C++26 should refer to C23 not C17

Doc. no.: P3348R4
Author: Jonathan Wakely
Audience: LWG
Date: 2025-06-19

Revision History:

- R4: Remove stray “behavior is undefined unless” words from specification of realloc. Clarify drafting note for cuchar functions being added to [c.mb.wcs]. Strike asctime and ctime from [ctime.syn] and add wording for them to [depr.format]. Add additional context to [cstdint.syn] edits. Added change to [diff.char16].
- R3: Added nextup, nextdown and friends. Added const-correct overloads of bsearch. Replaced realloc wording. Added freestanding comment to memalignment. Fixed subclause numbering to account for new Text processing clause in current draft. Added a subset of __STDC_VERSION_xxx_H__ feature test macros. Added wording for va_start.
- R2: Links to related papers P3479R0 and P2746R6 on floating-point environments. SG6 review in Wrocław suggested moving all floating-point changes to a separate paper. Add missing declarations to <cstdlib>, <cstring>, and <ctime>, addressing “TODO” notes. Remove removal of `std::time_put` footnote which was handled editorially (#7553). Rebase on latest working draft, N5001.
- R1: Incorporated changes to [cstdarg.syn] from P2537R2. Added missing changes in [cctype.syn], [cwctype.syn], [cuchar.syn], and [c.mb.wcs]. Added `timespec_getres` function.

Introduction

There is a new version of the ISO C standard, so we should update our normative reference, and update the header synopses in the library clauses to match the content of the C standard library.

A similar change to rebase C++17 on C11 was previously done via P0063R3: C++17 should refer to C11 instead of C99 (see R0 for more detailed rationale) and then to trivially rebase on C17 via C++20 ballot comment US-019 (which just updated the normative reference, as there were no changes affecting the contents of the library in C17).

Changes in C23 since C17

The new C23 standard was published as ISO/IEC 9899:2024. All dated references to ISO/IEC 9899:2018 should change to reference the new document instead.

C23 adds versioning macros to its headers, for example `__STDC_VERSION_STDDEF_H__` is defined in `<stddef.h>`. In the cases of <stddef.h> and <stdlib.h>, the C++ header does not have the same content as the C header, so defining the macro to say “C23 support is present” seems misleading. The approach taken in the proposed wording is to define those macros when all features from the C header are present in the corresponding C++ header, and not define them otherwise.

C23 adds a typedef `nullptr_t` and a macro `unreachable` to `<stddef.h>`, which are already present in C++ (in `<cstddef>` and `<utility>` respectively).

The type `once_flag` and the function `call_once` are added to `<stdlib.h>`. We do not want those, as we have our own `std::once_flag` and `std::call_once` in `<mutex>`.

In C23 the `alignas` and `alignof` macros are now keywords. As a result, the header `<stdalign.h>` is empty in C23. In C++ it only defines two deprecated macros, which no longer exist at all in C23 (note related issues LWG 3827 and LWG 4036). I would prefer to deprecate the entire header, and eventually remove it (and add it to the zombie headers table) but that would be going beyond what C23 does. For the purposes of the current proposal, the header is unchanged and its content remains deprecated.

In C23 the `bool`, `true` and `false` macros are now keywords. As a result, the header `<stdbool.h>` is empty in C23, except for the obsolescent (i.e. deprecated) macro, `__bool_true_false_are_defined`. That macro is already deprecated in C++23. As with `stdalign.h`, I would prefer to deprecate then remove the entire header, but for the purposes of the current proposal, the headers are unchanged and their content remains deprecated.

The changes to `va_start` in `<stdarg.h>` have already been proposed separately in P2537R2, but that paper has not been updated after Core review. This paper supersedes P2537R2, aligning the specification for va_start more closely with the final spec in C23. The intention is that `va_start(ap)` is valid, and `va_start(ap, parmN)` is accepted for compatibility with previous revisions of C++ (without evaluating the second argument), but implementations should diagnose `va_start(ap,)` and `va_start(ap,,)`, `va_start(ap,x,y)`, `va_start(ap,foo<1>)`, and `va_start(ap,[unbalanced)`.

In C23, the `asctime` and `ctime` functions are deprecated. We should do the same. In C23 they are marked with the `[[deprecated]]` attribute. The wording below does the same, for consistency and to avoid diverging from the C header contents. But for C++ we might prefer to remove them from the synopsis and declare them in Annex D as deprecated, rather than using the attribute.

In C23 the `DECIMAL_DIG` macro is deprecated. We should do the same. This doesn't affect the `FLT_DECIMAL_DIG`, `DBL_DECIMAL_DIG` and `LDBL_DECIMAL_DIG` macros, which are the ones that are used by `std::numeric_limits`.

In C23 the use of `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM` and `LDBL_HAS_SUBNORM` macros is marked as obsolescent. The `std::numeric_limits` equivalents were already deprecated for C++23 by P2614R2. Guidance from SG6 in Wrocław was to be consistent with C, i.e. deprecate them.

In C23 the `INFINITY` and `NAN` macros are defined in `<float.h>`. In C17 they were defined in `<math.h>`, and are still there in C23, but defining them there is deprecated. Guidance from SG6 in Wrocław was to be consistent with C, i.e. move them and deprecate the old location.

`FLT_SNAN`, `DBL_SNAN`, `LDBL_SNAN` added to `<float.h>`. We have equivalents in `std::numeric_limits` already, but it seems harmless to add them. SG6 agreed.

