constexpr for <cmath> and <cstdlib>

Document: D0533R9
Date: November 12, 2021
Project: Programming Language C++, Library Working Group
Audience: LWG & CWG
Reply to: Edward J. Rosten (erosten@snap.com) / Oliver J. Rosten (oliver.rosten@gmail.com)

Abstract

We propose simple criteria for selecting functions in <cmath> which should be declared constexpr. There is a small degree of overlap with <cstdlib>. The aim is to transparently select a sufficiently large portion of <cmath> in order to be useful but without placing too much burden on compiler vendors.

CONTENTS

I. REVISION HISTORY

R1 Includes discussion of rounding mode and future directions.

R2 More stable tags utilized.

R3 Lifted the unnecessary restriction not to include functions which modify an argument with external visibility. Proposed a modification to [library.c].

R4 Leverage std::is_constant_evaluated() as a mechanism to allow implementation of this proposal as a pure library extension—see the final paragraph of III A and the first of V. Clarifications to the 'state of the art'.

R5 Highlighting of proposed changes improved plus minor tweaks to ensure consistency with the latest draft of the standard.

R6 Transfer proposed wording from [expr.const] to [library.c]; propose the addition of a feature-test macro; further improvements to highlighting; Update the location of the feature-test macro to [version.syn].

R7 Rebase to N4878 and add a note that lerp, which is declared constexpr, has been added to <cmath>; minor tweaks to wording following review by LWG.

R8 Improve and correct some examples; correct the first criterion for selecting functions. Provide a better critique of the State of the Art. Amend the wording in line with the spirit of the 2021-05-17 LEWG Telecon.

R9 Refine the wording in line with the spirit of the 2021-10-15 LWG Telecon—with input from the 2021-11-12 CWG Telecon—facilitated by adding a new definition [defns.nonconst.libcall] and normative wording to [expr.const]. The first of the original changes to [library.c] has been removed following the 2021-10-22 LWG Telecon.

II. INTRODUCTION

This paper seeks to rectify the almost complete absence of constexpr in <cmath> (and also in <cstdlib>), so as to broaden the range of numeric computations that can be performed using standard library facilities. While in principle almost every function in <cmath> could be declared constexpr, we strike a balance between coverage and onus on compiler/library vendors.

III. MOTIVATION & SCOPE

The introduction of constexpr has facilitated intuitive compile-time programming. However, in <cmath>, only the recently added lerp is currently declared constexpr, thereby artificially restricting what can be done at compile-time within the standard library. Furthermore, it is incongruous that, while std::chrono::abs can

be used in a `constexpr` context, the same is not true for `std::abs`. Nevertheless, from casual inspection of `<cmath>`, it may not be immediately obvious precisely which set of functions should be declared `constexpr`. In this paper, we seek an organizing principle which selects functions which are in a sense no more complicated than the elementary arithmetic operations $(+, -, \times, /)$. This is justified since the latter already support `constexpr`.

Indeed, two subtleties can be resolved by appealing to the fact that they must be dealt with in implementing `constexpr` for the arithmetic operators. In particular, various functions in `<cmath>` may set global flags and/or depend on the rounding mode. These issues are discussed in the next two subsections. Following this, a justification is given for declaring functions in `<cmath>` which modify an argument with external visibility to be `constexpr`. These considerations lead to a concrete statement of the conditions under which a function should be declared `constexpr`.

A. Global Flags

Under certain conditions, various functions in `<cmath>` may set global flags. Specifically, `errno` may be set and/or the various floating-point exception flags, `FE_DIVBYZERO`, `FE_INVALID`, `FE_OVERFLOW`, `FE_UNDERFLOW` and `FE_INEXACT` may be raised. The strategy adopted is that if this occurs during evaluation of a mathematical function in a `constexpr` context then compilation fails, except in the situation where only `FE_INEXACT` is raised. The question as to when these flags should be raised is answered by deferring, wherever possible, to the C Standard's Annex F [n2176]. The latter applies, in its entirety, to floating-point models with signed zeros, extended with NaNs and signed infinities. Thus, whenever the floating-point model under consideration renders part of Annex F applicable, this is used to unambiguously specify the conditions under which floating-point exceptions are raised.

