More `constexpr` for `<cmath>` and `<complex>`

Document: D1383R1
Date: October 15, 2022
Project: Programming Language C++, Library Working Group
Audience: SG6 → LEWG → LWG → CWG
Reply to: Oliver J. Rosten (`oliver.rosten@gmail.com`)

Abstract

A scattering of `constexpr`, principally throughout `<cmath>`, was proposed in [P0533] and accepted into C++23. This was subject to a constraint that the affected functions be limited to those which are, in a well-defined sense, no more complicated than the arithmetic operators $+, -, \times, /$. It is proposed to remove this restriction, thereby allowing a richer spectrum of mathematical functions to be used in a `constexpr` context.

CONTENTS

I. REVISION HISTORY

R1 Greatly expanded analysis of the design space.

II. INTRODUCTION

Since its inception, `constexpr` has become an invaluable ingredient in compile-time programming. Indeed, part of its appeal is that the sharp distinction between meta-programming and runtime programming has in many instances become blurred. The interest in `constexpr` is reflected by the numerous papers proposing to increase the range of core language features and library functionality that may be used in a `constexpr` context. As such, it is essential for the long-term uniformity of C++ that parts of the standard library are not left behind in this process.

This paper is the natural extension of [P0533] and seeks to significantly expand the number of functions in `<cmath>` (and also `<complex>`) which may be used in a `constexpr` context. The potential utility of this for numerics is noteworthy. However, it is clear that there are new hurdles to overcome: floating-point is very subtle and different people may want implementations to prioritize different things [P2337]. In other words, there is a non-trivial design space.

Consider some function $f(x)$ and its floating-point implementation $F_R(x_n)$, where the x_n are numbers representable by the floating-point type being used and R denotes the rounding mode. A central issue is whether, for a given x_n and R, $F_R(x_n)$ should always give the same answer, regardless of compiler settings and/or platform. In some cases, the answer is clearly yes: for example if f corresponds to multiplication by two. However, to take a different extreme, what about `std::sin(1e100)`?[1]

Whilst it is of course true that there is an unambiguous result to $\sin 10^{100}$, we may well wonder how meaningful it is, since shifting the argument by a tiny amount—of relative size 10^{-100}—can cause the output to change dramatically. Indeed, consider that the ratio between the size of the observable universe and the Planck scale[2] is 'only' of order 10^{62}. Suppose that we tell an astronomer that we expect them to measure this ratio in order that we can take its sin, and that we further expect to at least get the right sign. This is not entirely different from expecting a standard library implementer to ensure that `std::sin` gives the canonical answer, no matter how large the argument.

But how to reconcile this with the other extreme exemplified by the requirement that multiplication by two should always gives the correct answer? Floating-point numbers can be thought of in two ways: as points on the number line or as intervals (c.f. interval arithmetic [Kahan]). For a given x_n and rounding mode, the corresponding interval is such that all numbers in the interval collapse to x_n under rounding. The granularity of

[1] Thanks to Richard Smith for first bringing this example to our attention.
[2] The unimaginably small distance scale at which quantum gravity presumably reigns.

floating-point is such that $\sin x$ sweeps across its entire range from [-1,+1] many times in the interval containing 10^{100}. Contrariwise, multiplication by two maps the interval associated with x_n into the interval associated with $2x_n$.

Thus, within the interval approach, it is reasonable for different implementations to produce different values for the same input; or even for the same implementation to do so depending on things such as the degree of optimization. This is true of the way C++ works at present. For an example, see https://godbolt.org/z/M3fhYhx84:

```
float num() { return 3.14f; }

int main()
{
    std::cout << std::hexfloat;
    std::cout << num() / 3.14f;
}
```

With the compiler setting `-O2 -ffast-math` the answer is not precisely one; however, with `-O0` it is one. From an interval perspective, this can be rationalized. Perhaps some will baulk at this example due to the use of `-ffast-math`; after all it does not conform to IEEE. But sometimes the flexibility afforded by `-ffast-math`, such as re-ordering of operations, allows it to give a more mathematically correct answer https://godbolt.org/z/bnef8WjYP.

```
float big() { return 1e20f;}

int main()
{
    std::cout << std::hexfloat;
    std::cout << big() + 3.14f - big();
}
```

Removing the `-ffast-math` flag causes the output to regress from the correct answer to zero.

