 timeframe by Lawrence Crowl and Alberto Ganesh Barbati in N3256.

CppCon talks on getting C++ support in various unusual environments:

[Baker2017] CppCon 2017: Billy Baker "Almost Unlimited Modern C++ in Kernel-Mode Applications"

[Quinn2016] CppCon 2016: Rian Quinn "Making C++ and the STL Work in the Linux / Windows Kernels"

[Bratterud2017] CppCon 2017: Alfred Bratterud "Deconstructing the OS: The devil's In the side effects"