Additions to `<fenv.h>`: The `femode_t` type and `FE_DFL_MODE` macro. New rounding direction, `FE_TONEARESTFROMZERO`. `FENV_ROUND` pragma. The `fesetexcept` and `fetestexceptflag` functions. The `fegetmode` and `fesetmode` functions. See related papers P3479R0 (Enabling C pragma support in C++) and P2746R6 (Deprecate and Replace Fenv Rounding Modes). Guidance from SG6 in Wrocław was to propose these in a separate paper.

Additions to `<math.h>`: Decimal floating-point types (optional in C). New functions `fromfp`, `ufromfp`, `fromfpx`, `fromupx`, and the math rounding directions, `FP_INT_UPWARD` etc. New macros `FP_FAST_FMA`, `FP_FAST_FMAF`, and `FP_FAST_FMAL`. 18 new macros, `FP_FAST_FADD` etc. Guidance from SG6 in Wrocław was to propose these in a separate paper. The new `nextup` and `nextdown` functions are included here.

Also in <math.h>: New iscanonical macro and core language concept of canonical representations and non-canonical representations in floating-point types. These are primarily needed for decimal floating-point types, which we don't have. No need to add these to <cmath> at this time. SG6 agreed.

New header <stdbit.h> with overlapping functionality to the C++ header <bit>. LEWG had consensus for adding <stdbit.h> with the content re-specified using C++ features, but with the same names as C uses. That is not part of this rebasing proposal, but was proposed by P3370R0 and approved in Wrocław. There was no consensus to add a <cstdbit> to C++. I am very strongly against adding such a header, because code that needs to be compatible with C should use <stdbit.h> and code that doesn't need to be compatible with C should use <bit>. There is no reason for <cstdbit> to exist.

New header <stdckdint.h> with functions for checking for overflow in addition, subtraction and multiplication. C++ has no equivalent currently, but we probably don't want type-generic macros like C has. The APIs would be better as templates with clear *Mandates*: requirements for suitable integer types. LEWG had consensus for adding <stdckdint.h> with the content re-specified using C++ features. That is not part of this rebasing proposal, but was proposed by P3370R0.

LEWG discussed this at a 2024-07-30 telecon and took some polls, with consensus to add the new functions to <string.h>, <time.h> and <stdlib.h>, and to include the new %OB and %Ob formats for strftime. The new functions are shown in the proposed wording below. There's no change shown for strftime, because it happens automatically by making C23 our reference.

In C17 passing a size of zero to realloc was deprecated and had implementation defined behaviour, see WG14 DR 400. In C23 it has undefined behaviour, see N2464. This doesn't result in a change to the declaration, but there is a change in the function's contract, making previously valid C++ code now undefined. This seems like a bad direction given our increased focus on safety and reducing sources of undefined behaviour in the C++ standard. C++ could choose to define that behaviour, or to say it's unspecified rather than undefined. But it's difficult to require that when realloc is typically provided by the operating system's C library, not by the C++ implementation. Individual implementations might continue to support it with their historical semantics (e.g. POSIX.1-2024 defines it), but it's difficult for the C++ standard to require or specify any particular semantics. We could say it's unspecified, which would work if we can be sure that all C++ implementations are able to provide a realloc which either defines it as described by POSIX or does something else with well-defined behaviour - just as long as it's not undefined. Or, we could say it's erroneous. This still requires the implementation to have some well-defined behaviour if it doesn't diagnose it as an error (e.g. trapping, reporting a UBsan/Asan error, or setting errno as POSIX does). The wording below says it has erroneous behavior, with implementation-defined effects that permit the POSIX behaviour.

Future Directions

The omitted floating-point features from <float.h> and <math.h> might need to be added later.

For , we might want to consider putting 'using std::unreachable;' in <stddef.h> for compatibility with C hearts which include stddef.h and then expect to be able to use the C unreachable macro (and then maybe making std::unreachable available via <cstddef> for consistency?).

Wording

The changes shown are relative to N5001, Working Draft (2024-12-17).

All dated references to ISO/IEC 9899:2018 should change. This is done by updating the `\IsoC` LaTeX macro in one place, but all affected text is shown below so the changes can be reviewed for correctness. N.B. The C23 standard was published by ISO with a publication date of 2024.

Update 1.2 [intro.refs] p1.3:

> (1.3) — ISO/IEC 9899:~~2018~~2024, Programming languages — C

Update 3.8 [defns.c.lib]

> **C standard library**
> library described in ISO/IEC 9899:~~2018~~2024, Clause 7
> [*Note 1 to entry*: With the qualifications noted in Clause 17 through Clause 33 and in C.8, the C standard library is a subset of the C++ standard library. — *end note*]

Update 16.2 [library.c] p3:

> A call to a C standard library function is a non-constant library call (3.34) if it raises a floating-point exception other than `FE_INEXACT`. The semantics of a call to a C standard library function evaluated as a core constant expression are those specified in ISO/IEC 9899:~~2018~~2024, Annex F[133] to the extent applicable to the floating-point types (6.8.2) that are parameter types of the called function.
>
> 133) See also ISO/IEC 9899:~~2018~~2024, 7.6.
> [*Drafting note*: This subclause is “Floating-point environment <fenv.h>” and is still 7.6 in C23.]

Update 16.4.2.3 [headers] p10:

> ISO/IEC 9899:~~2018~~2024, Annex K describes a large number of functions, with associated types and macros, which “promote safer, more secure programming” than many of the traditional C library functions. The names of the functions have a suffix of `_s`; most of them provide the same service as the C library function with the unsuffixed name, but generally take an additional argument whose value is the size of the result array. If any C++ header is included, it is implementation-defined whether any of these names is declared in the global namespace. (None of them is declared in namespace `std`.)

