Freestanding Proposal

Document number: D0829R1.3
Date: 2017-10-14
Reply-to: Ben Craig <ben dot craig at gmail dot com>
Audience: SG14, Library Evolution Working Group

Change history

R0 -> R1

R1 adds an abstract.
R1 no longer removes any existing freestanding libraries.
R1 no longer adds portions of classes to freestanding. It is now the entirety of a class or none of it. Related functions (like `operator<<`) may still be omitted even when the associated class is included.
R1 no longer gives implementations freedom to add `static_asserts` to allocation functions in freestanding mode.
`unique_ptr` is no longer included in freestanding, with the hopes that `unique_resource` and `scope_exit` will fill the void
R1 adds naming alternatives for freestanding.
R1 includes substantially less of `<random>`, and white-lists inclusions rather than black-lists omissions.
R1 adds some allocator machinery, though it does not add the default allocator.
R1 adds `<compare>`, as it seems to have been accidentally omitted from the freestanding headers list.

Abstract

Add everything to the freestanding implementation that can be implemented without OS calls and space overhead.

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. Language features requiring space overhead or OS support will remain intact.

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.

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.

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 availability of the standard library on GPUs.

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

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.

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.

Note that we cannot just take the list of `constexpr` or conditionally `noexcept` functions and make those functions the freestanding subset. We also can't do the reverse, and make everything freestanding `constexpr` or conditionally `noexcept`. `memcpy` cannot currently be made `constexpr` because it must convert from `cv void*` to `unsigned char[]`. Several floating point functions could be made `constexpr`, but would not be permitted in freestanding. The "Lakos Rule"[Meredith11] prohibits standard library functions from being `noexcept`, unless they have a wide contract. Regardless, if a function or class is `constexpr` or `noexcept`, and it doesn't involve floating point, then that function or class is a strong candidate to be put into freestanding mode.

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.

Technical Specifications

Complete headers newly required for freestanding implementations

- `<utility>`
- `<tuple>`
- `<ratio>`
- `<compare>`

Partial headers newly required for freestanding implementations

Portions of `<cstdlib>`

- `size_t`
- `div_t`
- `ldiv_t`
- `lldiv_t`
- `NULL`
- `EXIT_FAILURE`
- `EXIT_SUCCESS`

- `_Exit`
- `atoi`
- `atol`
- `atoll`
- `bsearch`
- `qsort`
- `abs(int)`
- `abs(long int)`
- `abs(long long int)`
- `labs`
- `llabs`
- `div`
- `ldiv`
- `lldiv`

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

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

Portions of `<memory>`.

- `pointer_traits`
- `to_address`
- `align`
- `allocator_arg_t`
- `allocator_arg`
- `uses_allocator`
- `allocator_traits`
- 23.10.11, specialized algorithms, except for the ExecutionPolicy overloads
- `uses_allocator_v`

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`

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 included:

- linear_congruential_engine
- mersenne_twister_engine
- subtract_with_carry_engine
- discard_block_engine
- independent_bits_engine

- shuffle_order_engine
- [rand.predef]
- uniform_int_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.

strtok and rand aren't required to use thread-local storage, but good implementations do. I don't want to encourage bad implementations.

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, as default_delete is very involved with the heap. In the future, unique_resource and scope_exit may satisfy the non-deleting use cases that unique_ptr currently fills.

seed_seq uses the heap. Users can create their own classes satisfying the seed sequence requirements if they wish to use the Sseq constructors on the engine templates.

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.

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.

Wording

Wording is based off of Working Draft, Standard for Programming Language C++, N4713.

Add a new subclause [freestanding.membership] after [objects.within.classes]:

20.4.2.5 Freestanding membership [freestanding.membership]

1 Several headers are required to be implemented in a freestanding implementation. The synopsis for these headers will start with a comment that ends with *freestanding*, as in:

```
// freestanding
namespace std {
```

2 Several macros, values, types, classes, class templates, functions, function templates, and objects are required to be implemented in a freestanding implementation, even when the entirety of the containing header is not required to be implemented in a freestanding environment. The declarations for such entities will be followed with a comment that ends with *freestanding*, as in:

```
#define E2BIG see below // freestanding
```

3 Classes and class templates that are required to be implemented in a freestanding implementation shall be implemented in their entirety.

Rework Table 19 referenced from 20.5.1.3 [compliance]:

~~Table 19 — C++ headers for freestanding implementations~~

~~Subclause~~		~~Header(s)~~
		~~`<iso646>`~~
~~21.2~~	~~Types~~	~~`<cstddef>`~~
~~21.3~~	~~Implementation properties~~	~~`<cfloat>` `<limits>` `<climits>`~~
~~21.4~~	~~Integer types~~	~~`<cstdint>`~~
~~21.5~~	~~Start and termination~~	~~`<cstdlib>`~~
~~21.6~~	~~Dynamic memory management~~	~~`<new>`~~
~~21.7~~	~~Type identification~~	~~`<typeinfo>`~~
~~21.8~~	~~Exception handling~~	~~`<exception>`~~
~~21.9~~	~~Initializer lists~~	~~`<initializer_list>`~~
~~21.11~~	~~Other runtime support~~	~~`<cstdarg>`~~

~~23.15~~	~~Type traits~~	~~<type_traits>~~
~~Clause 32~~	~~Atomics~~	~~<atomic>~~
~~D.4.2, D.4.3~~	~~Deprecated headers~~	~~<cstdalign> <cstdbool>~~

Table 19 — C++ headers for freestanding implementations

<algorithm>	<cmath>	<cwchar>	<random>
<atomic>	<compare>	<exception>	<ratio>
<cerrno>	<cstdalign>	<functional>	<string>
<cfloat>	<cstdarg>	<initializer_list>	<system_error>
<charconv>	<cstdbool>	<iterator>	<tuple>
<chrono>	<cstddef>	<limits>	<type_traits>
<cinttypes>	<cstdint>	<memory>	<typeinfo>
<ciso646>	<cstdlib>	<new>	<utility>
<climits>	<cstring>	<numeric>	

Change in [compliance] (20.5.1.3) paragraph 3:

~~The supplied version of the header <cstdlib> shall declare at least the functions abort, atexit, at_quick_exit, exit, and quick_exit (21.5). The other headers listed in this table shall meet the same requirements as for a hosted implementation.~~ **The supplied versions of the headers <atomic>, <cfloat>, <ciso646>, <climits>, <compare>, <cstdalign>, <cstdarg>, <cstdbool>, <cstddef>, <cstdint>, <exception>, <initializer_list>, <limits>, <new>, <ratio>, <tuple>, <type_traits>, <typeinfo>, and <utility> shall meet the same requirements as for a hosted implementation. The other headers listed in this table shall meet the requirements for a freestanding implementation, as specified in the respective header synopsis.**

Change in [cstddef.syn] (21.2.1):

21.2.1 Header <cstddef> synopsis [cstddef.syn]

```
// freestanding
namespace std {
```

Change in [cstdlib.syn] (21.2.2):

21.2.2 Header <cstdlib> synopsis [cstdlib.syn]

```
namespace std {
  using size_t = see below; // freestanding
  using div_t = see below; // freestanding
  using ldiv_t = see below; // freestanding
  using lldiv_t = see below; // freestanding
}

#define NULL see below // freestanding
#define EXIT_FAILURE see below // freestanding
#define EXIT_SUCCESS see below // freestanding
#define RAND_MAX see below
#define MB_CUR_MAX see below

namespace std {
  // Exposition-only function type aliases
  extern "C" using c-atexit-handler = void();                          // exposition only
  extern "C++" using atexit-handler = void();                          // exposition only
  extern "C" using c-compare-pred = int(const void*, const void*);     // exposition only
```

```
extern "C++" using compare-pred = int(const void*, const void*);    // exposition only

// 21.5, start and termination
[[noreturn]] void abort() noexcept; // freestanding
int atexit(c-atexit-handler* func) noexcept; // freestanding
int atexit(atexit-handler* func) noexcept; // freestanding
int at_quick_exit(c-atexit-handler* func) noexcept; // freestanding
int at_quick_exit(atexit-handler* func) noexcept; // freestanding
[[noreturn]] void exit(int status); // freestanding
[[noreturn]] void _Exit(int status) noexcept; // freestanding
[[noreturn]] void quick_exit(int status) noexcept; // freestanding

char* getenv(const char* name);
int system(const char* string);

// 23.10.12, C library memory allocation
void* aligned_alloc(size_t alignment, size_t size);
void* calloc(size_t nmemb, size_t size);
void free(void* ptr);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);

double atof(const char* nptr);
int atoi(const char* nptr); // freestanding
long int atol(const char* nptr); // freestanding
long long int atoll(const char* nptr); // freestanding
double strtod(const char* nptr, char** endptr);
float strtof(const char* nptr, char** endptr);
long double strtold(const char* nptr, char** endptr);
long int strtol(const char* nptr, char** endptr, int base);
long long int strtoll(const char* nptr, char** endptr, int base);
unsigned long int strtoul(const char* nptr, char** endptr, int base);
unsigned long long int strtoull(const char* nptr, char** endptr, int base);

// 24.5.6, multibyte / wide string and character conversion functions
int mblen(const char* s, size_t n);
int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);

// 28.8, C standard library algorithms
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              c-compare-pred* compar); // freestanding
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              compare-pred* compar); // freestanding
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar); // freestanding
void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar); // freestanding

// 29.6.9, low-quality random number generation
int rand();
void srand(unsigned int seed);

// 29.9.2, absolute values
int abs(int j); // freestanding
long int abs(long int j); // freestanding
long long int abs(long long int j); // freestanding
float abs(float j);
double abs(double j);
long double abs(long double j);

long int labs(long int j); // freestanding
long long int llabs(long long int j); // freestanding

div_t div(int numer, int denom); // freestanding
ldiv_t div(long int numer, long int denom);             // see 20.2, freestanding
lldiv_t div(long long int numer, long long int denom);  // see 20.2, freestanding
ldiv_t ldiv(long int numer, long int denom); // freestanding
```

```
  lldiv_t lldiv(long long int numer, long long int denom); // freestanding
}
```