For example, consider `std::lround(double x)`, which rounds its argument to the nearest integer value. What should this do if its argument is NaN or $\pm\infty$? Without concrete guidance, implementors could reasonably adopt a variety of strategies. However, according to [n2176] `FE_INVALID` is raised in these cases and so that is the solution advocated by this paper.

It is important to note that the issue of raising exception flags in a `constexpr` context is nothing new: it is already faced by the standard arithmetic operators. Nevertheless, the latter are available for use in constant expressions. A concrete example is given by

```
std::numeric_limits<double>::max() * 2;
```

which raises both `FE_INVALID` and `FE_OVERFLOW`; indeed, GCC rejects this as a constant expression.

When not used in a `constexpr` context, the various global flags should be set as normal. This distinction between these two contexts previously implied that any implementation cannot be done as a pure library extension. However, the emergence of `std::is_constant_evaluated()` [P0595] may allow for this be circumvented. Either way, below we will introduce a criterion which restricts the proposed set of `constexpr` functions to those which are, in a sense, simple. Consequently, while there will be some burden on compiler vendors it is hoped to be acceptable.

B. Rounding Mode

Some of the functions in `<cmath>` depend on the rounding mode, which is something that may be changed at runtime. To facilitate the discussion, we wish to distinguish two situations, which we will call *weak*/*strong* rounding mode dependence.

Weak dependence is that already experienced by the arithmetic operators. For example, consider 1.0/3.0: the result depends on the rounding mode. We refer to this rounding mode dependence as weak since it is an artefact of the limited precision of floating-point numbers. However, it is perfectly legitimate to declare

```
constexpr double x{1.0/3.0}.
```
(3.1)

Therefore, when deciding which functions in `<cmath>` should be `constexpr`, we will not rule out functions with weak rounding mode dependence. As for (3.1), what result should we expect? According to [cfenv.syn] footnote 1, the result is implementation defined. However, this issue is currently under active discussion.

The key point for this paper is that, whatever decision is made, the approach can be consistently applied to those functions in `<cmath>` which we propose should be declared `constexpr`. It is worth noting that the number of functions in this proposal which are dependent on the rounding mode is rather small (see VI).

Having dealt with weak rounding mode dependence, now consider `float nearbyint(float x)`. This function rounds its argument to the nearest integer *taking account of the current rounding mode*. Thus, a change to the rounding mode can change the answer by unity. This dependence on the rounding mode is not an artefact of limited precision and hence we call it strong.

In this proposal, we chose to exclude functions with strong rounding mode dependence from being declared `constexpr`. This respects the fact that these functions are explicitly designed to depend on the runtime environment.

C. Arguments with External Visibility

At first sight, it may appear pointless to declare functions like

```
float frexp(float value, int* exp)
```

to be `constexpr` since such functions modify arguments with external visibility. However, declaring functions of this type `constexpr` means that they can be used in `constexpr` contexts. In other words this would allow functions such as

```
constexpr int foo(float x) {
  int a{};
  std::frexpr(x, &a);
  return a;
}
```

to be used to do things like

```
constexpr int i{foo(0.5f)}.
```

D. Conditions for `constexpr`

Taking into account the above consideration, we propose the following in order to put the application of `constexpr` on a rigorous footing:

Proposal. *A function in* `<cmath>` *shall be declared* `constexpr` *if and only if:*

1. *When taken to act on the set of rational numbers, or every subset thereof which is nowhere dense in the reals*[1]*, the function is closed (excluding division by, or logarithms*[2] *of, zero);*
2. *The function is not strongly dependent on the rounding mode.*

By means of a brief illustration, `abs` satisfies all three criteria; however, functions such as `exp`, `sqrt`, `cos`, `sin` fall foul of the first criterion and so are excluded as `constexpr` candidates. Finally, as discussed above, `nearbyint` fails the second criterion.

IV. STATE OF THE ART

Both GCC and clang already support `constexpr` within `<cmath>` to varying extents. Indeed, at least as early as GCC 5.3.0 almost all functions, besides the special functions, those taking a pointer argument (cf. III C) and those with an explicit dependence on the runtime rounding mode (III B) are declared `constexpr`. However, there are a handful of edge cases which GCC treats inconsistently, all involving either infinity or NaN. For brevity, the following shorthand will be employed:

```
constexpr auto inf{
    std::numeric_limits<double>::infinity()};
```

Now, consider, for example, `std::ldexp(inf, 1)`. Despite the fact that GCC neither sets errno nor raises any floating-point exception flags, it prohibits the use of this expression in a `constexpr` context. By contrast, GCC is perfectly happy for `std::logb(inf)` to be used in a `constexpr` context. A breakdown of GCC's current behaviour can be found here [GCC Behaviour].

