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

Document: P1383R2
Date: June 15, 2023
Project: Programming Language C++, Library Working Group
Audience: SG6 → LEWG → LWG
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.

R2 Rebase wording to N4944 and include overlooked subsections of `<complex>`.

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. First, shifting the argument by a tiny amount—of relative size 10^{-100}—can cause the output to change dramatically. Secondly, the distance between adjacent (double precision) floating-point numbers at this scale is of order 10^{84}. To put this into perspective, 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 first few significant figures right. This is not entirely different from expecting a standard library implementer to ensure that `std::sin` gives the 'correct' answer to the last bit, no matter how large the argument: it is not obvious how to ascribe meaning to the answer.

But how to reconcile this with the other extreme exemplified by the requirement that multiplication by two

[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.

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 floating-point is such that $\sin x$ sweeps across its entire range from [-1,+1] an enormous number of 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, removing the `-ffast-math` flag and/or using `-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 reordering of operations, allows it to give what is, intuitively at any rate, a more mathematically correct answer https://godbolt.org/z/3vehd87h8.

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

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

Removing the `-ffast-math` flag causes the output to change from `3.14f` to zero. However care needs to be taken over-interpreting this particular example. After all, what is meant by a number such as `1e20f`? A set of numbers within a range of order 10^{13} reduce to this under rounding. It is simply not expressible within the language which number within this range is meant by `1e20f`; and how could it be? Floating-point has finite precision. Within the interval approach, all numbers within the appropriate range (once rounding is taking into account) are equally valid. Following this logic through, from an interval approach, the results from both using and not using `-ffast-math` can be considered equally valid. Then again, the interval approach actually permits any answer within what intuitively feels like a large range (though in relative terms, with the scale set by `big()`, the range is small).

In more general terms, the interval approach can be understood as follows. Under the action of the true mathematical function, f, the mapping of the floating-point interval containing x intersects 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 as one measure of the Quality of Implementation (QoI).

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

Canon The output of the floating-point implementation of a mathematical function coincides with that of the underlying function itself, down to the last bit.

Interval The output of the floating-point implementation of a mathematical function 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 either the desired output range or the canonical answer may be 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 [Canon] may be an issue. 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 [Approx] may be perfectly sound. Indeed, [Approx] may allow vendors to write faster implementations than [Canon] and why should users pay for a spurious 'improvement' in accuracy?

All of this suggests the following avenues (non-exhaustive and not all of which are mutually-exclusive) for standardization.

1. Allow existing [Approx] implementations of the mathematical functions to be used in a `constexpr` context demanding either
   (a) No additional requirements, whatsoever;
   (b) No hard requirements but encouraging QoI to be specified in terms of degree of adherence to either [Canon] or [Interval];
   (c) That, in a `constexpr` context, functions satisfy [Canon];

   (d) That, in a `constexpr` context, functions satisfy [Interval].

2. Introduce a new set of mathematical functions, satisfying the requirements of [Canon], and allow them to be used in a `constexpr` context.

3. Demand existing functions always satisfy the constraints of [Interval].

4. Demand existing functions be [Canon] and introduce a new set with weaker constraints.

Actually, as far as this paper is concerned the final option is not viable. First and foremost, this would compel compiler vendors to change the behaviour of existing implementations, which could have unexpected effects on the behaviour and performance of programs. Beyond this, as alluded to above, it is not obvious whether it is constructive to demand [Canon] in all circumstances. Insisting on the 'correct' answer to the last bit can be an exercise in absurdity. Perhaps it may be reasonable to prescribe certain answers—e.g. zero for `std::sin(1e100)`—but how to agree on this value and how to decide where evaluation switches from computation to prescription?[3] There is also the question as to whether relationships such as the trigonometric identities should be preserved[4]; but again it is worth at least critically assessing whether floating-point calculations have any real meaning at all in certain domains. Finally, even away from awkward cases, actually verifying that answers are canonical could be 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 similar 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 1b. This does not preclude proposing the second option at a future date, though a considered resolution of how to deal with things like `std::sin(1e100)` should 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 [Canon] 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 or not proposal 1b is ultimately accepted boils down to the relative strength of usefulness and implementability versus potential for misuse. For 1c the balancing act shifts more towards whether the impact on library implementors and the commensurate delay for end-users is worth it: doing this is as hard as 2. It is also worth noting that the 1c approach of switching between [Appox] and [Canon] implementations via an `if consteval` branch will, in some cases, significantly increase the difference between the compile-time and runtime outputs, compared to 1b.