Change in [limits.syn] (21.3.2):

21.3.2 Header `<limits>` synopsis [limits.syn]

```
// freestanding
namespace std {
```

Change in [climits.syn] (21.3.5):

21.3.5 Header `<climits>` synopsis [climits.syn]

```
// freestanding
#define CHAR_BIT see below
```

Change in [cfloat.syn] (21.3.6):

21.3.6 Header `<cfloat>` synopsis [cfloat.syn]

```
// freestanding
#define FLT_ROUNDS see below
```

Change in [cstdint.syn] (21.4.1):

21.4.1 Header `<cstdint>` synopsis [cstdint.syn]

```
// freestanding
namespace std {
```

Change in [new.syn] (21.6.1):

21.6.1 Header `<new>` synopsis [new.syn]

```
// freestanding
namespace std {
```

Change in [typeinfo.syn] (21.7.1):

21.7.1 Header `<typeinfo>` synopsis [typeinfo.syn]

```
// freestanding
namespace std {
```

Change in [exception.syn] (21.8.1):

21.8.1 Header `<exception>` synopsis [exception.syn]

```
// freestanding
namespace std {
```

Change in [initializer_list.syn] (21.9.1):

21.9.1 Header `<initializer_list>` synopsis [initializer_list.syn]

```
// freestanding
namespace std {
```

Change in [cmp.syn] (21.10.1):

21.10.1 Header `<compare>` synopsis [cmp.syn]

1 The header `<compare>` specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (8.5.8).

```
// freestanding
namespace std {
```

Change in [cstdarg.syn] (21.11.1):

21.11.1 Header `<cstdarg>` synopsis [cstdarg.syn]

```
// freestanding
namespace std {
```

Change in [cerrno.syn] (22.4.1):

22.4.1 Header `<cerrno>` synopsis [cerrno.syn]

```
#define errno see below

#define E2BIG see below // freestanding
#define EACCES see below // freestanding
#define EADDRINUSE see below // freestanding
#define EADDRNOTAVAIL see below // freestanding
#define EAFNOSUPPORT see below // freestanding
#define EAGAIN see below // freestanding
#define EALREADY see below // freestanding
#define EBADF see below // freestanding
#define EBADMSG see below // freestanding
#define EBUSY see below // freestanding
#define ECANCELED see below // freestanding
#define ECHILD see below // freestanding
#define ECONNABORTED see below // freestanding
#define ECONNREFUSED see below // freestanding
#define ECONNRESET see below // freestanding
#define EDEADLK see below // freestanding
#define EDESTADDRREQ see below // freestanding
#define EDOM see below // freestanding
```

```
#define EEXIST see below // freestanding
#define EFAULT see below // freestanding
#define EFBIG see below // freestanding
#define EHOSTUNREACH see below // freestanding
#define EIDRM see below // freestanding
#define EILSEQ see below // freestanding
#define EINPROGRESS see below // freestanding
#define EINTR see below // freestanding
#define EINVAL see below // freestanding
#define EIO see below // freestanding
#define EISCONN see below // freestanding
#define EISDIR see below // freestanding
#define ELOOP see below // freestanding
#define EMFILE see below // freestanding
#define EMLINK see below // freestanding
#define EMSGSIZE see below // freestanding
#define ENAMETOOLONG see below // freestanding
#define ENETDOWN see below // freestanding
#define ENETRESET see below // freestanding
#define ENETUNREACH see below // freestanding
#define ENFILE see below // freestanding
#define ENOBUFS see below // freestanding
#define ENODATA see below // freestanding
#define ENODEV see below // freestanding
#define ENOENT see below // freestanding
#define ENOEXEC see below // freestanding
#define ENOLCK see below // freestanding
#define ENOLINK see below // freestanding
#define ENOMEM see below // freestanding
#define ENOMSG see below // freestanding
#define ENOPROTOOPT see below // freestanding
#define ENOSPC see below // freestanding
#define ENOSR see below // freestanding
#define ENOSTR see below // freestanding
#define ENOSYS see below // freestanding
#define ENOTCONN see below // freestanding
#define ENOTDIR see below // freestanding
#define ENOTEMPTY see below // freestanding
#define ENOTRECOVERABLE see below // freestanding
#define ENOTSOCK see below // freestanding
#define ENOTSUP see below // freestanding
#define ENOTTY see below // freestanding
#define ENXIO see below // freestanding
#define EOPNOTSUPP see below // freestanding
#define EOVERFLOW see below // freestanding
#define EOWNERDEAD see below // freestanding
#define EPERM see below // freestanding
#define EPIPE see below // freestanding
#define EPROTO see below // freestanding
#define EPROTONOSUPPORT see below // freestanding
#define EPROTOTYPE see below // freestanding
#define ERANGE see below // freestanding
#define EROFS see below // freestanding
#define ESPIPE see below // freestanding
#define ESRCH see below // freestanding
#define ETIME see below // freestanding
#define ETIMEDOUT see below // freestanding
#define ETXTBSY see below // freestanding
#define EWOULDBLOCK see below // freestanding
#define EXDEV see below // freestanding
```

1 The meaning of the macros in this header is defined by the POSIX standard.

See also: ISO C 7.5

Change in [system_error.syn] (22.5.1):

22.5.1 Header <system_error> synopsis [system_error.syn]

```
namespace std {
  class error_category;
  const error_category& generic_category() noexcept;
  const error_category& system_category() noexcept;

  class error_code;
  class error_condition;
  class system_error;

  template<class T>
    struct is_error_code_enum : public false_type {};

  template<class T>
    struct is_error_condition_enum : public false_type {};

  enum class errc { // freestanding
    address_family_not_supported,       // EAFNOSUPPORT
    address_in_use,                     // EADDRINUSE
    address_not_available,              // EADDRNOTAVAIL
```

Change in [utility.syn] (23.2.1):

23.2.1 Header <utility> synopsis [utility.syn]

```
// freestanding
namespace std {
```

Change in [tuple.syn] (23.5.2):

23.5.2 Header <tuple> synopsis [tuple.syn]

```
// freestanding
namespace std {
```

Change in [memory.syn] (23.10.2):