Note that, aside from these edge cases, the only function for which the behaviour does not conform to that specified in this paper is `std::fmod` which does not seem useable in constant expressions. Indeed, it should be emphasised that, in almost all cases, GCC behaves exactly as one would want. Therefore, an implementation of the changes to the standard proposed in this paper is mostly available (indeed, in some regards the GCC implementation goes beyond what we propose since it declares additional functions `constexpr`). While clang does not go nearly as far as GCC, it does offer some functions as builtins and is able to use them to perform compile time computations, constant propagation and so on. It is therefore hoped that any burden on compiler vendors implicit in this proposal is acceptable.

V. IMPACT ON THE STANDARD

To facilitate the primary wording changes, a new definition [defns.nonconst.subexpr] is added to [intro.defs], together with normative wording in [expr.const]. It is anticipated that these will be broadly useful, beyond the scope of this paper. Following this, an extra statement is added to [library.c], indicating that if a (mathematical) function raises a floating-point flag other than `FE_INEXACT`, then it is a non-constant subexpression. Annex F of the C Standard is invoked to define precisely when floating-point exceptions should be raised, and the behaviour when NaNs and/or infinities are passed as arguments. Beyond this, a new feature-test macro is added to [support.limits.general]. The remaining changes amount to scattering `constexpr` throughout the existing headers `<cmath>` and `<cstdlib>`, according to the rules specified earlier.

With `std::is_constant_evaluated()` it may be possible to implement the desired behaviour as a pure library extension; previously, this was not the case.

In this proposal, we have chosen for the standard to remain silent on the issue of the interaction of rounding mode dependence with constant expressions. On the one hand, this is no worse than the current situation regarding the arithmetic operators. On the other, the active discussion about how to optimally resolve this matter suggests to us that the issue is better served by a separate proposal.

VI. DESIGN DECISIONS

There are several obvious candidates in `<cmath>` to which `constexpr` should be applied, such as `abs`, `floor`,

[1] This clause ensures that `std::nexttoward` and `std::nextafter` are admitted.

[2] `ilogb` and `logb` both involve the integral part of a logarithm.

`ceil`. But, beyond these, it is desirable to apply `constexpr` throughout `<cmath>` in a consistent manner. Ideally, one would like to achieve this via the application of one or more criteria rooted in mathematics. On the one hand, any such approach must select the basic arithmetic operations, $(+, -, \times, /)$, since these may already be used in a `constexpr` context. On the other, it should ideally encompass prior work on `complex`, since it has already been proposed that, in addition to the arithmetic operations, `complex::norm` and a few other functions be declared `constexpr` [P0415R0].

Mathematically, a field is closed under the elementary operations of addition and multiplication. Numeric types do not form a field; however, since the basic arithmetic operations are already declared `constexpr`, this suggests that it may be possible to utilize a field which captures enough of the properties of numeric types in order to be useful in formulating criteria for the application of `constexpr`. The set of rational numbers is the natural candidate since all valid values of numeric types are elements of this set and, moreover, the rationals close over $(+, -, \times, /)$ (with zero excluded for division).

The subtlety of global flags being set upon encountering floating-point exceptions presents a challenge. If all functions which can set such flags are excluded from the list to tag `constexpr`, then the remaining list is rather sparse. To achieve something more useful suggests expanding the set to include those functions which are 'simple enough'. These considerations lead to the first condition of the proposal. Tables II–V contain the functions in `<cmath>` satisfying this criterion and indicate whether or not they pass the second criterion as well.

To reduce space, the following convention is observed. The functions listed in [c.math] are divided into blocks of closely related functions such as those shown in table I. Note that while the first three functions form an overload set, while the fourth and fifth have differing names. When classifying those functions which satisfy the first criterion, we will present just the first function in each such block, with the understanding that the others are similar in this regard. Furthermore, we supply various comments in the third column of the tables, observing the following shorthands:

```
int ilogb(float arg)
int ilogb(double arg)
int ilogb(long double arg)
int ilogbf(float arg)
int ilogbl(long double arg)
```

TABLE I. Example of a family of functions which appear as a block in the standard.