However, even within the interval approach, it is not a free-for-all. Under the action of the true mathematical function, f, the mapping of the floating-point interval containing x intersects with a union of floating-point intervals. The latter define what can reasonably be considered equally valid output of $F_R(x_n)$. Do the various C++ library implementations conform to this? Doubtful! As far as we are aware implementations were not developed with this in mind. Attempting to remedy this via standardization seems counterproductive. The effect would likely be to render all the existing implementations—which for many purposes work perfectly well—non-conforming. Rather, it is suggested that consideration of intervals can in principle be used to quantify the Quality of Implementation.

With all this in mind, the design space divides up as follows:

Canonical The output of mathematical functions is exactly prescribed, down to the last bit.

Interval The output of mathematical functions may fall within a range, generated by considering the mapping of intervals corresponding to floating-point numbers.

Approx A weaker version of [Interval], where deviations from the desired output range are taken to reflect the QoI.

We assume that C++ implementations currently fall within the last of these (if any fall within the second that would be wonderful, but there is a burden of proof to demonstrate this to be the case). Is this a problem? It depends. If you are writing a networked multiplayer game, then lack of support for [Canonical] may be problematic. On the other hand, if you are doing physical modelling where the errors in the boundary conditions are large compared to the floating-point granularity at the appropriate scale then [Approximate] may be perfectly sound. Indeed, [Approximate] may allow vendors to write faster implementations than [Canonical] and why should users pay for a spurious 'improvement' in accuracy?

All of this suggests four (non-exhaustive) avenues for standardization.

1. Allowing existing [Approximate] implementations to be used in a `constexpr` context.

2. A new set of mathematical functions, satisfying the requirements of [Canonical].

3. Demanding existing functions satisfy the constraints of [Interval].

4. Demanding existing functions be [Canonical] and introduce a new set with weaker constraints.

Actually, as far as this paper is concerned the final option is not viable. First, this would compel compiler vendors to change the behaviour of existing implementations; this could have unexpected effects on the behaviour and performance of programs recompiled according to a new standard. Secondly, as discussed above, for periodic functions given a large argument, trying to define a canonical output is an exercise in absurdity. Finally, even away from awkward cases, actually verifying that answers are canonical could be very difficult, especially for `double`s.

As for the third option, this has a certain appeal. However, on balance it is preferred to encourage implementors to use this (or something along the same lines) to quantify their QoI. Insisting that existing implementations strictly adhere to this seems too prescriptive.

While there may be merit in the second option above, this paper proposes standardization of the first. This does not preclude proposing the second option at a future date, though a proper resolution of how to deal with things like `std::sin(1e100)` must be found. Any potential tension is more about the philosophical question of whether [Approx] functions should be usable in a `constexpr` context. Strict adherents to the position of only allowing [Canonical] functions to be used in this way may object to the first option out of principle but, again, let us emphasise that practically speaking there is no reason the two approaches cannot coexists.

To conclude the introduction, given our belief that declaring more functions within `<cmath>` to be `constexpr` is useful, we seek to adhere to one of the core principles of C++ [D&E]:

> It is more important to allow a useful feature than to prevent every misuse.

Whether this paper is ultimately accepted or not boils down to the relative strength of usefulness versus potential for misuse.

III. MOTIVATION & SCOPE

Prior to [P0533], no effort had been made to allow for functions in `<cmath>` to be declared `constexpr`; this despite there being glaring instances, such as `std::abs`, for which this was arguably perverse. Indeed, between [P0415R0] and [P0533] being adopted, the situation was actually been better for `<complex>` than for `<cmath>`! The aim of [P0533] was to at least partially rectify the situation, while recognizing that attempting to completely resolve this issue in a single shot was too ambitious.