Also in 16.4.2.3 [headers], update the caption of Table 26 [tab:c.annex.k.names]:

> **Table 26 — Names from ISO/IEC 9899:~~2018~~2024, Annex K [tab:c.annex.k.names]**

Update the footnote in 16.4.3.3 [using.linkage]

> Whether a name from the C standard library declared with external linkage has extern "C" or extern "C++" linkage is implementation-defined. It is recommended that an implementation use extern "C++" linkage for this purpose.[152]

152) The only reliable way to declare an object or function signature from the C standard library is by including the header that declares it, notwithstanding the latitude granted in ISO/IEC 9899:~~2018~~2024, 7.1.4.
[*Drafting note*: This subclause is “Use of library functions” and is still 7.1.4 in C23.]

Update 17.2.1 [cstddef.syn] p1:

The contents and meaning of the header `<cstddef>` are the same as the C standard library header `<stddef.h>`, except that it does not declare the type `wchar_t`, that it does not define the macro `unreachable`, that it also declares the type `byte` and its associated operations (17.2.5), and as noted in 17.2.3 and 17.2.4.
[*Drafting note*: 17.2.3 describes how `nullptr_t` is defined in C++, overriding how C defines its version.]
See also: ISO/IEC 9899:~~2018~~2024, 7.~~19~~22
[*Drafting note*: This subclause is “Common definitions <stddef.h>”, which is 7.22 in C23.]
[*Drafting note*: The omission of __STDC_VERSION_STDDEF_H__ is intentional.]

Update 17.2.2 [cstdlib.syn]:

```
// 20.2.12, 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 free_sized(void* ptr, size_t size);
void free_aligned_sized(void* ptr, size_t alignment, size_t size);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
size_t memalignment(const void* p);     // freestanding

double atof(const char* nptr);
int atoi(const char* nptr);
long int atol(const char* nptr);
long long int atoll(const char* nptr);
double strtod(const char* nptr, char** endptr);
int strfromd(char* s, size_t n, const char* format, double fp);
int strfromf(char* s, size_t n, const char* format, float fp);
int strfroml(char* s, size_t n, const char* format, long double fp);
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);
```

[...]

```
// 26.13, C standard library algorithms
void* bsearch(const void* key, ~~const~~ void* base, size_t nmemb, size_t size, //
freestanding
              c-compare-pred* compar);
void* bsearch(const void* key, ~~const~~ void* base, size_t nmemb, size_t size, //
freestanding
              compare-pred* compar);
const void* bsearch(const void* key, const void* base, size_t nmemb, // freestanding
                    size_t size, c-compare-pred* compar);
```

```
const void* bsearch(const void* key, const void* base, size_t nmemb, // freestanding
                    size_t size, compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar);
```

[...]

The contents and meaning of the header <cstdlib> are the same as the C standard library header <stdlib.h>, except that it does not declare ~~the~~ types wchar_t or once_flag, and does not declare the function call_once, and except as noted in 17.2.3, 17.2.4, 17.5, 20.2.12, 28.7.5, 26.13, 29.5.10, and 29.7.2.
[*Note 1* : Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — *end note*]
See also: ISO/IEC 9899:~~2018~~2024, 7.~~22~~24
[*Drafting note*: This subclause is “General utilities <stdlib.h>” , which is 7.24 in C23.]
[*Drafting note*: The omission of __STDC_VERSION_STDLIB_H__ is intentional.]

Update 17.2.3 [support.types.nullptr] p2:

-1- The type nullptr_t is a synonym for the type of a nullptr expression, and it has the characteristics described in 6.8.2 and 7.3.12.
[*Note 1*: Although nullptr’s address cannot be taken, the address of another nullptr_t object that is an lvalue can be taken. — *end note*]

-2- The macro NULL is an implementation-defined null pointer constant.[161]
See also: ISO/IEC 9899:~~2018~~2024, 7.~~19~~22
[*Drafting note*: This subclause is “Common definitions <stddef.h>”, which is 7.22 in C23.]

Update 17.2.4 [support.types.layout] p5:

The type max_align_t is a trivial standard-layout type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context (6.7.6).
See also: ISO/IEC 9899:~~2018~~2024, 7.~~19~~22
[*Drafting note*: This subclause is “Common definitions <stddef.h>”, which is 7.22 in C23.]

Update 17.3.6 [climits.syn] p1:

```
// all freestanding
#define BOOL_WIDTH see below
#define CHAR_BIT see below
#define CHAR_WIDTH see below
#define SCHAR_WIDTH see below
#define UCHAR_WIDTH see below
#define USHRT_WIDTH see below
#define SHRT_WIDTH see below
#define UINT_WIDTH see below
#define INT_WIDTH see below
#define ULONG_WIDTH see below
#define LONG_WIDTH see below
#define ULLONG_WIDTH see below
#define LLONG_WIDTH see below
#define SCHAR_MIN see below
#define SCHAR_MAX see below
#define UCHAR_MAX see below
```

```
#define CHAR_MIN see below
#define CHAR_MAX see below
#define MB_LEN_MAX see below
#define SHRT_MIN see below
#define SHRT_MAX see below
#define USHRT_MAX see below
#define INT_MIN see below
#define INT_MAX see below
#define UINT_MAX see below
#define LONG_MIN see below
#define LONG_MAX see below
#define ULONG_MAX see below
#define LLONG_MIN see below
#define LLONG_MAX see below
#define ULLONG_MAX see below
```

The header <climits> defines all macros the same as the C standard library header <limits.h>, except that it does not define the macro BITINT_MAXWIDTH.
[*Note 1*: Except for the WIDTH macros, CHAR_BIT, and MB_LEN_MAX, a macro referring to an integer type T defines a constant whose type is the promoted type of T (7.3.7). — *end note*]
See also: ISO/IEC 9899:~~2018~~2024, 5.2.4.2.1
[*Drafting note*: This subclause is “Sizes of integer types <limits.h>” and has changed name to “Characteristics of integer types <limits.h>” but is still 5.2.4.2.1 in C23.]
[*Drafting note*: The omission of __STDC_VERSION_LIMITS_H__ is intentional.]