1. G: May set global variable;
2. S: Depends strongly on rounding mode;
3. W: Depends weakly on the rounding mode;
4. w: Depends weakly on the rounding mode only if `FLT_RADIX` is not 2;
5. U: Depends weakly on the rounding mode only in the case of underflow.

If more than one of these applies, then this is indicated using |; for example, if a function may set a global variable and also depends strongly on the rounding mode, this would be indicated by G|S. Finally, implementation dependence is denoted by a $\star$ so that, for example, G$\star$ means that whether or not a global variable may be set depends on the implementation.

Function	Pass	Comment
`float frexp(float value, int* exp)`	Yes	w
`int ilogb(float arg)`	Yes	G
`float ldexp(float x, int exp)`	Yes	G|w
`float logb(float arg)`	Yes	G
`float modf(float value, float* iptr)`	Yes	
`float scalbn(float x, int n)`	Yes	G|U
`float scalbln(float x, long int n)`	Yes	G|U

TABLE II. Various functions declared in [cmath.syn] which close on the rationals.

Function	Pass
`int abs(int j)`	Yes
`float fabs(float x)`	Yes

TABLE III. Absolute values declared in [cmath.syn] which close on the rationals.

VII. FUTURE DIRECTIONS

Ultimately, it is desirable to follows GCC's lead and to declare almost all functions in `<cmath>` as `constexpr`. This will amount to removing the first criterion of our proposal which, particularly once the issue of the interaction of rounding mode with `constexpr` has been fully resolved, should hopefully be relatively uncontroversial.

ACKNOWLEDGMENTS

We would like to thank Daniel Krügler, Antony Polukhin and especially Walter E. Brown for encouragement and advice. Sincere thanks also to Richard Smith, Jens Maurer, Tim Song and Davis Herring for help with standardese and Geoffrey Romer, Hubert Tong and Jonathan Wakely for additional feedback and help.

REFERENCES

[n2176] ISO/IEC 9899:2018 Standard for Programming Languages — C

[P0595] Richard Smith, Andrew Sutton and Daveed Vandevoorde, `std::is_constant_evaluated()`.

[GCC Behaviour] `https://godbolt.org/z/dMGxh61xj`

[P0415R0] Antony Polukhin, Constexpr for std::complex.

[N4885] Thomas Köppe, ed., Working Draft, Standard for Programming Language C++.

Function	Pass	Comment
`float ceil(float x)`	Yes	G⋆
`float floor(float x)`	Yes	G⋆
`float nearbyint(float x)`	No	S
`float rint(float x)`	No	G|S
`long int lrint(float x)`	No	G|S
`long long int llrint(float x)`	No	G|S
`float round(float x)`	Yes	G
`float lround(float x)`	Yes	G
`float llround(float x)`	Yes	G
`float trunc(float x)`	Yes	G
`float fmod(float x, float y)`	Yes	G|W
`float remainder(float x, float y)`	Yes	G|W⋆
`float remquo(float x, float y, int* quo)`	Yes	G|W⋆
`float copysign(float x, float y)`	Yes	
`float nextafter(float x, float y)`	Yes	G
`float nexttoward(float x, long double y)`	Yes	G
`float fdim(float x, float y)`	Yes	G|U
`float fmax(float x, float y)`	Yes	
`float fmin(float x, float y)`	Yes	
`float fma(float x, float y, float z)`	Yes	G|W

TABLE IV. Additional functions declared in [cmath.syn] which close on the rationals.

Function	Pass	Comment
`int fpclassify(float x);`	Yes	
`int isfinite(float x)`	Yes	
`int isinf(float x)`	Yes	
`int isnan(float x)`	Yes	
`int isnormal(float x)`	Yes	
`int signbit(float x)`	Yes	
`int isgreater(float x, float y)`	Yes	
`int isgreaterequal(float x, float y)`	Yes	
`int isless(float x, float y)`	Yes	
`int islessequal(float x, float y)`	Yes	
`int islessgreater(float x, float y)`	Yes	
`int isunordered(float x, float y)`	Yes	

TABLE V. Comparison operators belonging to [cmath.syn] which close on the rationals.