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`.[5] However, later discussion—crystalized in [P2337]—revealed significant worries.

Various, related concerns to the goal of this paper to declare more functions in `<cmath>` to be `constexpr` have been raised:

1. Implementations of certain functions in `<cmath>` do not produce results which are 'correctly' rounded to the last bit.[6]

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

[3] One reasonable approach would be to make the shift when the floating-point interval between numbers becomes sufficiently large so as to saturate the range of a function.

[4] Thanks to Hans Boehm for pointing this out.

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

[6] As discussed in the introduction, it is doubtful whether rounding of e.g. `std:sin(1e100)` can be meaningfully considered correct, though it may nevertheless serve as a useful prescription for giving a standard answer.

3. The output of certain functions in `<cmath>` may vary from platform to platform.

4. 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.[7]

5. The output of certain functions in `<cmath>` may differ when the same binary is executed on different CPUs within the same architectural family.

Actually, concerns like these can be subsumed into three broader worries:

1. Is it acceptable for evaluation of mathematical functions to differ between translation time and runtime?

2. Is is acceptable for the evaluation of mathematical functions at runtime to depend on things other than the rounding mode?

3. Is it acceptable for constant evaluation of mathematical functions to differ between platforms?

A conceptual framework for reasoning about these issues has been given in the introduction. If one had the luxury of possessing both a [Canon] and [Interval] implementation then much of the concern regarding the first and second issues could be allayed by simply choosing the appropriate implementation for the task in hand, accepting the tradeoffs. If an identical answer is required in all situations (except possibly when the runtime rounding mode is changed), then use the [Canon] implementation. If the cost of this is undesirable then use [Interval]: the same function may produce different results in different contexts, but in 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]. Nevertheless, C++ has long allowed for differences between translation time and runtime; and runtime evaluation itself depends on things such as the degree of optimization. It is certainly the case that quantification of this may very well be useful and can feed into QoI; but in of itself long-standing practice shouldn't present an obstacle to the goals of this paper. But what of whether it is acceptable for constant evaluation of mathematical functions to differ between platforms? 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();
```

That being said, even if the result were guaranteed to be the same on every platform, the code is no less ridiculous. Indeed, 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?

Furthermore, there is a case to be made that allowing additional functions within `<cmath>` to be used in constant expressions actually confers a greater degree of control to users. As things stand, whether constant folding is performed can depend on a variety of factors, such as the degree of optimization and the ambient code (as illustrated earlier). By allowing people to write things such as

```
constexpr auto x{std::exp(1.0)};
```

they are able to specify their intent within the language: evaluate this at translation time.

This naturally leads us to consider the fact that the latest C working draft carves out identifiers prefixed with `cr_` to represent [Canon] implementations of mathematical functions. If we are to work on the assumption that such functions will at some point materialize, then why not leave `<cmath>` as it is and only allow the `cr_` functions in a `constexpr` context? After all, in the above example the intent to evaluate at translation time could be perfectly well specified, without any appreciable downside, by