The broad strategy of [P0533] is to focus on those functions which are, in a well-defined sense, no more complicated than the arithmetic operators $+, -, \times, /$; the rationale for this being that the latter are already available in a `constexpr` context. As [P0533] proceeded through the standardisation process, LEWG expressed a desire to extend the scope to include a significant amount of what remains in `<cmath>`, in particular common mathematical functions such as `std::exp`.[3] However, later discussion—crystalized in [P2337]—revealed significant worries.

There are several related concerns to the goal of this paper to declare more functions in `<cmath>` to be `constexpr`:

1. Implementations of certain functions in `<cmath>` do not produce results which are 'correctly' rounded to the last bit.[4]
2. The output of certain functions in `<cmath>` may depend on the level of optimization; a corollary is that even with a given level of (non-trivial) optimization, the same function in `<cmath>` may give different answers, depending on the ambient code https://godbolt.org/z/js7rGvPbf.[5]
3. The output of certain functions in `<cmath>` may differ when the same binary is executed on different CPUs within the same architectural family.
4. The output of certain functions in `<cmath>` may differ depending on whether they are evaluated at runtime or translation time.

A conceptual framework for reasoning about these issues has been given in the introduction. If one had the luxury of possessing both a [Canonical] and [Interval] implementation then much of the concern could be allayed by simply choosing the appropriate implementation for the task in hand: in the case of [Interval] the same function may produce different results in different contexts, but it a manner such that all outputs are, in a well defined sense, equally valid.

However, the reality is that the only available implementations are [Approx]. Furthermore, even if an [Interval] implementation were available, some disquiet may follow from the fact that different platforms may do different things in the following example:

```
template<double D>
struct do_stuff
{
  static void execute() {}
};

template<>
struct do_stuff<1.0>
{
  static void execute() { destroy_everything(); }
};

// Do I feel lucky?
do_stuff<std::sin(1e100)>::execute();
```

Nevertheless, we believe that the utility of rolling out `constexpr` to touch more of `<cmath>` outweighs the fact that it may be misused, bringing us back to the core principle cited in the introduction, from [D&E]. It is perfectly reasonable for people to want to generate a `constexpr` lookup table for (say) `std::sin`. If they are operating in the domain where they do not care whether their values differ between platforms etc. is it really right that we continue to prohibit this entirely legitimate use-case?

IV. STATE OF THE ART AND IMPACT ON IMPLEMENTERS

A. Current Implementations

With the exception of the special functions [sf.cmath], functions taking a pointer argument and those with an explicit dependence on the runtime rounding mode, GCC currently renders almost everything in `<cmath>` `constexpr`. Though clang does not have `constexpr` implementations, it does perform compile time evaluation of

[3] It seems too ambitious at this stage to include the mathematical special functions [sf.cmath] and so they are excluded from this proposal.
[4] As discussed in the introduction, it is doubtful whether rounding of e.g. `std:sin(1e100)` can be meaningfully considered correct.
[5] Thanks to Matthias Kretz for supplying this example.

many mathematical functions (but not the special functions) during optimization. The existence of compile time evaluation in GCC and clang demonstrates that implementation of this proposal is plausibly feasible.

Nevertheless, even for GCC's implementation of the relatively simple functions which [P0533] declares `constexpr`, there are subtleties. In particular GCC is not entirely consistent with the way in which it presently deals with NaNs and/or infinities when they are passed as arguments to various mathematical functions.

B. Special Values

Two problems that [P0533] had to deal with was situations in which

1. Floating-point exceptions (other than `FE_INEXACT`) are raised;

2. NaNs and/or infinities are passed as arguments to functions in `<cmath>` declared `constexpr`.

The chosen solution was to delegate to Annex F of the C standard insofar as it applicable. Recall that Annex F specifies C language support for IEC 60559 arithmetic; thus, to the extent that a floating-point type conforms to this, the behaviour in the aforementioned situations is exactly prescribed in C++, following the adoption of [P0533]. Should a floating-point type not conform to relevant parts of IEC 60559, then its behaviour in these situations is unspecified.