VIII. PROPOSED WORDING

The following proposed changes refer to the Working Paper [N4885].

A. Modification to "Terms and definitions" [intro.defs]

3.35
multibyte character **[defns.multibyte]**
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment
[*Note 1 to entry: The extended character set is a superset of the basic character set (5.3). —end note*]

3.36
non-constant library call **[defns.nonconst.libcall]**
invocation of a library function that, as part of evaluating any expression E, prevents E from being a core constant expression

~~**3.36**~~ **3.37**
NTCTS **[defns.ntcts]**

B. Modification to "Constant expressions" [expr.const]

[5] An expression E is a core constant expression unless the evaluation of E, following the rules of the abstract machine ([intro.execution]), would evaluate one of the following:

...

(5.27) — an invocation of the `va_arg` macro ([cstdarg.syn]).

(5.28) — a non-constant library call ([defns.nonconst.libcall]).

C. Modification to "The C standard library" [library.c]

[3] A call to a C standard library function is a non-constant library call ([defns.nonconst.libcall]) 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 Annex F of the C standard [Footnote: see also ISO/IEC 9899:2018 section 7.6.] to the extent applicable to the floating-point types ([basic.fundamental]) that are parameter types of the called function. [Note: Annex F specifies the conditions under which floating-point exceptions are raised and the behavior when NaNs and/or infinities are passed as arguments.] [Note: Equivalently, a call to a C standard library function is a non-constant library call if `errno` is set when `math_errhandling & MATH_ERRNO` is `true`.]

D. Modifications to "Header <version> synposis" [version.syn]

A new row is to be added:

```
#define __cpp_lib_constexpr_cmath          20????L          // also in <cmath>, <cstdlib>
```

E. Modifications to "Header <cstdlib> synopsis" [cstdlib.syn]

```
namespace std{

...

//[c.math.abs], absolute values
constexpr int abs(int j);
constexpr long int abs(long int j);
constexpr long long int abs(long long int j);
constexpr float abs(float j);
constexpr double abs(double j);
constexpr long double abs(long double j);

constexpr long int labs(long int j);
constexpr long long int llabs(long long int j);

constexpr div_t div(int numer, int denom);
constexpr ldiv_t div(long int numer, long int denom); // see [library.c]
constexpr lldiv_t div(long long int numer, long long int denom); // see [library.c]
constexpr ldiv_t ldiv(long int numer, long int denom);
constexpr lldiv_t lldiv(long long int numer, long long int denom);
}
```