23.10.2 Header <memory> synopsis [memory.syn]

1 The header <memory> defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, destroy objects, and construct multiple objects in uninitialized memory buffers (23.10.3–23.10.11). The header also defines the templates unique_ptr, shared_ptr, weak_ptr, and various function templates that operate on objects of these types (23.11).

```
namespace std {
  // 23.10.3, pointer traits
  template<class Ptr> struct pointer_traits; // freestanding
  template<class T> struct pointer_traits<T*>; // freestanding

  // 23.10.4, pointer conversion
  template<class Ptr>
    auto to_address(const Ptr& p) noexcept; // freestanding
  template<class T>
```

```
  constexpr T* to_address(T* p) noexcept; // freestanding

// 23.10.5, pointer safety
enum class pointer_safety { relaxed, preferred, strict };
void declare_reachable(void* p);
template<class T>
  T* undeclare_reachable(T* p);
void declare_no_pointers(char* p, size_t n);
void undeclare_no_pointers(char* p, size_t n);
pointer_safety get_pointer_safety() noexcept;

// 23.10.6, pointer alignment function
void* align(size_t alignment, size_t size, void*& ptr, size_t& space); // freestanding

// 23.10.7, allocator argument tag
struct allocator_arg_t { explicit allocator_arg_t() = default; }; // freestanding
inline constexpr allocator_arg_t allocator_arg{}; // freestanding

// 23.10.8, uses_allocator
template<class T, class Alloc> struct uses_allocator; // freestanding

// 23.10.9, allocator traits
template<class Alloc> struct allocator_traits; // freestanding

// 23.10.10, the default allocator
template<class T> class allocator;
template<class T, class U>
  bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
template<class T, class U>
  bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

// 23.10.11, specialized algorithms
template<class T>
  constexpr T* addressof(T& r) noexcept; // freestanding
template<class T>
  const T* addressof(const T&&) = delete; // freestanding
template<class ForwardIterator>
  void uninitialized_default_construct(ForwardIterator first, ForwardIterator last); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  void uninitialized_default_construct(ExecutionPolicy&& exec, // see 28.4.5
                                       ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ExecutionPolicy&& exec, // see 28.4.5
                                                    ForwardIterator first, Size n);
template<class ForwardIterator>
  void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);  // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  void uninitialized_value_construct(ExecutionPolicy&& exec, // see 28.4.5
                                     ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, // see 28.4.5
                                                  ForwardIterator first, Size n);
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result); // freestanding
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(ExecutionPolicy&& exec, // see 28.4.5
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result); // freestanding
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec, // see 28.4.5
```

```
                                    InputIterator first, Size n,
                                    ForwardIterator result);
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result); // freestanding
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(ExecutionPolicy&& exec, // see 28.4.5
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator> uninitialized_move_n(InputIterator first, Size n,
                                                            ForwardIterator result); // freestanding
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator> uninitialized_move_n(ExecutionPolicy&& exec, // see 28.4.5
                                                            InputIterator first, Size n,
                                                            ForwardIterator result);
template<class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy&& exec, // see 28.4.5
                          ForwardIterator first, ForwardIterator last, const T& x);
template<class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see 28.4.5
                                       ForwardIterator first, Size n, const T& x);
template<class T>
  void destroy_at(T* location); // freestanding
template<class ForwardIterator>
  void destroy(ForwardIterator first, ForwardIterator last); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  void destroy(ExecutionPolicy&& exec, // see 28.4.5
               ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator destroy_n(ForwardIterator first, Size n); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator destroy_n(ExecutionPolicy&& exec, // see 28.4.5
                            ForwardIterator first, Size n);

// 23.11.1, class template unique_ptr
template<class T> struct default_delete;
template<class T> struct default_delete<T[]>;
template<class T, class D = default_delete<T>> class unique_ptr;
template<class T, class D> class unique_ptr<T[], D>;

template<class T, class... Args> unique_ptr<T>
  make_unique(Args&&... args);                                          // T is not array
template<class T> unique_ptr<T>
  make_unique(size_t n);                                                // T is U[]
template<class T, class... Args>
  unspecified make_unique(Args&&...) = delete;                          // T is U[N]

template<class T, class D>
  void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template<class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template<class T, class D>
```

```
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template<class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template<class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template<class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template<class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

template<class E, class T, class Y, class D>
  basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);

// 23.11.2, class bad_weak_ptr
class bad_weak_ptr;

// 23.11.3, class template shared_ptr
template<class T> class shared_ptr;

// 23.11.3.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);                              // T is not array
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);              // T is not array

template<class T>
  shared_ptr<T> make_shared(size_t N);                                    // T is U[]
template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a, size_t N);                    // T is U[]

template<class T>
  shared_ptr<T> make_shared();                                            // T is U[N]
template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a);                              // T is U[N]

template<class T>
  shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u);       // T is U[]
template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a, size_t N,
                                const remove_extent_t<T>& u);             // T is U[]

template<class T> shared_ptr<T>
  make_shared(const remove_extent_t<T>& u);                               // T is U[N]
template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u); // T is U[N]

// 23.11.3.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

```
  template<class T, class U>
    bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
  template<class T, class U>
    bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

  template<class T>
    bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
  template<class T>
    bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
  template<class T>
    bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
  template<class T>
    bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
  template<class T>
    bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
  template<class T>
    bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
  template<class T>
    bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;

  // 23.11.3.8, shared_ptr specialized algorithms
  template<class T>
    void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

  // 23.11.3.9, shared_ptr casts
  template<class T, class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
  template<class T, class U>
    shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
  template<class T, class U>
    shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
  template<class T, class U>
    shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

  // 23.11.3.10, shared_ptr get_deleter
  template<class D, class T>
    D* get_deleter(const shared_ptr<T>& p) noexcept;

  // 23.11.3.11, shared_ptr I/O
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const shared_ptr<Y>& p);

  // 23.11.4, class template weak_ptr
  template<class T> class weak_ptr;

  // 23.11.4.6, weak_ptr specialized algorithms
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

  // 23.11.5, class template owner_less
  template<class T = void> struct owner_less;

  // 23.11.6, class template enable_shared_from_this
  template<class T> class enable_shared_from_this;

  // 23.11.7, hash support
  template<class T> struct hash;
  template<class T, class D> struct hash<unique_ptr<T, D>>;
  template<class T> struct hash<shared_ptr<T>>;

  // 23.11.8, atomic smart pointers
```

```
  template<class T> struct atomic<shared_ptr<T>>;
  template<class T> struct atomic<weak_ptr<T>>;

  // 23.10.8.1, uses_allocator
  template<class T, class Alloc>
    inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value; // freestanding
}
```

Change in [functional.syn] (23.14.1):

23.14.1 Header <functional> synopsis [functional.syn]

```
namespace std {
  // 23.14.4, invoke
  template<class F, class... Args>
    invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
      noexcept(is_nothrow_invocable_v<F, Args...>); // freestanding

  // 23.14.5, reference_wrapper
  template<class T> class reference_wrapper; // freestanding

  template<class T> reference_wrapper<T> ref(T&) noexcept; // freestanding
  template<class T> reference_wrapper<const T> cref(const T&) noexcept; // freestanding
  template<class T> void ref(const T&&) = delete; // freestanding
  template<class T> void cref(const T&&) = delete; // freestanding

  template<class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept; // freestanding
  template<class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept; // freestanding

  // 23.14.6, arithmetic operations
  template<class T = void> struct plus; // freestanding
  template<class T = void> struct minus; // freestanding
  template<class T = void> struct multiplies; // freestanding
  template<class T = void> struct divides; // freestanding
  template<class T = void> struct modulus; // freestanding
  template<class T = void> struct negate; // freestanding
  template<> struct plus<void>; // freestanding
  template<> struct minus<void>; // freestanding
  template<> struct multiplies<void>; // freestanding
  template<> struct divides<void>; // freestanding
  template<> struct modulus<void>; // freestanding
  template<> struct negate<void>; // freestanding

  // 23.14.7, comparisons
  template<class T = void> struct equal_to; // freestanding
  template<class T = void> struct not_equal_to; // freestanding
  template<class T = void> struct greater; // freestanding
  template<class T = void> struct less; // freestanding
  template<class T = void> struct greater_equal; // freestanding
  template<class T = void> struct less_equal; // freestanding
  template<> struct equal_to<void>; // freestanding
  template<> struct not_equal_to<void>; // freestanding
  template<> struct greater<void>; // freestanding
  template<> struct less<void>; // freestanding
  template<> struct greater_equal<void>; // freestanding
  template<> struct less_equal<void>; // freestanding

  // 23.14.8, logical operations
  template<class T = void> struct logical_and; // freestanding
  template<class T = void> struct logical_or; // freestanding
  template<class T = void> struct logical_not; // freestanding
  template<> struct logical_and<void>; // freestanding
  template<> struct logical_or<void>; // freestanding
  template<> struct logical_not<void>; // freestanding
```

```
// 23.14.9, bitwise operations
template<class T = void> struct bit_and; // freestanding
template<class T = void> struct bit_or; // freestanding
template<class T = void> struct bit_xor; // freestanding
template<class T = void> struct bit_not; // freestanding
template<> struct bit_and<void>; // freestanding
template<> struct bit_or<void>; // freestanding
template<> struct bit_xor<void>; // freestanding
template<> struct bit_not<void>; // freestanding

// 23.14.10, function template not_fn
template<class F> unspecified not_fn(F&& f); // freestanding

// 23.14.11, bind
template<class T> struct is_bind_expression; // freestanding
template<class T> struct is_placeholder; // freestanding

template<class F, class... BoundArgs>
  unspecified bind(F&&, BoundArgs&&...); // freestanding
template<class R, class F, class... BoundArgs>
  unspecified bind(F&&, BoundArgs&&...); // freestanding

namespace placeholders {
  // M is the implementation-defined number of placeholders
  see below _1; // freestanding
  see below _2; // freestanding
             .
             .
             .
  see below _M; // freestanding
}