This strategy is applicable in its current form to this paper, though the range of scenarios in which Annex F may be invoked is somewhat richer. For example, an implementation conforming to IEC 60559 must give `acos(1) = +0`.

C. Interaction with the C Standard Library

For a mathematical function which may be evaluated at translation time, putting all peculiarities of floating-point momentarily to one side, it is desirable for there to be consistency with the values computed at runtime. However, the fact that the rounding mode may be changed at runtime indicates that this is not, in general, possible.

However, for more complicated mathematical functions there is an additional subtlety due to the interaction with the C standard library. In [library.c] it is noted that `<cmath>` makes available the facilities of the C standard library. One interpretation of this is that the C++ implementation could use one of several different C standard libraries. If so, constraining translation time behaviour so that it is consistent with the runtime behaviour could be very difficult, quite apart from the issue of the runtime rounding mode.

Let us return to an earlier example:

```
#include <cmath>
double f() { return std::sin(1e100); }
```

It turns out that on clang (targeting x64), the following code is emitted:

```
.LCPI0_0:
        .quad         -4622843457162800295
_Z1fv:
        movsd         .LCPI0_0(%rip), %xmm0
        retq
```

with equivalent code generated by GCC. This demonstrates that both compilers are already generating the results at translation time and, therefore, independently of the runtime C library. For this particular example, it appears that current practise does indeed achieve consistency between translation time and runtime, though effectively by ignoring the latter!

The story does not end here. For more complicated examples and/or removing optimization, it may be that a runtime call to the C library is made, after all. Bearing in mind that any value in the range [-1, 1] could be considered reasonable, this implies that the value of, say, `std::sin(1e100)` evaluated in one part of a code base may be very different from the (translation time) value evaluated elsewhere. Nevertheless, it seems reasonable in our opinion that both clang and GCC tacitly allow this, as already discussed in detail.

V. DESIGN DECISIONS

The key design decision advocated in this paper is that it is acceptable for evaluation of a mathematical function to differ between translation time and runtime. Let us recapitulate the various points.

1. Allowing a broader range of mathematical functions to be used within constant expressions is useful; GCC already supports this.

2. Since the advent of `constexpr`, the standard has implicitly allowed for differences between translation time and runtime evaluation: the arithmetic operators $+, -, \times, /$ may be used in either context, but only in a runtime context may the rounding mode be changed.

3. Even without `constexpr`, current practice has long allowed for differences in the output of mathematical functions between any of translation time, runtime, runtime with different compiler flags, and runtime on a different platform. For example, optimization may emit code which entirely bypasses runtime calls to the C library, instead generating results at translation time. However, under other circumstances, optimization might not do this.

4. The philosophy of this paper is not to accept an impasse. Rather, it is preferred to support a useful

extension to existing practice in a non-prejudicial fashion, while not precluding orthogonal developments which may cater for a different range of use-cases. It also advocates for (but does not require) that implementers provide more information on their QoI, perhaps using interval arithmetic as a coherent framework for doing so.

One way or another, all of this boils down to the question of whether it is really acceptable for mathematical functions to give different results in different contexts, given the same input. Again, our answer is yes and we emphasise again that this is already part of C++. That being said, there are cases where people may want a mathematical function to produce the same result, given the same input, in all situations (except, presumably, when the rounding mode is changed). Our opinion is that this is best served by a separate proposal, which perhaps introduces new types. It is worth pointing out that Java gives a good idea of what this may look like.

VI. IMPACT ON THE STANDARD

This proposal amounts to a (further) liberal sprinkling of constexpr in <cmath>, together with a smattering in <complex>.

