More constexpr for <cmath> and <complex>

Document: D1383R1
Date: April 6, 2022
Project: Programming Language C++, Library Working Group
Audience: SG6 → LEWG → LWG → CWG
Reply to: Edward J. Rosten (erosten@snap.com) / Oliver J. Rosten (oliver.rosten@gmail.com)

Abstract

In [P0533], a scattering of constexpr, principally throughout <cmath>, was proposed, 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. To help justify this, techniques from elementary topology are utilized, to rigorously define a framework into which existing practice fits reasonably well. Deviations are quantifiable and can be used to compute a number representing the QoI.

CONTENTS

I. Introduction 1
II. Motivation & Scope 2
III. State of the Art and Impact on Implementers 4
IV. Design Decisions 5
V. Impact On the Standard 5
VI. Future Directions 5
Acknowledgments 5
References 5
VII. Proposed Wording 6

I. 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 from 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. Crudely, should implementations of floating-point functions be fast, or should they be accurate?

In fact, the issues run much deeper than this: how is accuracy even meaningfully defined for functions accepting floating-point input? To get a flavour for the subtleties, consider `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, the granularity of floating-point numbers at the scale of 10^{100} is such that $\sin x$ sweeps across its entire range from [-1,+1] many times as we go from one floating-point input to the next available one. Therefore, in this context, the fuzziness of floating-point numbers makes the question of accuracy non-trivial.

On the other hand, if multiplying a number exactly representable as a floating-point number by two produced something off by the last bit, this would be highly concerning! One of the components of this paper is to show how elementary topology can provide a framework to understand both of these extremes without providing a prejudicial opinion on the design space. The purpose of doing so is two fold:

1. The lack of standardization of the output of mathematical functions in C++ means that practice has grown organically and largely unconstrained. A danger of this is that discussions have no grounding within an underlying formalism which can make progress hard. By providing a rigorous conceptual framework, this paper seeks to provide such

[1] Thanks to Richard Smith for first bringing this example to our attention.

grounding but without attempting to artificially and retrospectively constrain existing practice.

2. The framework allows for QoI to be quantified in a way which doesn't simply label all existing implementations as 'bad', as might be the case if one insisted on 'accuracy' to the last bit. Of course, there is no unique measure of QoI but this paper suggests a family which may help inform discussions.

There is one final introductory point to make. 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.

II. MOTIVATION & SCOPE

A. Initial Motivation & Concerns

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`.[2] However, later discussion—crystalized in [P2337]—revealed significant concerns. Our hope is that these concerns can be overcome; to aid meaningful discussion, this proposal devotes space to describing a framework for rigorously conceptualizing the various issues.

There are two 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.

2. The output of certain functions in `<cmath>` may differ depending on whether they are evaluated at runtime or translation time.

B. A Conceptual Framework

1. The Main Idea

The key shift in perspective proposed is to stop thinking of floating-point numbers as simply numbers, and to instead think of them as *labels*. In particular, consider the real numbers, $\mathbf{R}^1$. Next define a sequence of overlapping open sets which cover the entire real number line (excluding $\pm\infty$). The open sets are called charts and the union of these charts is an atlas. Each individual chart, C_{x_n}, holds within it a single floating-point number, x_n, which may be considered its label. The subscript n is used for convenience to index the floating-point numbers. This is ordered such that the next floating-point number is x_{n+1}. Each number is assumed to be unique and so ± 0 are taken to be the same. The closure of C_{x_n} includes the numbers x_{n-1} and x_{n+1} (which may be $\pm\infty$). Thus, each chart goes up to be doesn't quite touch the numbers on either side. This is illustrated in figure 1.

The picture is somewhat reminiscent of Interval Arithmetic [**?**], but with some key differences. First, as will be described, the goal is not to quantify rounding errors, as such. Rather, the goal is to provide a set of *equally valid* answers to a floating-point computations in a sense which will be made precise. Secondly, rather than the closed intervals of Interval Arithmetic, we employ open sets in a way which will be familiar from elementary topolgy.