// 23.14.12, member function adaptors
template<class R, class T>
  unspecified mem_fn(R T::*) noexcept; // freestanding

// 23.14.13, polymorphic function wrappers
class bad_function_call;

template<class> class function; // not defined
template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

template<class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template<class R, class... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
template<class R, class... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
template<class R, class... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

// 23.14.14, searchers
template<class ForwardIterator, class BinaryPredicate = equal_to<>>
  class default_searcher; // freestanding

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_searcher;

template<class RandomAccessIterator,
         class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_horspool_searcher;
```

```
  // 23.14.15, hash function primary template
  template<class T>
    struct hash; // freestanding

  // 23.14.11, function object binders
  template<class T>
    inline constexpr bool is_bind_expression_v = is_bind_expression<T>::value; // freestanding
  template<class T>
    inline constexpr int is_placeholder_v = is_placeholder<T>::value; // freestanding
}
```

Change in [meta.type.synop] (23.15.2):

23.15.2 Header <type_traits> synopsis [meta.type.synop]

```
// freestanding
namespace std {
```

Change in [ratio.syn] (23.16.2):

23.16.2 Header <ratio> synopsis [ratio.syn]

```
// freestanding
namespace std {
```

Change in [time.syn] (23.17.2):

23.17.2 Header <chrono> synopsis [time.syn]

```
namespace std {
  namespace chrono {
    // 23.17.5, class template duration
    template<class Rep, class Period = ratio<1>> class duration; // freestanding

    // 23.17.6, class template time_point
    template<class Clock, class Duration = typename Clock::duration> class time_point; // freestanding
  }

  // 23.17.4.3, common_type specializations
  template<class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
                       chrono::duration<Rep2, Period2>>; // freestanding

  template<class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>,
                       chrono::time_point<Clock, Duration2>>; // freestanding

  namespace chrono {
    // 23.17.4, customization traits
    template<class Rep> struct treat_as_floating_point; // freestanding
    template<class Rep> struct duration_values; // freestanding
    template<class Rep>
      inline constexpr bool treat_as_floating_point_v = treat_as_floating_point<Rep>::value; // freestanding

    // 23.17.5.5, duration arithmetic
    template<class Rep1, class Period1, class Rep2, class Period2>
      constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
        operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding
```

```
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
    operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
    operator*(const duration<Rep1, Period>& d, const Rep2& s); // freestanding
template<class Rep1, class Rep2, class Period>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
    operator*(const Rep1& s, const duration<Rep2, Period>& d); // freestanding
template<class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
    operator/(const duration<Rep1, Period>& d, const Rep2& s); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<Rep1, Rep2>
    operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period, class Rep2>
  constexpr duration<common_type_t<Rep1, Rep2>, Period>
    operator%(const duration<Rep1, Period>& d, const Rep2& s); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
    operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding

// 23.17.5.6, duration comparisons
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator==(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator< (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator> (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs); // freestanding

// 23.17.5.7, duration_cast
template<class ToDuration, class Rep, class Period>
  constexpr ToDuration duration_cast(const duration<Rep, Period>& d); // freestanding
template<class ToDuration, class Rep, class Period>
  constexpr ToDuration floor(const duration<Rep, Period>& d); // freestanding
template<class ToDuration, class Rep, class Period>
  constexpr ToDuration ceil(const duration<Rep, Period>& d); // freestanding
template<class ToDuration, class Rep, class Period>
  constexpr ToDuration round(const duration<Rep, Period>& d); // freestanding

// convenience typedefs
using nanoseconds  = duration<signed integer type of at least 64 bits, nano>; // freestanding
using microseconds = duration<signed integer type of at least 55 bits, micro>; // freestanding
using milliseconds = duration<signed integer type of at least 45 bits, milli>; // freestanding
using seconds      = duration<signed integer type of at least 35 bits>; // freestanding
using minutes      = duration<signed integer type of at least 29 bits, ratio<  60>>; // freestanding
using hours        = duration<signed integer type of at least 23 bits, ratio<3600>>; // freestanding

// 23.17.6.5, time_point arithmetic
template<class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
    operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding
template<class Rep1, class Period1, class Clock, class Duration2>
  constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
    operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs); // freestanding
template<class Clock, class Duration1, class Rep2, class Period2>
```

```
    constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
      operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
    constexpr common_type_t<Duration1, Duration2>
      operator-(const time_point<Clock, Duration1>& lhs,
                const time_point<Clock, Duration2>& rhs); // freestanding

  // 23.17.6.6, time_point comparisons
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator==(const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator!=(const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator< (const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator> (const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding
  template<class Clock, class Duration1, class Duration2>
     constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                               const time_point<Clock, Duration2>& rhs); // freestanding

  // 23.17.6.7, time_point_cast
  template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration>
      time_point_cast(const time_point<Clock, Duration>& t); // freestanding
  template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration> floor(const time_point<Clock, Duration>& tp); // freestanding
  template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp); // freestanding
  template<class ToDuration, class Clock, class Duration>
    constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp); // freestanding

  // 23.17.5.9, specialized algorithms
  template<class Rep, class Period>
    constexpr duration<Rep, Period> abs(duration<Rep, Period> d); // freestanding

  // 23.17.7, clocks
  class system_clock;
  class steady_clock;
  class high_resolution_clock;
}

inline namespace literals {
inline namespace chrono_literals {
  // 23.17.5.8, suffixes for duration literals
  constexpr chrono::hours                                operator""h(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified, ratio<3600, 1>> operator""h(long double); // freestanding
  constexpr chrono::minutes                              operator""min(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified, ratio<60, 1>>  operator""min(long double); // freestanding
  constexpr chrono::seconds                              operator""s(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified>                operator""s(long double); // freestanding
  constexpr chrono::milliseconds                         operator""ms(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified, milli>         operator""ms(long double); // freestanding
  constexpr chrono::microseconds                         operator""us(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified, micro>         operator""us(long double); // freestanding
  constexpr chrono::nanoseconds                          operator""ns(unsigned long long); // freestanding
  constexpr chrono::duration<unspecified, nano>          operator""ns(long double); // freestanding
}
}

namespace chrono {
  using namespace literals::chrono_literals; // freestanding
```

```
  }
}
```

Change in [charconv.syn] (23.20.1):

23.20.1 Header <charconv> synopsis [charconv.syn]

```
namespace std {
  // floating-point format for primitive numerical conversion
  enum class chars_format {
    scientific = unspecified,
    fixed = unspecified,
    hex = unspecified,
    general = fixed | scientific
  };

  // 23.20.2, primitive numerical output conversion
  struct to_chars_result {
    char* ptr;
    errc ec;
  }; // freestanding

  to_chars_result to_chars(char* first, char* last, see below value, int base = 10); // freestanding

  to_chars_result to_chars(char* first, char* last, float value);
  to_chars_result to_chars(char* first, char* last, double value);
  to_chars_result to_chars(char* first, char* last, long double value);

  to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
  to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
  to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

  to_chars_result to_chars(char* first, char* last, float value,
                           chars_format fmt, int precision);
  to_chars_result to_chars(char* first, char* last, double value,
                           chars_format fmt, int precision);
  to_chars_result to_chars(char* first, char* last, long double value,
                           chars_format fmt, int precision);

  // 23.20.3, primitive numerical input conversion
  struct from_chars_result {
    const char* ptr;
    errc ec;
  }; // freestanding

  from_chars_result from_chars(const char* first, const char* last,
                               see below& value, int base = 10); // freestanding

  from_chars_result from_chars(const char* first, const char* last, float& value,
                               chars_format fmt = chars_format::general);
  from_chars_result from_chars(const char* first, const char* last, double& value,
                               chars_format fmt = chars_format::general);
  from_chars_result from_chars(const char* first, const char* last, long double& value,
                               chars_format fmt = chars_format::general);
}
```

Change in [string.syn] (24.3.1):

24.3.1 Header <string> synopsis [string.syn]

```
#include <initializer_list>

namespace std {
```

```
  // 24.2, character traits
  template<class charT> struct char_traits; // freestanding
  template<> struct char_traits<char>; // freestanding
  template<> struct char_traits<char16_t>; // freestanding
  template<> struct char_traits<char32_t>; // freestanding
  template<> struct char_traits<wchar_t>; // freestanding

  // 24.3.2, basic_string
```

Change in [cstring.syn] (24.5.3):

24.5.3 Header <cstring> synopsis [cstring.syn]