VII. FUTURE DIRECTIONS

It would be desirable for library vendors to quantify the QoI of mathematical functions, along the lines indicated in section II. Orthogonal to this, it is worth considering separate implementations of mathematical functions with strict guarantees on their outputs. Finally, it would be ultimately desirable to extend constexpr to some, if not all, of the special functions.

ACKNOWLEDGMENTS

I would like to thank Richard Smith for his usual perceptive comments and Matthias Kretz and Nick Timmons for some very helpful feedback and discussions. Particular thanks to Ed Rosten for collaborating on the first incarnation of this paper.

REFERENCES

[P0533] Edward J. Rosten and Oliver J. Rosten, constexpr for <cmath> and <cstdlib>.

[P2337] Nicholas G. Timmons, Less constexpr for <cmath>.

[D&E] Bjarne Stroustrup, The Design and Evolution of C++.

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

[N4910] Thomas Köppe, ed., Working Draft, Standard for Programming Language C++.

[Kahan] William Kahan. 2006. How Futile Are Mindless Assessments of Roundoff in Floating-Point Computation? Retrieved October 15, 2022 from https://people.eecs.berkeley.edu/?wkahan/Mindless.pdf.

VIII. PROPOSED WORDING

The following proposed changes refer to the Working Paper [N4910]. Highlighting in green indicates changes proposed in this paper, whilst blue indicates changes proposed in the companion paper, [P0533].

A. Modifications to "Header `<complex>` synposis" [complex.syn]

// [complex.value.ops], *values*

```
template<class T> constexpr T real(const complex<T>&);
template<class T> constexpr T imag(const complex<T>&);

template<class T> constexpr T abs(const complex<T>&);
template<class T> constexpr T arg(const complex<T>&);
template<class T> constexpr T norm(const complex<T>&);

template<class T> constexpr conj(const complex<T>&);
template<class T> constexpr proj(const complex<T>&);
template<class T> constexpr polar(const T&, const T& = T());
```

// [complex.transcendentals], *transcendentals*

```
template<class T> constexpr complex<T> acos(const complex<T>&);
template<class T> constexpr complex<T> asin(const complex<T>&);
template<class T> constexpr complex<T> atan(const complex<T>&);

template<class T> constexpr complex<T> acosh(const complex<T>&);
template<class T> constexpr complex<T> asinh(const complex<T>&);
template<class T> constexpr complex<T> atanh(const complex<T>&);

template<class T> constexpr complex<T> cos (const complex<T>&);
template<class T> constexpr complex<T> cosh (const complex<T>&);
template<class T> constexpr complex<T> exp (const complex<T>&);
template<class T> constexpr complex<T> log (const complex<T>&);
template<class T> constexpr complex<T> log10(const complex<T>&);

template<class T> constexpr complex<T> pow (const complex<T>&, const T&);
template<class T> constexpr complex<T> pow (const complex<T>&, const complex<T>&);
template<class T> constexpr complex<T> pow (const T&, const complex<T>&);

template<class T> constexpr complex<T> sin (const complex<T>&);
template<class T> constexpr complex<T> sinh (const complex<T>&);
template<class T> constexpr complex<T> sqrt (const complex<T>&);
template<class T> constexpr complex<T> tan (const complex<T>&);
template<class T> constexpr complex<T> tanh (const complex<T>&);
```