Update 17.3.7 [cfloat.syn] p1:

```
// all freestanding
#define __STDC_VERSION_FLOAT_H__ 202311L

#define FLT_ROUNDS see below
#define FLT_EVAL_METHOD see below
~~#define FLT_HAS_SUBNORM see below~~
~~#define DBL_HAS_SUBNORM see below~~
~~#define LDBL_HAS_SUBNORM see below~~
#define FLT_RADIX see below
#define INFINITY see below
#define NAN see below
#define FLT_SNAN see below
#define DBL_SNAN see below
#define LDBL_SNAN see below
#define FLT_MANT_DIG see below
#define DBL_MANT_DIG see below
#define LDBL_MANT_DIG see below
#define FLT_DECIMAL_DIG see below
#define DBL_DECIMAL_DIG see below
#define LDBL_DECIMAL_DIG see below
~~#define DECIMAL_DIG see below~~
#define FLT_DIG see below
#define DBL_DIG see below
#define LDBL_DIG see below
…
```

The header <cfloat> defines all macros the same as the C standard library header <float.h>.
[*Drafting note*: See Annex D entry for DECIMAL_DIG being deprecated, which is “the same as” <float.h>.]
See also: ISO/IEC 9899:~~2018~~2024, 5.2.4.2.2

[*Drafting note*: This subclause is “Characteristics of floating types <float.h>” and is still 5.2.4.2.2 in C23.]

Update 17.4.1 [cstdint.syn] p1:

```
  using uintptr_t = unsigned integer type; // optional
}

#define __STDC_VERSION_STDINT_H__ 202311L

#define INTN_MIN see below
#define INTN_MAX see below
#define UINTN_MAX see below
#define INTN_WIDTH see below
#define UINTN_WIDTH see below

#define INT_FASTN_MIN see below
#define INT_FASTN_MAX see below
#define UINT_FASTN_MAX see below
#define INT_FASTN_WIDTH see below
#define UINT_FASTN_WIDTH see below

#define INT_LEASTN_MIN see below
#define INT_LEASTN_MAX see below
#define UINT_LEASTN_MAX see below
#define INT_LEASTN_WIDTH see below
#define UINT_LEASTN_WIDTH see below

#define INTMAX_MIN see below
#define INTMAX_MAX see below
#define UINTMAX_MAX see below
#define INTMAX_WIDTH see below
#define UINTMAX_WIDTH see below

#define INTPTR_MIN see below              // optional
#define INTPTR_MAX see below              // optional
#define UINTPTR_MAX see below             // optional
#define INTPTR_WIDTH see below            // optional
#define UINTPTR_WIDTH see below           // optional

#define PTRDIFF_MIN see below
#define PTRDIFF_MAX see below
#define PTRDIFF_WIDTH see below
#define SIZE_MAX see below
#define SIZE_WIDTH see below

#define SIG_ATOMIC_MIN see below
#define SIG_ATOMIC_MAX see below
#define SIG_ATOMIC_WIDTH see below

#define WCHAR_MIN see below
#define WCHAR_MAX see below
#define WCHAR_WIDTH see below

#define WINT_MIN see below
#define WINT_MAX see below
#define WINT_WIDTH see below
```

[…]

The header defines all types and macros the same as the C standard library header `<stdint.h>`.
The types denoted by `intmax_t` and `uintmax_t` are not required to be able to represent all values of extended integer types wider than `long long` and `unsigned long long`, respectively.
[*Drafting note*: This text was added to 31.13.2 [cinttypes.syn] by LWG 3028, but <cinttypes> doesn't define these types, so this is the correct place to say it.]
See also: ISO/IEC 9899:~~2018~~2024, 7.~~20~~22
[*Drafting note*: This subclause is "Integer types <stdint.h>" and is 7.22 in C23.]

Update 17.5 [support.start.term] p14:

Remarks: The function `quick_exit` is signal-safe (17.13.5) when the functions registered with `at_quick_exit` are.

See also: ISO/IEC 9899:~~2018~~2024, 7.~~22~~24.4
[*Drafting note*: This subclause is "Communication with the environment" and is 7.24.4 in C23.]