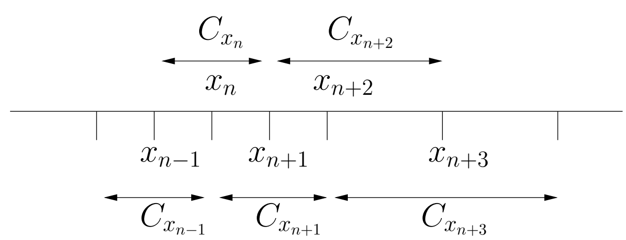

FIG. 1. An atlas for the real numbers, with charts labelled by the floating-point numbers. Note how the gap between floating-point numbers increases, as the exponent increments.

The next step is to consider a function of a single floating-point argument, f, i.e. a function which maps $\mathbf{R}^1$ into itself. This procedure can be straightforwardly extended to $\mathbf{R}^d$ for functions of d arguments. If the function is analytic within C_{x_n}, then this open set will be mapped into some other open set of $\mathbf{R}^1$, denoted $D^{f(x_n)}$. By construction, the intersection of $D^{f(x_n)}$ with the charts, C, is non-empty. Indeed, $D^{f(x_n)}$ intersects at least one chart, and in the case there are several they must have consecutive labels. In this manner, we can instead think of the function as mapping one label, x_n, into a set of labels

[2] It seems too ambitious at this stage to include the mathematical special functions [sf.cmath] and so they are excluded from this proposal.

$\{x_m, \ldots\}$, where the set is non-empty, and the elements are consecutive. This is illustrated in figure 2.

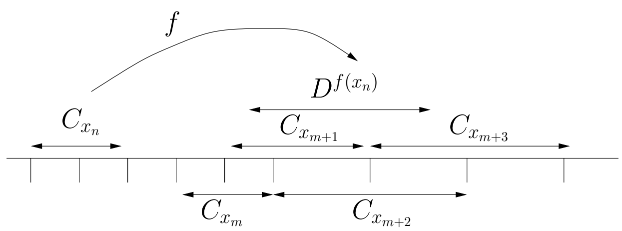

FIG. 2. A function, analytic on the open set C_{x_n}, which maps it into the open set $D^{f(x_n)}$. The latter intersects with $C_{x_m}, \ldots, C_{x_{m+3}}$. In terms of sets of labels, the function can be considered to map the element x_n into the set $\{x_m, \ldots, x_{m+3}\}$.

How does all of this abstract machinery relate to floating-point functions on real hardware? We define a *faithful* floating-point implementation of the function, f, to be a projection from $\{x_m, \ldots\}$ to a single element of this set. This projection will depend on many things: the precise implementation of f, the optimization used, the compiler flags used (such as `-ffast-math`), the rounding mode and the hardware. However, the idea is that every element of the set $\{x_m, \ldots\}$ should be considered equally valid. Some examples may clarify this.

First, consider taking a number which has an exact floating-point representation and multiplying it by two. In this case, $\{x_m, \ldots\}$ comprises just a single entry and our prescription demands an exact calculation. Secondly, take the example of `sin(1e100)`. In this case, $\{x_m, \ldots\}$ comprises all charts labelled by every representable number within the range $-1 \leq y \leq 1$. Our prescription states that every one of these numbers is an equally valid output from a faithful implementation. This puts our earlier intuition on a firm footing.

2. Singularities

There are two types of singularities to consider: those arising from arguments that are/are not exactly representable as a floating-point number. An example of the former is $\ln 0$. An example of the latter is $\tan \pi/2$, since $\pi/2$ cannot be exactly represented as a floating-point number. Both can be incorporated into our framework in a similar way. Any number on the real line will appear in either exactly one of the C_{x_n} or in exactly two: it appears in one if the number is exactly representable as a floating-point number and two, otherwise. Regardless, the next step is to introduce new open sets by copying those charts in which a singular point of the function, f, occurs and deleting said point. Each open set for which this is done spawns a pair of open sets in which the function is now analytic. Under the action of f, these open sets are now mapped into open sets. To complete the construction, the partial closure of these mapped sets is considered by reinstating the singular point. This is illustrated in figure 3, which nicely illustrates how the procedure captures the fact that $\tan \pi/2$ can be either $\pm\infty$, depending on the direction from which the singularity is approached.