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

```
namespace std{
```

```
...

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

constexpr float asin(float x); // see [library.c]
constexpr double asin(double x);
constexpr long double asin(long double x); // see [library.c]
constexpr float asinf(float x);
constexpr long double asinl(long double x);

constexpr float atan(float x); // see [library.c]
constexpr double atan(double x);
constexpr long double atan(long double x); // see [library.c]
constexpr float atanf(float x);
constexpr long double atanl(long double x);

constexpr float atan2(float y, float x); // see [library.c]
constexpr double atan2(double y, double x);
constexpr long double atan2(long double y, long double x); // see [library.c]
constexpr float atan2f(float y, float x);
constexpr long double atan2l(long double y, long double x);

constexpr float cos(float x); // see [library.c]
constexpr double cos(double x);
constexpr long double cos(long double x); // see [library.c]
constexpr float cosf(float x);
constexpr long double cosl(long double x);

constexpr float sin(float x); // see [library.c]
constexpr double sin(double x);
constexpr long double sin(long double x); // see [library.c]
constexpr float sinf(float x);
constexpr long double sinl(long double x);

constexpr float tan(float x); // see [library.c]
constexpr double tan(double x);
constexpr long double tan(long double x); // see [library.c]
constexpr float tanf(float x);
constexpr long double tanl(long double x);

constexpr float acosh(float x); // see [library.c]
constexpr double acosh(double x);
constexpr long double acosh(long double x); // see [library.c]
constexpr float acoshf(float x);
constexpr long double acoshl(long double x);
```

```
constexpr float asinh(float x); // see [library.c]
constexpr double asinh(double x);
constexpr long double asinh(long double x); // see [library.c]
constexpr float asinhf(float x);
constexpr long double asinhl(long double x);

constexpr float atanh(float x); // see [library.c]
constexpr double atanh(double x);
constexpr long double atanh(long double x); // see [library.c]
constexpr float atanhf(float x);
constexpr long double atanhl(long double x);

constexpr float cosh(float x); // see [library.c]
constexpr double cosh(double x);
constexpr long double cosh(long double x); // see [library.c]
constexpr float coshf(float x);
constexpr long double coshl(long double x);

constexpr float sinh(float x); // see [library.c]
constexpr double sinh(double x);
constexpr long double sinh(long double x); // see [library.c]
constexpr float sinhf(float x);
constexpr long double sinhl(long double x);

constexpr float tanh(float x); // see [library.c]
constexpr double tanh(double x);
constexpr long double tanh(long double x); // see [library.c]
constexpr float tanhf(float x);
constexpr long double tanhl(long double x);

constexpr float exp(float x); // see [library.c]
constexpr double exp(double x);
constexpr long double exp(long double x); // see [library.c]
constexpr float expf(float x);
constexpr long double expl(long double x);

constexpr float exp2(float x); // see [library.c]
constexpr double exp2(double x);
constexpr long double exp2(long double x); // see [library.c]
constexpr float exp2f(float x);
constexpr long double exp2l(long double x);

constexpr float expm1(float x); // see [library.c]
constexpr double expm1(double x);
constexpr long double expm1(long double x); // see [library.c]
constexpr float expm1f(float x);
constexpr long double expm1l(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);

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

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

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

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

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

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

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

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

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

constexpr float sqrt(float x); // see [library.c]
constexpr double sqrt(double x);
constexpr long double sqrt(double x); // see [library.c]
constexpr float sqrtf(float x);
constexpr long double sqrtl(long double x);

constexpr float erf(float x); // see [library.c]
constexpr double erf(double x);
constexpr long double erf(long double x); // see [library.c]
constexpr float erff(float x);
constexpr long double erfl(long double x);

constexpr float erfc(float x); // see [library.c]
constexpr double erfc(double x);
constexpr long double erfc(long double x); // see [library.c]
constexpr float erfcf(float x);
constexpr long double erfcl(long double x);

constexpr float lgamma(float x); // see [library.c]
constexpr double lgamma(double x);
constexpr long double lgamma(long double x); // see [library.c]
constexpr float lgammaf(float x);
constexpr long double lgammal(long double x);

constexpr float tgamma(float x); // see [library.c]
constexpr double tgamma(double x);
constexpr long double tgamma(long double x); // see [library.c]
constexpr float tgammaf(float x);
constexpr long double tgammal(long double x);

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.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);
```

C. Modifications to “Three-dimensional hypotenuse” [c.math.hpot3]

```
constexpr float hypot(float x, float y);
constexpr double hypot(double x, double y);
constexpr long double hypot(double x, double y);
```