Update 17.13.2 [cstdarg.syn] p1:

```
// all freestanding
#define __STDC_VERSION_STDARG_H__ 202311L

namespace std {
  using va_list = see below;
}
#define va_arg(V, P) see below
#define va_copy(VDST, VSRC) see below
#define va_end(V) see below
#define va_start(V, ~~P~~ ...) see below
```

The contents of the header `<cstdarg>` are the same as the C standard library header `<stdarg.h>`, with the following changes:
(1.1) — In lieu of the default argument promotions specified in ISO C 6.5.2.2, the definition in 7.6.1.3 applies.
[*Drafting note*: This subclause is "Function calls" and is still 6.5.2.2 in C23.]
(1.2) —~~ The restrictions that ISO C places on the second parameter to the `va_start` macro in header `<stdarg.h>` are different in this document. The parameter `parmN` is the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`).[196] If the parameter `parmN` is a pack expansion (13.7.4) or an entity resulting from a lambda capture (7.5.5), the program is ill-formed, no diagnostic required. If the parameter `parmN` is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is Undefined.~~ The preprocessing tokens comprising the second and subsequent arguments to va_start (if any) are discarded.
[*Note ?*: `va_start` accepts a second argument for compatibility with prior revisions of C++. – *end note*]
See also: ISO/IEC 9899:~~2018~~2024, 7.16~~.1.1~~
[*Drafting note*: This subclause is "The `va_arg` macro" and is still 7.16.1.1 C23, but it seems that 17.6 "Variable arguments <stdarg.h>" would be more appropriate here.]

Update 17.13.3 [csetjmp.syn]:

```
#define __STDC_VERSION_SETJMP_H__ 202311L
```

```
namespace std {
  using jmp_buf = see below;
  [[noreturn]] void longjmp(jmp_buf env, int val);
}

#define setjmp(env) see below
```

1 The contents of the header <csetjmp> are the same as the C standard library header <setjmp.h>.

2 The function signature longjmp(jmp_buf jbuf, int val) has more restricted behavior in this document. A setjmp/longjmp call pair has undefined behavior if replacing the setjmp and longjmp by catch and throw would invoke any non-trivial destructors for any objects with automatic storage duration. A call to setjmp or longjmp has undefined behavior if invoked in a suspension context of a coroutine (7.6.2.4).

See also: ISO/IEC 9899:~~2018~~2024, 7.13

[*Drafting note*: This subclause is "Non-local jumps <setjmp.h>" and is still 7.13 in C23.]

Update 17.13.4 [csignal.syn] p4:

The function signal is signal-safe if it is invoked with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

See also: ISO/IEC 9899:~~2018~~2024, 7.14

[*Drafting note*: This subclause is "Signal handling <signal.h>" and is still 7.14 in C23.]

Change the example in 17.14.1 [support.c.headers.general] p1:

For compatibility with the C standard library, the C++ standard library provides the C headers shown in Table 44. The intended use of these headers is for interoperability only. It is possible that C++ source files need to include one of these headers in order to be valid ISO C. Source files that are not intended to also be valid ISO C should not use any of the C headers.

[*Note 1*: The C headers either have no effect, such as <stdbool.h> (17.14.5) and <stdalign.h> (17.14.4), or otherwise the corresponding header of the form <*cname*> provides the same facilities and assuredly defines them in namespace std. — *end note*]

[*Example 1*: The following source file is both valid C++ and valid C. Viewed as C++, it declares a function with C language linkage; viewed as C it simply declares a function (and provides a prototype).

```
~~#include <stdbool.h>    // for bool in C, no effect in C++~~
#include <uchar.h>      // for char8_t in C, not necessary in C++
#include <stddef.h>     // for size_t

#ifdef __cplusplus      // see 15.11
extern "C"              // see 9.11
#endif
void f(~~bool b~~ char8_t s[], size_t n);
```

– *end example*]

Update 17.14.4 [stdalign.h.syn], C23 no longer defines this macro, so there is no difference:

17.14.4 Header <stdalign.h> synopsis [stdalign.h.syn]

The contents of the C++ header <stdalign.h> are the same as the C standard library header <stdalign.h>~~, with the following changes: The header <stdalign.h> does not define a macro named alignas~~.

See also: ISO/IEC 9899:~~2018~~2024, 7.15

[*Drafting note*: This subclause is “Alignment <stdalign.h>” and is still 7.15 in C23.]

Update 17.14.5 [stdbool.h.syn], C23 no longer defines these macros, so there is no difference:

17.14.5 Header <stdbool.h> synopsis [stdbool.h.syn]

The contents of the C++ header `<stdbool.h>` are the same as the C standard library header `<stdbool.h>`~~; with the following changes: The header `<stdbool.h>` does not define macros named `bool`, `true`, or `false`~~.

See also: ISO/IEC 9899:~~2018~~2024, 7.~~18~~19

[*Drafting note*: This subclause is “Boolean type and values <stdbool.h>” and is 7.19 in C23.]

Update 20.2.12 [c.malloc]:

1 [*Note 1*: The header <cstdlib> (17.2.2) declares the functions described in this subclause. — end note]

```
void* aligned_alloc(size_t alignment, size_t size);
void* calloc(size_t nmemb, size_t size);
void* malloc(size_t size);
~~void* realloc(void* ptr, size_t size);~~
```

2 *Effects*: These functions have the semantics specified in the C standard library.

3 *Remarks*: These functions do not attempt to allocate storage by calling `::operator new()` (17.6.3). These functions implicitly create objects (6.7.2) in the returned region of storage and return a pointer to a suitable created object. In the case of calloc ~~and realloc~~, the objects are created before the storage is zeroed ~~or copied, respectively~~.

```
void* realloc(void* ptr, size_t size);
```

4 *Preconditions*: `free(ptr)` has well-defined behavior.

5 *Effects*: If `ptr` is not null and `size` is zero, the behavior is erroneous and the effects are implementation-defined. Otherwise, this function has the semantics specified in the C standard library.

6 *Remarks*: This function does not attempt to allocate storage by calling `::operator new()` (17.6.3). When a non-null pointer is returned, this function implicitly creates objects (6.7.2) in the returned region of storage and returns a pointer to a suitable created object. The objects are created before the storage is copied.