F. Modifications to "Header <cmath> synopsis" [cmath.syn]

```
...

namespace std{

...

float acos(float x); // see [library.c]
double acos(double x);
long double acos(long double x); // see [library.c]
float acosf(float x);
long double acosl(long double x);

...

constexpr float frexp(float value, int* exp); // see [library.c]
constexpr double frexp(double value, int* exp);
constexpr long double frexp(long double value, int* exp); // see [library.c]
constexpr float frexpf(float value, int* exp);
constexpr long double frexpl(long double value, int* exp);

constexpr int ilogb(float x); // see [library.c]
constexpr int ilogb(double x);
constexpr int ilogb(long double x); // see [library.c]
constexpr int ilogbf(float x);
constexpr int ilogbl(long double x);

constexpr float ldexp(float x, int exp); // see [library.c]
constexpr double ldexp(double x, int exp);
constexpr long double ldexp(long double x, int exp); // see [library.c]
constexpr float ldexpf(float x, int exp);
constexpr long double ldexpl(long double x, int exp);

float log(float x); // see [library.c]
double log(double x);
long double log(long double x); // see [library.c]
float logf(float x);
long double logl(long double x);

float log10(float x); // see [library.c]
double log10(double x);
long double log10(long double x); // see [library.c]
float log10f(float x);
long double log10l(long double x);

float log1p(float x); // see [library.c]
double log1p(double x);
long double log1p(long double x); // see [library.c]
float log1pf(float x);
long double log1pl(long double x);

float log2(float x); // see [library.c]
double log2(double x);
long double log2(long double x); // see [library.c]
float log2f(float x);
```

```
long double log2l(long double x);

constexpr float logb(float x); // see [library.c]
constexpr double logb(double x);
constexpr long double logb(long double x); // see [library.c]
constexpr float logbf(float x);
constexpr long double logbl(long double x);

constexpr float modf(float value, float* iptr); // see [library.c]
constexpr double modf(double value, double* iptr);
constexpr long double modf(long double value, long double* iptr); // see [library.c]
constexpr float modff(float value, float* iptr);
constexpr long double modfl(long double value, long double* iptr);

constexpr float scalbn(float x, int n); // see [library.c]
constexpr double scalbn(double x, int n);
constexpr long double scalbn(long double x, int n); // see [library.c]
constexpr float scalbnf(float x, int n);
constexpr long double scalbnl(long double x, int n);

constexpr float scalbln(float x, long int n); // see [library.c]
constexpr double scalbln(double x, long int n);
constexpr long double scalbln(long double x, long int n); // see [library.c]
constexpr float scalblnf(float x, long int n);
constexpr long double scalblnl(long double x, long int n);

float cbrt(float x); // see [library.c]
double cbrt(double x);
long double cbrt(long double x); // see [library.c]
float cbrtf(float x);
long double cbrtl(long double x);

// [c.math.abs], absolute values
constexpr int abs(int j);
constexpr long int abs(long int j);
constexpr long long int abs(long long int j);
constexpr float abs(float j);
constexpr double abs(double j);
constexpr long double abs(long double j);

constexpr float fabs(float x); // see [library.c]
constexpr double fabs(double x);
constexpr long double fabs(long double x); // see [library.c]
constexpr float fabsf(float x);
constexpr long double fabsl(long double x);

float hypot(float x, float y); // see [library.c]
double hypot(double x, double y);
long double hypot(double x, double y); // see [library.c]
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

```
// [c.math.hypot3], three-dimensional hypotenuse
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);

...

constexpr float ceil(float x); // see [library.c]
constexpr double ceil(double x);
constexpr long double ceil(long double x); // see [library.c]
constexpr float ceilf(float x);
constexpr long double ceill(long double x);

constexpr float floor(float x); // see [library.c]
constexpr double floor(double x);
constexpr long double floor(long double x); // see [library.c]
constexpr float floorf(float x);
constexpr long double floorl(long double x);

float nearbyint(float x); // see [library.c]
double nearbyint(double x);
long double nearbyint(long double x); // see [library.c]
float nearbyintf(float x);
long double nearbyintl(long double x);

float rint(float x); // see [library.c]
double rint(double x);
long double rint(long double x); // see [library.c]
float rintf(float x);
long double rintl(long double x);

long int lrint(float x); // see [library.c]
long int lrint(double x);
long int lrint(long double x); // see [library.c]
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(float x); // see [library.c]
long long int llrint(double x);
long long int llrint(long double x); // see [library.c]
long long int llrintf(float x);
long long int llrintl(long double x);

constexpr float round(float x); // see [library.c]
constexpr double round(double x);
constexpr long double round(long double x); // see [library.c]
constexpr float roundf(float x);
constexpr long double roundl(long double x);

constexpr long int lround(float x); // see [library.c]
constexpr long int lround(double x);
constexpr long int lround(long double x); // see [library.c]
constexpr long int lroundf(float x);
constexpr long int lroundl(long double x);
```

```
constexpr long long int llround(float x); // see [library.c]
constexpr long long int llround(double x);
constexpr long long int llround(long double x); // see [library.c]
constexpr long long int llroundf(float x);
constexpr long long int llroundl(long double x);

constexpr float trunc(float x); // see [library.c]
constexpr double trunc(double x);
constexpr long double trunc(long double x); // see [library.c]
constexpr float truncf(float x);
constexpr long double truncl(long double x);

constexpr float fmod(float x, float y); // see [library.c]
constexpr double fmod(double x, double y);
constexpr long double fmod(long double x, long double y); // see [library.c]
constexpr float fmodf(float x, float y);
constexpr long double fmodl(long double x, long double y);

constexpr float remainder(float x, float y); // see [library.c]
constexpr double remainder(double x, double y);
constexpr long double remainder(long double x, long double y); // see [library.c]
constexpr float remainderf(float x, float y);
constexpr long double remainderl(long double x, long double y);