```
namespace std {
  using size_t = see 21.2.4; // freestanding

  void* memcpy(void* s1, const void* s2, size_t n); // freestanding
  void* memmove(void* s1, const void* s2, size_t n); // freestanding
  char* strcpy(char* s1, const char* s2); // freestanding
  char* strncpy(char* s1, const char* s2, size_t n); // freestanding
  char* strcat(char* s1, const char* s2); // freestanding
  char* strncat(char* s1, const char* s2, size_t n); // freestanding
  int memcmp(const void* s1, const void* s2, size_t n); // freestanding
  int strcmp(const char* s1, const char* s2); // freestanding
  int strcoll(const char* s1, const char* s2);
  int strncmp(const char* s1, const char* s2, size_t n); // freestanding
  size_t strxfrm(char* s1, const char* s2, size_t n);
  const void* memchr(const void* s, int c, size_t n);  // see 20.2, freestanding
  void* memchr(void* s, int c, size_t n);  // see 20.2, freestanding
  const char* strchr(const char* s, int c);  // see 20.2, freestanding
  char* strchr(char* s, int c);  // see 20.2, freestanding
  size_t strcspn(const char* s1, const char* s2); // freestanding
  const char* strpbrk(const char* s1, const char* s2);  // see 20.2, freestanding
  char* strpbrk(char* s1, const char* s2);  // see 20.2, freestanding
  const char* strrchr(const char* s, int c);  // see 20.2, freestanding
  char* strrchr(char* s, int c);  // see 20.2, freestanding
  size_t strspn(const char* s1, const char* s2); // freestanding
  const char* strstr(const char* s1, const char* s2);  // see 20.2, freestanding
  char* strstr(char* s1, const char* s2);  // see 20.2, freestanding
  char* strtok(char* s1, const char* s2);
  void* memset(void* s, int c, size_t n); // freestanding
  char* strerror(int errnum);
  size_t strlen(const char* s); // freestanding
}

#define NULL see 21.2.3 // freestanding
```

Change in [cwchar.syn] (24.5.4):

24.5.4 Header <cwchar> synopsis [cwchar.syn]

```
namespace std {
  using size_t = see 21.2.4; // freestanding
  using mbstate_t = see below;
  using wint_t = see below;

  struct tm;

  int fwprintf(FILE* stream, const wchar_t* format, ...);
  int fwscanf(FILE* stream, const wchar_t* format, ...);
  int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
```

```
  int swscanf(const wchar_t* s, const wchar_t* format, ...);
  int vfwprintf(FILE* stream, const wchar_t* format, va_list arg);
  int vfwscanf(FILE* stream, const wchar_t* format, va_list arg);
  int vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
  int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
  int vwprintf(const wchar_t* format, va_list arg);
  int vwscanf(const wchar_t* format, va_list arg);
  int wprintf(const wchar_t* format, ...);
  int wscanf(const wchar_t* format, ...);
  wint_t fgetwc(FILE* stream);
  wchar_t* fgetws(wchar_t* s, int n, FILE* stream);
  wint_t fputwc(wchar_t c, FILE* stream);
  int fputws(const wchar_t* s, FILE* stream);
  int fwide(FILE* stream, int mode);
  wint_t getwc(FILE* stream);
  wint_t getwchar();
  wint_t putwc(wchar_t c, FILE* stream);
  wint_t putwchar(wchar_t c);
  wint_t ungetwc(wint_t c, FILE* stream);
  double wcstod(const wchar_t* nptr, wchar_t** endptr);
  float wcstof(const wchar_t* nptr, wchar_t** endptr);
  long double wcstold(const wchar_t* nptr, wchar_t** endptr);
  long int wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
  long long int wcstoll(const wchar_t* nptr, wchar_t** endptr, int base);
  unsigned long int wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);
  unsigned long long int wcstoull(const wchar_t* nptr, wchar_t** endptr, int base);
  wchar_t* wcscpy(wchar_t* s1, const wchar_t* s2); // freestanding
  wchar_t* wcsncpy(wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  wchar_t* wmemcpy(wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  wchar_t* wmemmove(wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  wchar_t* wcscat(wchar_t* s1, const wchar_t* s2); // freestanding
  wchar_t* wcsncat(wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  int wcscmp(const wchar_t* s1, const wchar_t* s2); // freestanding
  int wcscoll(const wchar_t* s1, const wchar_t* s2);
  int wcsncmp(const wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
  int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n); // freestanding
  const wchar_t* wcschr(const wchar_t* s, wchar_t c);  // see 20.2, freestanding
  wchar_t* wcschr(wchar_t* s, wchar_t c);  // see 20.2, freestanding
  size_t wcscspn(const wchar_t* s1, const wchar_t* s2); // freestanding
  const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);  // see 20.2, freestanding
  wchar_t* wcspbrk(wchar_t* s1, const wchar_t* s2);  // see 20.2, freestanding
  const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);  // see 20.2, freestanding
  wchar_t* wcsrchr(wchar_t* s, wchar_t c);  // see 20.2, freestanding
  size_t wcsspn(const wchar_t* s1, const wchar_t* s2); // freestanding
  const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);  // see 20.2, freestanding
  wchar_t* wcsstr(wchar_t* s1, const wchar_t* s2);  // see 20.2, freestanding
  wchar_t* wcstok(wchar_t* s1, const wchar_t* s2, wchar_t** ptr); // freestanding
  const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);  // see 20.2, freestanding
  wchar_t* wmemchr(wchar_t* s, wchar_t c, size_t n);  // see 20.2, freestanding
  size_t wcslen(const wchar_t* s); // freestanding
  wchar_t* wmemset(wchar_t* s, wchar_t c, size_t n); // freestanding
  size_t wcsftime(wchar_t* s, size_t maxsize, const wchar_t* format, const struct tm* timeptr);
  wint_t btowc(int c);
  int wctob(wint_t c);

  // 24.5.6, multibyte / wide string and character conversion functions
  int mbsinit(const mbstate_t* ps);
  size_t mbrlen(const char* s, size_t n, mbstate_t* ps);
  size_t mbrtowc(wchar_t* pwc, const char* s, size_t n, mbstate_t* ps);
  size_t wcrtomb(char* s, wchar_t wc, mbstate_t* ps);
  size_t mbsrtowcs(wchar_t* dst, const char** src, size_t len, mbstate_t* ps);
  size_t wcsrtombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);
}

#define NULL see 21.2.3 // freestanding
#define WCHAR_MAX see below // freestanding
```

```
#define WCHAR_MIN see below // freestanding
#define WEOF see below // freestanding
```

Change in [iterator.synopsis] (27.3):

27.3 Header <iterator> synopsis [iterator.synopsis]

```
namespace std {
  // 27.4, primitives
  template<class Iterator> struct iterator_traits; // freestanding
  template<class T> struct iterator_traits<T*>; // freestanding

  struct input_iterator_tag { }; // freestanding
  struct output_iterator_tag { }; // freestanding
  struct forward_iterator_tag: public input_iterator_tag { }; // freestanding
  struct bidirectional_iterator_tag: public forward_iterator_tag { }; // freestanding
  struct random_access_iterator_tag: public bidirectional_iterator_tag { }; // freestanding

  // 27.4.3, iterator operations
  template<class InputIterator, class Distance>
    constexpr void
      advance(InputIterator& i, Distance n); // freestanding
  template<class InputIterator>
    constexpr typename iterator_traits<InputIterator>::difference_type
      distance(InputIterator first, InputIterator last); // freestanding
  template<class InputIterator>
    constexpr InputIterator
      next(InputIterator x,
           typename iterator_traits<InputIterator>::difference_type n = 1); // freestanding
  template<class BidirectionalIterator>
    constexpr BidirectionalIterator
      prev(BidirectionalIterator x,
           typename iterator_traits<BidirectionalIterator>::difference_type n = 1); // freestanding

  // 27.5, predefined iterators
  template<class Iterator> class reverse_iterator; // freestanding

  template<class Iterator1, class Iterator2>
    constexpr bool operator==(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding
  template<class Iterator1, class Iterator2>
    constexpr bool operator<(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding
  template<class Iterator1, class Iterator2>
    constexpr bool operator!=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding
  template<class Iterator1, class Iterator2>
    constexpr bool operator>(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding
  template<class Iterator1, class Iterator2>
    constexpr bool operator>=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding
  template<class Iterator1, class Iterator2>
    constexpr bool operator<=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y); // freestanding

  template<class Iterator1, class Iterator2>
    constexpr auto operator-(
      const reverse_iterator<Iterator1>& x,
```