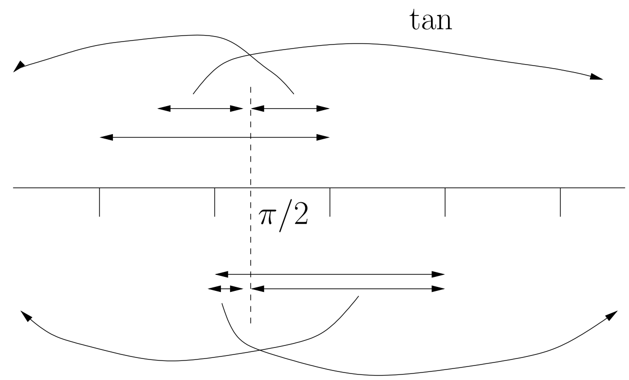

FIG. 3. Excising a singular point from two charts which include $\pi/2$. The curved lines show how the nascent open sets are mapped into $\mathbf{R}^1$ under the action of tan. The new charts to the left of $\pi/2$ are mapped to large positive numbers. Reinstating $\pi/2$ would provide the upper closure of these sets by $+\infty$. Similar considerations apply to the charts to the right of $\pi/2$, though here the mapping is in the vicinity of $-\infty$.

The rest of the prescription described above now follows, almost exactly as before. The only difference is that the $D^{(f_{x_n})}$ may now contain two disjoint, but individually consecutive, sequences.

C. Implications

What does all of this mean in practice, especially when it comes to extending the reach of `constexpr` throughout `<cmath>`? The goal is not to be overly prescriptive, but rather to find a way of coherently and rigorously providing a framework within which much of existing C++ practice and proposed extensions can reside. As such, we do not demand that the results of mathematical functions are necessarily the same at translation time and runtime. Indeed, we know that this can't be guaranteed and indeed differences have long occurred in practice. This is due, amongst other things, to the ability to change the rounding mode at runtime. But what we have done is to put limits on how answers may diverge, at least for faithful implementations.

It is conceivable that this proposal may be abused, causing different platforms to generate radically different code. For example, it will be possible to do things like this:

```
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();
```

However, 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]. As for whether certain results may come as a surprise to users, again this is true. But that is true of floating-point, period. It is a difficult, subtle area of programming fraught with difficulty due to its frequently counter-intuitive nature.

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++. The above framework hopefully makes it clear that floating-point numbers are inherently fuzzy and that it is, in many cases, useful to embrace this fuzziness. 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.

III. STATE OF THE ART AND IMPACT ON IMPLEMENTERS

A. Generalities

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 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. The solution advocated by [P0533] is that, wherever possible, the C++ Standard defers to Annex F of the C Standard. This precisely defines when floating-point exceptions should be raised, and how NaNs and infinities should be treated. Regarding the former, raising anything other than `FE_INEXACT` renders a call non-constant.

B. 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 discussed in detail in section II and further, below.

C. QoI

TO DO:

IV. 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.

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 difference 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—even if present—might not do this.

4. The philosophy of this paper is not to accept an impasse. Rather, it is preferred to describe a rigorous framework which can be used to understand existing practice in a non-prejudicial fashion and allow for extensions in the same spirit.

V. IMPACT ON THE STANDARD

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

VI. FUTURE DIRECTIONS

Ultimately it would be desirable to extend `constexpr` to some, if not all, of the special functions. Orthogonal to this, it may also be worth considering new types with strict guarantees on the values given by associated mathematical functions.

ACKNOWLEDGMENTS

We would like to thank Richard Smith for his usual perceptive comments.

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.
[N4901] Thomas Köppe, ed., Working Draft, Standard for Programming Language C++.

VII. PROPOSED WORDING

The following proposed changes refer to the Working Paper [N4901]. 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);
constexprlong 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);
```