constexpr float remquo(float x, float y, int* quo); // see [library.c]
constexpr double remquo(double x, double y, int* quo);
constexpr long double remquo(long double x, long double y, int* quo); // see [library.c]
constexpr float remquof(float x, float y, int* quo);
constexpr long double remquol(long double x, long double y, int* quo);

constexpr float copysign(float x, float y); // see [library.c]
constexpr double copysign(double x, double y);
constexpr long double copysign(long double x, long double y); // see [library.c]
constexpr float copysignf(float x, float y);
constexpr long double copysignl(long double x, long double y);

double nan(const char* tagp);
float nanf(const char* tagp);
long double nanl(const char* tagp);

constexpr float nextafter(float x, float y); // see [library.c]
constexpr double nextafter(double x, double y);
constexpr long double nextafter(long double x, long double y); // see [library.c]
constexpr float nextafterf(float x, float y);
constexpr long double nextafterl(long double x, long double y);

constexpr float nexttoward(float x, long double y); // see [library.c]
constexpr double nexttoward(double x, long double y);
constexpr long double nexttoward(long double x, long double y); // see [library.c]
```

```
constexpr float nexttowardf(float x, long double y);
constexpr long double nexttowardl(long double x, long double y);

constexpr float fdim(float x, float y); // see [library.c]
constexpr double fdim(double x, double y);
constexpr long double fdim(long double x, long double y); // see [library.c]
constexpr float fdimf(float x, float y);
constexpr long double fdiml(long double x, long double y);

constexpr float fmax(float x, float y); // see [library.c]
constexpr double fmax(double x, double y);
constexpr long double fmax(long double x, long double y); // see [library.c]
constexpr float fmaxf(float x, float y);
constexpr long double fmaxl(long double x, long double y);

constexpr float fmin(float x, float y); // see [library.c]
constexpr double fmin(double x, double y);
constexpr long double fmin(long double x, long double y); // see [library.c]
constexpr float fminf(float x, float y);
constexpr long double fminl(long double x, long double y);

constexpr float fma(float x, float y, float z); // see [library.c]
constexpr double fma(double x, double y, double z);
constexpr long double fma(long double x, long double y, long double z); // see [library.c]
constexpr float fmaf(float x, float y, float z);
constexpr long double fmal(long double x, long double y, long double z);

// [c.math.lerp], linear interpolation

constexpr float lerp(float a, float b, float t) noexcept;
constexpr double lerp(double a, double b, double t) noexcept;
constexpr long double lerp(long double a, long double b, long double t) noexcept;

// [c.math.fpclass], classification / comparison functions
constexpr int fpclassify(float x);
constexpr int fpclassify(double x);
constexpr int fpclassify(long double x);

constexpr int isfinite(float x);
constexpr int isfinite(double x);
constexpr int isfinite(long double x);

constexpr int isinf(float x);
constexpr int isinf(double x);
constexpr int isinf(long double x);

constexpr int isnan(float x);
constexpr int isnan(double x);
constexpr int isnan(long double x);

constexpr int isnormal(float x);
```

```
constexpr int isnormal(double x);
constexpr int isnormal(long double x);

constexpr int signbit(float x);
constexpr int signbit(double x);
constexpr int signbit(long double x);

constexpr int isgreater(float x, float y);
constexpr int isgreater(double x, double y);
constexpr int isgreater(long double x, long double y);

constexpr int isgreaterequal(float x, float y);
constexpr int isgreaterequal(double x, double y);
constexpr int isgreaterequal(long double x, long double y);

constexpr int isless(float x, float y);
constexpr int isless(double x, double y);
constexpr int isless(long double x, long double y);

constexpr int islessequal(float x, float y);
constexpr int islessequal(double x, double y);
constexpr int islessequal(long double x, long double y);

constexpr int islessgreater(float x, float y);
constexpr int islessgreater(double x, double y);
constexpr int islessgreater(long double x, long double y);

constexpr int isunordered(float x, float y);
constexpr int isunordered(double x, double y);
constexpr int isunordered(long double x, long double y);
```

G. Modifications to "Absolute Values" [c.math.abs]

...

```
constexpr int abs(int j);
constexpr long int abs(long int j);
constexpr long long int abs(long long int j);
constexpr float abs(float j);
constexpr double abs(double j);
constexpr long double abs(long double j);
```