```
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base()); // freestanding
template<class Iterator>
  constexpr reverse_iterator<Iterator>
    operator+(
  typename reverse_iterator<Iterator>::difference_type n,
  const reverse_iterator<Iterator>& x); // freestanding

template<class Iterator>
  constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i); // freestanding

template<class Container> class back_insert_iterator;
template<class Container>
  back_insert_iterator<Container> back_inserter(Container& x);

template<class Container> class front_insert_iterator;
template<class Container>
  front_insert_iterator<Container> front_inserter(Container& x);

template<class Container> class insert_iterator;
template<class Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

template<class Iterator> class move_iterator; // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator<=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator>(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding
template<class Iterator1, class Iterator2>
  constexpr bool operator>=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y); // freestanding

template<class Iterator1, class Iterator2>
  constexpr auto operator-(
  const move_iterator<Iterator1>& x,
  const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base()); // freestanding
template<class Iterator>
  constexpr move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x); // freestanding
template<class Iterator>
  constexpr move_iterator<Iterator> make_move_iterator(Iterator i); // freestanding

// 27.6, stream iterators
template<class T, class charT = char, class traits = char_traits<charT>,
         class Distance = ptrdiff_t>
class istream_iterator;
template<class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template<class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template<class T, class charT = char, class traits = char_traits<charT>>
    class ostream_iterator;

template<class charT, class traits = char_traits<charT>>
  class istreambuf_iterator;
template<class charT, class traits>
```

```
    bool operator==(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template<class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);

  template<class charT, class traits = char_traits<charT>>
    class ostreambuf_iterator;

  // 27.7, range access
  template<class C> constexpr auto begin(C& c) -> decltype(c.begin()); // freestanding
  template<class C> constexpr auto begin(const C& c) -> decltype(c.begin()); // freestanding
  template<class C> constexpr auto end(C& c) -> decltype(c.end()); // freestanding
  template<class C> constexpr auto end(const C& c) -> decltype(c.end()); // freestanding
  template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept; // freestanding
  template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept; // freestanding
  template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c)); // freestanding
  template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c)); // freestanding
  template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin()); // freestanding
  template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin()); // freestanding
  template<class C> constexpr auto rend(C& c) -> decltype(c.rend()); // freestanding
  template<class C> constexpr auto rend(const C& c) -> decltype(c.rend()); // freestanding
  template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]); // freestanding
  template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]); // freestanding
  template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il); // freestanding
  template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il); // freestanding
  template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c)); // freestanding
  template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c)); // freestanding

  // 27.8, container access
  template<class C> constexpr auto size(const C& c) -> decltype(c.size()); // freestanding
  template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept; // freestanding
  template<class C> [[nodiscard]] constexpr auto empty(const C& c) -> decltype(c.empty()); // freestanding
  template<class T, size_t N> [[nodiscard]] constexpr bool empty(const T (&array)[N]) noexcept; // freestanding
  template<class E> [[nodiscard]] constexpr bool empty(initializer_list<E> il) noexcept; // freestanding
  template<class C> constexpr auto data(C& c) -> decltype(c.data()); // freestanding
  template<class C> constexpr auto data(const C& c) -> decltype(c.data()); // freestanding
  template<class T, size_t N> constexpr T* data(T (&array)[N]) noexcept; // freestanding
  template<class E> constexpr const E* data(initializer_list<E> il) noexcept; // freestanding
}
```

Change in [algorithm.syn] (28.2):

28.2 Header `<algorithm>` synopsis [algorithm.syn]

```
#include <initializer_list>

namespace std {
  // 28.5, non-modifying sequence operations
  // 28.5.1, all of
  template<class InputIterator, class Predicate>
    constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred); // freestanding
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool all_of(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last, Predicate pred);

  // 28.5.2, any of
  template<class InputIterator, class Predicate>
    constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred); // freestanding
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool any_of(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last, Predicate pred);
```

```
// 28.5.3, none of
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool none_of(ExecutionPolicy&& exec, // see 28.4.5
               ForwardIterator first, ForwardIterator last, Predicate pred);

// 28.5.4, for each
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Function>
  void for_each(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last, Function f);
template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  ForwardIterator for_each_n(ExecutionPolicy&& exec, // see 28.4.5
                             ForwardIterator first, Size n, Function f);

// 28.5.5, find
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, // see 28.4.5
                       ForwardIterator first, ForwardIterator last,
                       const T& value);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, // see 28.4.5
                          ForwardIterator first, ForwardIterator last,
                          Predicate pred);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec, // see 28.4.5
                              ForwardIterator first, ForwardIterator last,
                              Predicate pred);

// 28.5.6, find end
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2); // freestanding
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

// 28.5.7, find first
template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
```

```
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2); // freestanding
template<class InputIterator, class ForwardIterator, class BinaryPredicate>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2,
                  BinaryPredicate pred);

// 28.5.8, adjacent find
template<class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator
    adjacent_find(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator first, ForwardIterator last,
                  BinaryPredicate pred);

// 28.5.9, count
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec, // see 28.4.5
          ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last, Predicate pred);

// 28.5.10, mismatch
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2); // freestanding
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred); // freestanding
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2); // freestanding
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
```

```
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

// 28.5.11, equal
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2); // freestanding
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred); // freestanding
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2); // freestanding
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool equal(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  bool equal(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

// 28.5.12, is permutation
template<class ForwardIterator1, class ForwardIterator2>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2); // freestanding
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, BinaryPredicate pred); // freestanding
template<class ForwardIterator1, class ForwardIterator2>
```

```
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2); // freestanding
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2,
                                BinaryPredicate pred); // freestanding

// 28.5.13, search
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2); // freestanding
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see 28.4.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    search(ExecutionPolicy&& exec, // see 28.4.5
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value); // freestanding
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator
    search_n(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);

template<class ForwardIterator, class Searcher>
  constexpr ForwardIterator
    search(ForwardIterator first, ForwardIterator last, const Searcher& searcher); // freestanding

// 28.6, mutating sequence operations
// 28.6.1, copy
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 copy(ExecutionPolicy&& exec, // see 28.4.5
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class Size,
         class ForwardIterator2>
```

```
  ForwardIterator2 copy_n(ExecutionPolicy&& exec, // see 28.4.5
                          ForwardIterator1 first, Size n,
                          ForwardIterator2 result);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2 copy_if(ExecutionPolicy&& exec, // see 28.4.5
                           ForwardIterator1 first, ForwardIterator1 last,
                           ForwardIterator2 result, Predicate pred);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result); // freestanding

// 28.6.2, move
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result); // freestanding
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2>
  ForwardIterator2 move(ExecutionPolicy&& exec, // see 28.4.5
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
template<class BidirectionalIterator1, class BidirectionalIterator2>
  constexpr BidirectionalIterator2
    move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result); // freestanding

// 28.6.3, swap
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ExecutionPolicy&& exec, // see 28.4.5
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2);
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b); // freestanding

// 28.6.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first, InputIterator last,
              OutputIterator result, UnaryOperation op); // freestanding
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  ForwardIterator2
    transform(ExecutionPolicy&& exec, // see 28.4.5
              ForwardIterator1 first, ForwardIterator1 last,
              ForwardIterator2 result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  ForwardIterator
    transform(ExecutionPolicy&& exec, // see 28.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);

// 28.6.5, replace
template<class ForwardIterator, class T>
  constexpr void replace(ForwardIterator first, ForwardIterator last,
```

```
                        const T& old_value, const T& new_value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  void replace(ExecutionPolicy&& exec, // see 28.4.5
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
                                        OutputIterator result,
                                        const T& old_value, const T& new_value); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 replace_copy(ExecutionPolicy&& exec, // see 28.4.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                           OutputIterator result,
                                           Predicate pred, const T& new_value); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec, // see 28.4.5
                                   ForwardIterator1 first, ForwardIterator1 last,
                                   ForwardIterator2 result,
                                   Predicate pred, const T& new_value);

// 28.6.6, fill
template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  void fill(ExecutionPolicy&& exec, // see 28.4.5
            ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator,
         class Size, class T>
  ForwardIterator fill_n(ExecutionPolicy&& exec, // see 28.4.5
                         ForwardIterator first, Size n, const T& value);

// 28.6.7, generate
template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  void generate(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last,
                Generator gen);
template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  ForwardIterator generate_n(ExecutionPolicy&& exec, // see 28.4.5
                             ForwardIterator first, Size n, Generator gen);

// 28.6.8, remove
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec, // see 28.4.5
                         ForwardIterator first, ForwardIterator last,
                         const T& value);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
```

```
                                    Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec, // see 28.4.5
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);
template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  ForwardIterator2
    remove_copy(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec, // see 28.4.5
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);

// 28.6.9, unique
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy&& exec, // see 28.4.5
                         ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy&& exec, // see 28.4.5
                         ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result); // freestanding
template<class InputIterator, class OutputIterator, class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);