Update 26.13 [alg.c.library]:

1 [*Note 1*: The header `<cstdlib>` (17.2.2) declares the functions described in this subclause. — *end note*]

```
void* bsearch(const void* key, ~~const~~ void* base, size_t nmemb, size_t size,
              c-compare-pred* compar);
void* bsearch(const void* key, ~~const~~ void* base, size_t nmemb, size_t size,
              compare-pred* compar);
const void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                    c-compare-pred* compar);
const void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                    compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar);
void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar);
```

2 *Preconditions*: For `qsort`, the objects in the array pointed to by `base` are of trivially copyable type.

3 *Effects*: These functions have the semantics specified in the C standard library.
4 *Throws*: Any exception thrown by `compar` (16.4.6.13).
See also: ISO/IEC 9899:~~2018~~2024, 7.~~22~~24.5

Update 27.5.1 [cstring.syn]:

```
#define __STDC_VERSION_STRING_H__ 202311L

namespace std {
  using size_t = see 17.2.4;
  void* memcpy(void* s1, const void* s2, size_t n);  // freestanding
  void* memccpy(void* s1, const void* s2, int c, 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* strdup(const char* s);
  char* strndup(const char* s, size_t size);
  char* strcat(char* s1, const char* s2);            // freestanding
  char* strncat(char* s1, const char* s2, size_t n); // freestanding
  ...
  void* memset(void* s, int c, size_t n);            // freestanding
  void* memset_explicit(void* s, int c, size_t n);   // freestanding
  char* strerror(int errnum);
  ...
```

See also: ISO/IEC 9899:~~2018~~2024, 7.~~24~~26
[*Drafting note*: This subclause is “String handling <string.h>” and is 7.26 in C23.]

Update 28.7.1 [cctype.syn]:

The contents and meaning of the header `<cctype>` are the same as the C standard library header `<ctype.h>`.
See also: ISO/IEC 9899:~~2018~~2024, 7.4
[*Drafting note*: This subclause is “Character handling <ctype.h>” and is still 7.4 in C23.]

Update 28.7.2 [cwctype.syn]:

The contents and meaning of the header `<cwctype>` are the same as the C standard library header `<wctype.h>`.
See also: ISO/IEC 9899:~~2018~~2024, 7.4
[*Drafting note*: This subclause is “Character handling <ctype.h>” and is still 7.4 in C23.]

Update 28.7.3 [cwchar.syn]:

```
#define __STDC_VERSION_WCHAR_H__ 202311L
```

[...]

```
#define NULL see 17.2.3         // freestanding
#define WCHAR_MAX see below     // freestanding
#define WCHAR_MIN see below     // freestanding
#define WCHAR_WIDTH see below   // freestanding
#define WEOF see below          // freestanding
```

1 The contents and meaning of the header `<cwchar>` are the same as the C standard library header `<wchar.h>`, except that it does not declare a type `wchar_t`.
2 [*Note 1*: The functions `wcschr`, `wcspbrk`, `wcsrchr`, `wcsstr`, and `wmemchr` have different signatures in this document, but they have the same behavior as in the C standard library (16.2). — *end note*]
See also: ISO/IEC 9899:~~2018~~2024, 7.~~29~~31
[*Drafting note*: This subclause is "Extended multibyte and wide character utilities <wchar.h>" and is 7.31 in C23.]

Update 28.7.4 [cuchar.syn]:

```
#define __STDC_VERSION_UCHAR_H__ 202311L

namespace std {
  using mbstat_t = see below;
```
[...]

The contents and meaning of the header `<cuchar>` are the same as the C standard library header `<uchar.h>`, except that it ~~declares the additional `mbrtoc8` and `c8rtomb` functions and~~ does not declare types `char8_t`, `char16_t`, ~~n~~or `char32_t`.

See also: ISO/IEC 9899:~~2018~~2024, 7.~~28~~30
[*Drafting note*: This subclause is "Unicode utilities <uchar.h>" and is 7.30 in C23.]

Update the list of declarations preceding 28.7.5 [c.mb.wcs] p5:

```
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 mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
size_t c8rtomb(char* s, char8_t c8, mbstate_t* ps);
size_t mbrtoc16(char16_t* pc16, const char* s, size_t n, mbstate_t* ps);
size_t c16rtomb(char* s, char16_t c16, mbstate_t* ps);
size_t mbrtoc32(char32_t* pc32, const char* s, size_t n, mbstate_t* ps);
size_t c32rtomb(char* s, char32_t c32, 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);
```
[*Drafting note*: This adds previously missing functions which are already present in the header synopsis in [cuchar.syn], and moves `mbrtoc8` and `c8rtomb` here now that they're in C23 and can be specified by reference to C23.]

5 *Effects*: These functions have the semantics specified in the C standard library.
6 *Remarks*: …
See also: ISO/IEC 9899:~~2018~~2024, 7.30.1, 7.~~29~~31.6.3, 7.31.6.4
[*Drafting note*: This adds previously missing section references for declarations in <uchar.h> and <wchar.h>]

Remove 28.7.5 [c.mb.wcs] p7-p11:

~~`size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);`~~

~~*7 Effects: …*~~

~~*8 Returns: …*~~

~~`size_t c8rtomb(char* s, char8_t c8, mbstate_t* ps);`~~

~~*9 Effects: …*~~

~~*10 Returns: …*~~

~~*11 Remarks: …*~~

[*Drafting note*: These paragraphs describe functions added in C++20 that were subsequently added to C23, so are now specified in p5 and p6.]