// 28.6.10, reverse
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last); // freestanding
template<class ExecutionPolicy, class BidirectionalIterator>
  void reverse(ExecutionPolicy&& exec, // see 28.4.5
               BidirectionalIterator first, BidirectionalIterator last);
template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result); // freestanding
```

```
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  ForwardIterator
    reverse_copy(ExecutionPolicy&& exec, // see 28.4.5
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);

// 28.6.11, rotate
template<class ForwardIterator>
  ForwardIterator rotate(ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy&& exec, // see 28.4.5
                         ForwardIterator first,
                         ForwardIterator middle,
                         ForwardIterator last);
template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first, ForwardIterator1 middle,
                ForwardIterator1 last, ForwardIterator2 result);

// 28.6.12, sample
template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomBitGenerator&& g); // freestanding

// 28.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
  void shuffle(RandomAccessIterator first,
               RandomAccessIterator last,
               UniformRandomBitGenerator&& g); // freestanding

// 28.7.4, partitions
template<class InputIterator, class Predicate>
  constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy&& exec, // see 28.4.5
                      ForwardIterator first, ForwardIterator last, Predicate pred);

template<class ForwardIterator, class Predicate>
  ForwardIterator partition(ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator partition(ExecutionPolicy&& exec, // see 28.4.5
                            ForwardIterator first,
                            ForwardIterator last,
                            Predicate pred);
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(ExecutionPolicy&& exec, // see 28.4.5
                                         BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
template<class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false,
```

```
                  Predicate pred); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
  pair<ForwardIterator1, ForwardIterator2>
    partition_copy(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator first, ForwardIterator last,
                  ForwardIterator1 out_true, ForwardIterator2 out_false,
                  Predicate pred);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last,
                    Predicate pred); // freestanding

// 28.7, sorting and related operations
// 28.7.1, sorting
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp); // freestanding
template<class ExecutionPolicy, class RandomAccessIterator>
  void sort(ExecutionPolicy&& exec, // see 28.4.5
            RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy&& exec, // see 28.4.5
            RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);

template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  void stable_sort(ExecutionPolicy&& exec, // see 28.4.5
                   RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec, // see 28.4.5
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);

template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp); // freestanding
template<class ExecutionPolicy, class RandomAccessIterator>
  void partial_sort(ExecutionPolicy&& exec, // see 28.4.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy&& exec, // see 28.4.5
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last); // freestanding
template<class InputIterator, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
```

```
                      Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,  // see 28.4.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,  // see 28.4.5
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  bool is_sorted(ExecutionPolicy&& exec, // see 28.4.5
                 ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy&& exec, // see 28.4.5
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see 28.4.5
                    ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec, // see 28.4.5
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);

// 28.7.2, Nth element
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp); // freestanding
template<class ExecutionPolicy, class RandomAccessIterator>
  void nth_element(ExecutionPolicy&& exec, // see 28.4.5
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy&& exec, // see 28.4.5
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);

// 28.7.3, binary search
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value); // freestanding
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    lower_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp); // freestanding
```

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value); // freestanding
template<class ForwardIterator, class T, class Compare>
  constexpr ForwardIterator
    upper_bound(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp); // freestanding

template<class ForwardIterator, class T>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value); // freestanding
template<class ForwardIterator, class T, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    equal_range(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp); // freestanding

template<class ForwardIterator, class T>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value); // freestanding
template<class ForwardIterator, class T, class Compare>
  constexpr bool
    binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp); // freestanding

// 28.7.5, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result); // freestanding
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see 28.4.5
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    merge(ExecutionPolicy&& exec, // see 28.4.5
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);

template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec, // see 28.4.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
```

```
  void inplace_merge(ExecutionPolicy&& exec, // see 28.4.5
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

// 28.7.6, set operations
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2); // freestanding
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool includes(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool includes(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result); // freestanding
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
              set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_union(ExecutionPolicy&& exec, // see 28.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
    set_union(ExecutionPolicy&& exec, // see 28.4.5
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result); // freestanding
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  ForwardIterator
```

```
    set_intersection(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);

  template<class InputIterator1, class InputIterator2, class OutputIterator>
    constexpr OutputIterator
      set_difference(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result); // freestanding
  template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
    constexpr OutputIterator
      set_difference(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp); // freestanding
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class ForwardIterator>
    ForwardIterator
      set_difference(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class ForwardIterator, class Compare>
    ForwardIterator
      set_difference(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);

  template<class InputIterator1, class InputIterator2, class OutputIterator>
    constexpr OutputIterator
      set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result); // freestanding
  template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
    constexpr OutputIterator
      set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result, Compare comp); // freestanding
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class ForwardIterator>
    ForwardIterator
      set_symmetric_difference(ExecutionPolicy&& exec, // see 28.4.5
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2, ForwardIterator2 last2,
                               ForwardIterator result);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class ForwardIterator, class Compare>
    ForwardIterator
      set_symmetric_difference(ExecutionPolicy&& exec, // see 28.4.5
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2, ForwardIterator2 last2,
                               ForwardIterator result, Compare comp);

  // 28.7.7, heap operations
  template<class RandomAccessIterator>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last); // freestanding
  template<class RandomAccessIterator, class Compare>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp); // freestanding

  template<class RandomAccessIterator>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last); // freestanding
  template<class RandomAccessIterator, class Compare>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp); // freestanding

  template<class RandomAccessIterator>
```

```
  void make_heap(RandomAccessIterator first, RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp); // freestanding

template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp); // freestanding

template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp); // freestanding
template<class ExecutionPolicy, class RandomAccessIterator>
  bool is_heap(ExecutionPolicy&& exec, // see 28.4.5
               RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  bool is_heap(ExecutionPolicy&& exec, // see 28.4.5
               RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last); // freestanding
template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp); // freestanding
template<class ExecutionPolicy, class RandomAccessIterator>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see 28.4.5
                  RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec, // see 28.4.5
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

// 28.7.8, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b); // freestanding
template<class T, class Compare>
  constexpr const T& min(const T& a, const T& b, Compare comp); // freestanding
template<class T>
  constexpr T min(initializer_list<T> t); // freestanding
template<class T, class Compare>
  constexpr T min(initializer_list<T> t, Compare comp); // freestanding

template<class T> constexpr const T& max(const T& a, const T& b); // freestanding
template<class T, class Compare>
  constexpr const T& max(const T& a, const T& b, Compare comp); // freestanding
template<class T>
  constexpr T max(initializer_list<T> t); // freestanding
template<class T, class Compare>
  constexpr T max(initializer_list<T> t, Compare comp); // freestanding

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b); // freestanding
template<class T, class Compare>
  constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp); // freestanding
template<class T>
  constexpr pair<T, T> minmax(initializer_list<T> t); // freestanding
template<class T, class Compare>
  constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp); // freestanding

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
```

```
                                      Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see 28.4.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy&& exec, // see 28.4.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see 28.4.5
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy&& exec, // see 28.4.5
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last); // freestanding
template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see 28.4.5
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec, // see 28.4.5
                   ForwardIterator first, ForwardIterator last, Compare comp);

// 28.7.9, bounded value
template<class T>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi); // freestanding
template<class T, class Compare>
  constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp); // freestanding

// 28.7.10, lexicographical comparison
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2); // freestanding
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 28.4.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 28.4.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);

// 28.7.11, three-way comparison algorithms
template<class T, class U>
  constexpr auto compare_3way(const T& a, const U& b); // freestanding
template<class InputIterator1, class InputIterator2, class Cmp>
  constexpr auto
```

```
    lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                 InputIterator2 b2, InputIterator2 e2,
                                 Cmp comp)
      -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>; // freestanding
  template<class InputIterator1, class InputIterator2>
    constexpr auto
      lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                   InputIterator2 b2, InputIterator2 e2); // freestanding

  // 28.7.12, permutations
  template<class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last); // freestanding
  template<class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp); // freestanding
  template<class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last); // freestanding
  template<class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp); // freestanding
}
```

Change in [rand.synopsis] (29.6.2):