Update 29.3.1 [cfenv.syn] synopsis before p1:

-1- The contents and meaning of the header `<cfenv>` are a subset of ~~are the same as~~ the C standard library header `<fenv.h>` and only the declarations shown in the synopsis above are present.
[*Note 1*: …]
See also: ISO/IEC 9899:~~2018~~2024, 7.6
[*Drafting note*: This subclause is “Floating-point environment <fenv.h>” and is still 7.6 in C23.]

TODO: can I say “above” above?

Update 29.7.1 [cmath.syn] synopsis before p1:

```
constexpr floating-point-type nexttoward(floating-point-type x, long double y);
constexpr float nexttowardf(float x, long double y);
constexpr long double nexttowardl(long double x, long double y);

constexpr floating-point-type nextup(floating-point-type x);
constexpr float nextupf(float x);
constexpr long double nextupl(long double x);

constexpr floating-point-type nextdown(floating-point-type x);
constexpr float nextdownf(float x);
constexpr long double nextdownl(long double x);

constexpr floating-point-type fdim(floating-point-type x, floating-point-type y);
constexpr float fdimf(float x, float y);
constexpr long double fdiml(long double x, long double y);
```

Update 29.7.1 [cmath.syn] p1:

The contents and meaning of the header `<cmath>` are a subset of ~~are the same as~~ the C standard library header `<math.h>` and only the declarations shown in the synopsis above are present, with the addition of a three-dimensional hypotenuse function (29.7.4), a linear interpolation function (28.7.4), and the mathematical special functions described in 29.7.6.
[*Note 1*: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — *end note*]

Update 30.15 [ctime.syn]:

```
#define __STDC_VERSION_TIME_H__ 202311L

#define NULL see 17.2.3
#define CLOCKS_PER_SEC see below
#define TIME_UTC see below
#define TIME_MONOTONIC see below            // optional
#define TIME_ACTIVE see below               // optional
#define TIME_THREAD_ACTIVE see below        // optional
```

```
namespace std {
  using size_t = see 17.2.4;
  using clock_t = see below;
  using time_t = see below;

  struct timespec;
  struct tm;

  clock_t clock();
  double difftime(time_t time1, time_t time0);
  time_t mktime(tm* timeptr);
  time_t timegm(tm* timeptr);
  time_t time(time_t* timer);
  int timespec_get(timespec* ts, int base);
  int timespec_getres(timespec* ts, int base);
  char* asctime(const tm* timeptr);
  char* ctime(const time_t* timer);
  tm* gmtime(const time_t* timer);
  tm* gmtime_r(const time_t* timer, tm* buf);
  tm* localtime(const time_t* timer);
  tm* localtime_r(const time_t* timer, tm* buf);
  size_t strftime(char* s, size_t maxsize, const char* format, const tm* timeptr);
}
```

1 The contents of the header <ctime> are the same as the C standard library header <time.h>.

2 The functions ~~asctime, ctime,~~ gmtime~~,~~ and localtime are not required to avoid data races ([res.on.data.races]).
See also: ISO/IEC 9899:~~2018~~2024, 7.~~27~~29

Update 31.13.1 [cstdio.syn]:

```
#define __STDC_VERSION_STDIO_H__ 202311L

namespace std {
  using size_t = see 17.2.4;
  using FILE = see below;
  using fpos_t = see below;
}
#define NULL see 17.2.3
#define _IOFBF see below
#define _IOLBF see below
#define _IONBF see below
#define BUFSIZ see below
#define EOF see below
#define FOPEN_MAX see below
#define FILENAME_MAX see below
#define __PRINTF_NAN_LEN_MAX see below
#define L_tmpnam see below
#define SEEK_CUR see below
#define SEEK_END see below
#define SEEK_SET see below
#define TMP_MAX see below
#define stderr see below
#define stdin see below
```

```
#define stdout see below
```

Update 31.13.2 [cinttypes.syn]:

```
#define __STDC_VERSION_INTTYPES_H__ 202311L

#define PRIdN see below
#define PRIiN see below
#define PRIoN see below
#define PRIuN see below
#define PRIxN see below
#define PRIXN see below
#define PRIbN see below
#define PRIBN see below
#define SCNdN see below
#define SCNiN see below
#define SCNoN see below
#define SCNuN see below
#define SCNxN see below
#define SCNbN see below
#define PRIdLEASTN see below
#define PRIiLEASTN see below
#define PRIoLEASTN see below
#define PRIuLEASTN see below
#define PRIxLEASTN see below
#define PRIXLEASTN see below
#define PRIbLEASTN see below
#define PRIBLEASTN see below
#define SCNdLEASTN see below
#define SCNiLEASTN see below
#define SCNoLEASTN see below
#define SCNuLEASTN see below
#define SCNxLEASTN see below
#define SCNbLEASTN see below
#define PRIdFASTN see below
#define PRIiFASTN see below
#define PRIoFASTN see below
#define PRIuFASTN see below
#define PRIxFASTN see below
#define PRIXFASTN see below
#define PRIbFASTN see below
#define PRIBFASTN see below
#define SCNdFASTN see below
#define SCNiFASTN see below
#define SCNoFASTN see below
#define SCNuFASTN see below
#define SCNxFASTN see below
#define SCNbFASTN see below
#define PRIdMAX see below
#define PRIiMAX see below
#define PRIoMAX see below
#define PRIuMAX see below
#define PRIxMAX see below
#define PRIXMAX see below
#define PRIbMAX see below
#define PRIBMAX see below
#define SCNdMAX see below
#define SCNiMAX see below
```

```
#define SCNoMAX see below
#define SCNuMAX see below
#define SCNxMAX see below
#define SCNbMAX see below
#define PRIdPTR see below
#define PRIiPTR see below
#define PRIoPTR see below
#define PRIuPTR see below
#define PRIxPTR see below
#define PRIXPTR see below
#define PRIbPTR see below
#define PRIBPTR see below
#define SCNdPTR see below
#define SCNiPTR see below
#define SCNoPTR see below
#define SCNuPTR see below
#define SCNxPTR see below
#define SCNbPTR see below
```