29.6.2 Header `<random>` synopsis [rand.synopsis]

```
#include <initializer_list>

namespace std {
  // 29.6.3.1, class template linear_congruential_engine
  template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine; // freestanding

  // 29.6.3.2, class template mersenne_twister_engine
  template<class UIntType, size_t w, size_t n, size_t m, size_t r,
           UIntType a, size_t u, UIntType d, size_t s,
           UIntType b, size_t t,
           UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine; // freestanding

  // 29.6.3.3, class template subtract_with_carry_engine
  template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine; // freestanding

  // 29.6.4.2, class template discard_block_engine
  template<class Engine, size_t p, size_t r>
    class discard_block_engine; // freestanding

  // 29.6.4.3, class template independent_bits_engine
  template<class Engine, size_t w, class UIntType>
    class independent_bits_engine; // freestanding

  // 29.6.4.4, class template shuffle_order_engine
  template<class Engine, size_t k>
    class shuffle_order_engine; // freestanding

  // 29.6.5, engines and engine adaptors with predefined parameters
  using minstd_rand0  = see below; // freestanding
  using minstd_rand   = see below; // freestanding
  using mt19937       = see below; // freestanding
  using mt19937_64    = see below; // freestanding
  using ranlux24_base = see below; // freestanding
  using ranlux48_base = see below; // freestanding
```

```
  using ranlux24      = see below; // freestanding
  using ranlux48      = see below; // freestanding
  using knuth_b       = see below; // freestanding

  using default_random_engine = see below; // freestanding

  // 29.6.6, class random_device
  class random_device;

  // 29.6.7.1, class seed_seq
  class seed_seq;

  // 29.6.7.2, function template generate_canonical
  template<class RealType, size_t bits, class URBG>
    RealType generate_canonical(URBG& g);

  // 29.6.8.2.1, class template uniform_int_distribution
  template<class IntType = int>
    class uniform_int_distribution; // freestanding

  // 29.6.8.2.2, class template uniform_real_distribution
  template<class RealType = double>
    class uniform_real_distribution;
```

Change in [numeric.ops.overview] (29.8.1):

29.8.1 Header <numeric> synopsis [numeric.ops.overview]

```
namespace std {
  // 29.8.2, accumulate
  template<class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init); // freestanding
  template<class InputIterator, class T, class BinaryOperation>
    T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op); // freestanding

  // 29.8.3, reduce
  template<class InputIterator>
    typename iterator_traits<InputIterator>::value_type
      reduce(InputIterator first, InputIterator last); // freestanding
  template<class InputIterator, class T>
    T reduce(InputIterator first, InputIterator last, T init); // freestanding
  template<class InputIterator, class T, class BinaryOperation>
    T reduce(InputIterator first, InputIterator last, T init, BinaryOperation binary_op); // freestanding
  template<class ExecutionPolicy, class ForwardIterator>
    typename iterator_traits<ForwardIterator>::value_type
      reduce(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last);
  template<class ExecutionPolicy, class ForwardIterator, class T>
    T reduce(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last, T init);
  template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
    T reduce(ExecutionPolicy&& exec, // see 28.4.5
             ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op);

  // 29.8.4, inner product
  template<class InputIterator1, class InputIterator2, class T>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init); // freestanding
  template<class InputIterator1, class InputIterator2, class T,
           class BinaryOperation1, class BinaryOperation2>
    T inner_product(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init,
                    BinaryOperation1 binary_op1,
                    BinaryOperation2 binary_op2); // freestanding
```

```
// 29.8.5, transform reduce
template<class InputIterator1, class InputIterator2, class T>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init); // freestanding
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2); // freestanding
template<class InputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  T transform_reduce(InputIterator first, InputIterator last,
                     T init,
                     BinaryOperation binary_op, UnaryOperation unary_op); // freestanding
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T>
  T transform_reduce(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T transform_reduce(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2,
                     T init,
                     BinaryOperation1 binary_op1,
                     BinaryOperation2 binary_op2);
template<class ExecutionPolicy,
         class ForwardIterator, class T,
         class BinaryOperation, class UnaryOperation>
  T transform_reduce(ExecutionPolicy&& exec, // see 28.4.5
                     ForwardIterator first, ForwardIterator last,
                     T init,
                     BinaryOperation binary_op, UnaryOperation unary_op);

// 29.8.6, partial sum
template<class InputIterator, class OutputIterator>
  OutputIterator partial_sum(InputIterator first,
                             InputIterator last,
                             OutputIterator result); // freestanding
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator partial_sum(InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             BinaryOperation binary_op); // freestanding

// 29.8.7, exclusive scan
template<class InputIterator, class OutputIterator, class T>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                T init); // freestanding
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
  OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                T init, BinaryOperation binary_op); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result,
                                  T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation>
  ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                  ForwardIterator1 first, ForwardIterator1 last,
```

```
                                ForwardIterator2 result,
                                T init, BinaryOperation binary_op);

// 29.8.8, inclusive scan
template<class InputIterator, class OutputIterator>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result); // freestanding
template<class InputIterator, class OutputIterator, class BinaryOperation>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryOperation binary_op); // freestanding
template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
  OutputIterator inclusive_scan(InputIterator first, InputIterator last,
                                OutputIterator result,
                                BinaryOperation binary_op, T init); // freestanding
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class T>
  ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result,
                                BinaryOperation binary_op, T init);

// 29.8.9, transform exclusive scan
template<class InputIterator, class OutputIterator, class T,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_exclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          T init,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op); // freestanding
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_exclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                            ForwardIterator1 first, ForwardIterator1 last,
                                            ForwardIterator2 result,
                                            T init,
                                            BinaryOperation binary_op,
                                            UnaryOperation unary_op);

// 29.8.10, transform inclusive scan
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op); // freestanding
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation, class T>
  OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                          OutputIterator result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op,
                                          T init); // freestanding
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation>
  ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                            ForwardIterator1 first, ForwardIterator1 last,
```

```
                                          ForwardIterator2 result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op);
  template<class ExecutionPolicy,
           class ForwardIterator1, class ForwardIterator2,
           class BinaryOperation, class UnaryOperation, class T>
    ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
                                          ForwardIterator1 first, ForwardIterator1 last,
                                          ForwardIterator2 result,
                                          BinaryOperation binary_op,
                                          UnaryOperation unary_op,
                                          T init);

  // 29.8.11, adjacent difference
  template<class InputIterator, class OutputIterator>
    OutputIterator adjacent_difference(InputIterator first,
                                       InputIterator last,
                                       OutputIterator result); // freestanding
  template<class InputIterator, class OutputIterator, class BinaryOperation>
    OutputIterator adjacent_difference(InputIterator first,
                                       InputIterator last,
                                       OutputIterator result,
                                       BinaryOperation binary_op); // freestanding
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
    ForwardIterator2 adjacent_difference(ExecutionPolicy&& exec, // see 28.4.5
                                         ForwardIterator1 first,
                                         ForwardIterator1 last,
                                         ForwardIterator2 result);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class BinaryOperation>
    ForwardIterator2 adjacent_difference(ExecutionPolicy&& exec, // see 28.4.5
                                         ForwardIterator1 first,
                                         ForwardIterator1 last,
                                         ForwardIterator2 result,
                                         BinaryOperation binary_op);

  // 29.8.12, iota
  template<class ForwardIterator, class T>
    void iota(ForwardIterator first, ForwardIterator last, T value); // freestanding

  // 29.8.13, greatest common divisor
  template<class M, class N>
    constexpr common_type_t<M,N> gcd(M m, N n); // freestanding

  // 29.8.14, least common multiple
  template<class M, class N>
    constexpr common_type_t<M,N> lcm(M m, N n); // freestanding
}
```

Change in [cmath.syn] (29.9.1):

```
  // 29.9.2, absolute values
  int abs(int j); // freestanding
  long int abs(long int j);// freestanding
  long long int abs(long long int j); // freestanding
  float abs(float j);
  double abs(double j);
  long double abs(long double j);
```

Change in [cinttypes.syn] (30.12.2):

30.12.2 Header `<cinttypes>` synopsis [cinttypes.syn]

```
#include <cstdint>  // see [cstdint.syn]

namespace std {
  using imaxdiv_t = see below; // freestanding

  intmax_t imaxabs(intmax_t j); // freestanding
  imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom); // freestanding
  intmax_t strtoimax(const char* nptr, char** endptr, int base);
  uintmax_t strtoumax(const char* nptr, char** endptr, int base);
  intmax_t wcstoimax(const wchar_t* nptr, wchar_t** endptr, int base);
  uintmax_t wcstoumax(const wchar_t* nptr, wchar_t** endptr, int base);

  intmax_t abs(intmax_t);  // optional, see below, freestanding
  imaxdiv_t div(intmax_t, intmax_t);  // optional, see below, freestanding
}
```

Change in [atomics.syn] (32.2):

32.2 Header <atomic> synopsis [atomics.syn]

```
// freestanding
namespace std {
```

Change in [depr.cstdalign.syn] (D.4.2):

D.4.2 Header <cstdalign> synopsis [depr.cstdalign.syn]

```
// freestanding
#define __alignas_is_defined 1
```

Change in [depr.cstdbool.syn] (D.4.3):

D.4.3 Header <cstdbool> synopsis [depr.cstdbool.syn]

```
// freestanding
#define __bool_true_false_are_defined 1
```

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"

[Meredith11] noexcept Prevents Library Validation. A. Meredith; J. Lakos. N3248.