1 The contents and meaning of the header <cinttypes> are the same as the C standard library header <inttypes.h>, with the following changes:

(1.1) — The header <cinttypes> includes the header <cstdint> (17.4.1) instead of <stdint.h>, and

~~(1.2) — intmax_t and uintmax_t are not required to be able to represent all values of extended integer types wider than long long and unsigned long long, respectively, and~~

[*Drafting note*: This text is moved to 17.4.1 [cstdint.syn], see above.]

(1.3) — if and only if the type intmax_t designates an extended integer type (6.8.2), the following function signatures are added:

```
constexpr intmax_t abs(intmax_t);
constexpr imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures constexpr intmax_t imaxabs(intmax_t) and constexpr imaxdiv_t imaxdiv(intmax_t, intmax_t), respectively.

See also: ISO/IEC 9899:~~2018~~2024, 7.8

[*Drafting note*: This subclause is “Format conversion of integer types <inttypes.h>” and is still 7.8 in C23.]

2 Each of the PRI macros listed in this subclause is defined if and only if the implementation defines the corresponding *typedef-name* in 17.4.1. Each of the SCN macros listed in this subclause is defined if and only if the implementation defines the corresponding *typedef-name* in 17.4.1 and has a suitable fscanf length modifier for the type. Each of the PRIB macros listed in this subclause is defined if and only if fprintf supports the B conversion specifier.

Add a new subclause after C.1.5 [diff.cpp23.library],:

C.1.? Clause 20: Memory management library

Affected subclause: [c.malloc]
Change: Calling realloc with a non-null pointer and zero size has erroneous behavior.
Rationale: The C standard library does not define this behavior.
Effect on original feature: Valid C++ 2023 code that calls realloc with a non-null pointer and a size of zero is erroneous and may change behavior.

Update C.8.3.1 [diff.char16]:

C.8.3.1 Types `char8_t`, `char16_t`, and `char32_t` [diff.char16]

1 The types `char8_t`, `char16_t`, and `char32_t` are distinct types rather than typedefs to existing integral types. The tokens `char8_t`, `char16_t`, and `char32_t` are keywords in C++ ([lex.key]). They do not appear as macro or type names defined in `<cuchar>`.

Remove C.8.3.3 [diff.header.assert.h]:

~~1 The token `static_assert` is a keyword in C++. It does not appear as a macro name defined in `<cassert>` (19.3.2).~~

Remove C.8.3.5 [diff.header.stdalign.h]:

~~1 The token `alignas` is a keyword in C++ (5.11), and is not introduced as a macro by `<stdalign.h>` (17.14.4).~~

Remove C.8.3.6 [diff.header.stdbool.h]:

~~1 The tokens `bool`, `true`, and `false` are keywords in C++ (5.11), and are not introduced as macros by `<stdbool.h>` (17.14.5).~~

Update D.11 [depr.c.macros]:

D.11 Deprecated C macros [depr.c.macros]

1 The header `<cfloat>` has the following macros:

```
#define FLT_HAS_SUBNORM see below
#define DBL_HAS_SUBNORM see below
#define LDBL_HAS_SUBNORM see below
#define DECIMAL_DIG see below
```

The header defines these macros the same as the C standard library header `<float.h>`.
See also: ISO/IEC 9899:2024, 5.2.4.2.2, 7.33.5

[*Drafting note*: C23 5.2.4.2.2 <float.h> has a cross-reference to 7.33.8 for DECIMAL_DIG being obsolescent, but that's incorrect and should be 7.33.5 as shown here.]

2 In addition to being available via inclusion of the `<cfloat>` header, the macros `INFINITY` and `NAN` are available when `<cmath>` is included.
See also: ISO/IEC 9899:2024, 7.12

~~1 The header `<stdalign.h>` has the following macros:~~

```
~~#define __alignas_is_defined 1~~
~~#define __alignof_is_defined 1~~
```

[*Drafting note*: The stdalign macros are removed entirely from C23, without deprecation.]

2 The header `<stdbool.h>` has the following macro:

```
#define __bool_true_false_are_defined 1
```

See also: ISO/IEC 9899:2024, 7.19

[*Drafting note*: This macro is still present in C23, but marked obsolete.]

Add a new subclause after D.20 [depr.format]:

D.? Deprecated [depr.ctime]

D.?.? Header <ctime> synopsis [depr.ctime.syn]

1 The header `<ctime>` ([ctime.syn]) has the following additions:

```
char* asctime(const tm* timeptr);
char* ctime(const time_t* timer);
```

The functions `asctime` and `ctime` are not required to avoid data races ([res.on.data.races]).

See also: ISO/IEC 9899:2024, 7.29

Acknowledgements

Thanks to Tom Honermann, Matthias Kretz, Joseph Myers, and Jens Gustedt for improvements and suggestions.