```
constexpr auto x{std::cr_exp(1.0)};
```

But here's the problem: mathematical functions are used to build up other functions and it may be desirable to use such functions both in a standard runtime context and also in a `constexpr` context, for all the reasons given above. For users who do not care that these functions are not [Canon]—indeed, they may very well desire the

[7] Thanks to Matthias Kretz for supplying this example.

runtime performance gains that a less prescriptive implementation may confer—it is not reasonable to expect them to implement all of their functions twice.

Thus, regardless of whether or not C++ ends up with [Canon] mathematical functions

1. There will always be a desire from some quarters for the existing `<cmath>` behaviour;
2. Allowing people to use these functions in a `constexpr` context has benefits.

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 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.

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 decisions advocated in this paper are that:

1. It is acceptable for evaluation of mathematical functions to differ between translation time and runtime.

2. It is acceptable for constant evaluation of mathematical functions to differ between platforms.
3. It is preferable to encourage quantification of QoI rather than mandate precise behaviour for existing functions within `<cmath>`.

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. Furthermore, runtime evaluation may invoke instructions such as fused multiply-add (fma), which are not necessarily utilized at translation time: https://godbolt.org/z/ceonfG4cT.
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 (but does not require) that implementers provide more information on their QoI.

One way or another, much 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; we emphasise again that this is already part of C++ and does not imply that C++ is broken! 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.

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 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.

[N4944] 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 June 15, 2023 from `https://people.eecs.berkeley.edu/?wkahan/Mindless.pdf`.

VIII. PROPOSED WORDING

The following proposed changes, indicated in green, refer to the Working Paper [N4944]

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 complex<T> conj(const complex<T>&);
template<class T> constexpr complex<T> proj(const complex<T>&);
template<class T> constexpr complex<T> 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 "Value Operations" [complex.value.ops]

```
...

template<class T> constexpr T abs(const complex<T>& x);
```

```
...

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

...

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

...

template<class T> constexpr complex<T> conj(const complex<T>& x);

...

template<class T> constexpr complex<T> proj(const complex<T>& x);

...

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

C. Modifications to “Transcendentals” [complex.transcendentals]

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

...

template<class T> constexpr complex<T> asin(const complex<T>& x);

...

template<class T> constexpr complex<T> atan(const complex<T>& x);

...

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

...

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

...

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

...

template<class T> constexpr complex<T> cos (const complex<T>& x);

...

template<class T> constexpr complex<T> cosh (const complex<T>& x);

...

template<class T> constexpr complex<T> exp (const complex<T>& x);
```

```
...

template<class T> constexpr complex<T> log (const complex<T>& x);

...

template<class T> constexpr complex<T> log10(const complex<T>& x);

...

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

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

...

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

...

template<class T> constexpr complex<T> sin (const complex<T>& x);

...

template<class T> constexpr complex<T> sinh (const complex<T>& x);

...

template<class T> constexpr complex<T> sqrt (const complex<T>& x);

...

template<class T> constexpr complex<T> tan (const complex<T>& x);

...

template<class T> constexpr complex<T> tanh (const complex<T>& x);

...
```

D. Modifications to "Additional overloads" [cmplx.over]

1 The following function templates shall have additional constexpr overloads:

```
arg           norm
conj          proj
imag          real
```

where `norm`, `conj`, `imag`, and `real` are constexpr overloads.

2 The additional constexpr overloads shall be are sufficient to ensure:

...

3 Function template `pow` has additional constexpr overloads sufficient to ensure...

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

```
namespace std{

...

constexpr floating-point-type acos(floating-point-type x);
constexpr float acosf(float x);
constexpr long double acosl(long double x);

constexpr floating-point-type asin(floating-point-type x);
constexpr float asinf(float x);
constexpr long double asinl(long double x);

constexpr floating-point-type atan(floating-point-type x);
constexpr float atanf(float x);
constexpr long double atanl(long double x);

constexpr floating-point-type atan2(floating-point-type y, floating-point-type x);
constexpr float atan2f(float y, float x);
constexpr long double atan2l(long double y, long double x);

constexpr floating-point-type cos(floating-point-type x);
constexpr float cosf(float x);
constexpr long double cosl(long double x);

constexpr floating-point-type sin(floating-point-type x);
constexpr float sinf(float x);
constexpr long double sinl(long double x);

constexpr floating-point-type tan(floating-point-type x);
constexpr float tanf(float x);
constexpr long double tanl(long double x);

constexpr floating-point-type acosh(floating-point-type x);
constexpr float acoshf(float x);
constexpr long double acoshl(long double x);

constexpr floating-point-type asinh(floating-point-type x);
constexpr float asinhf(float x);
constexpr long double asinhl(long double x);

constexpr floating-point-type atanh(floating-point-type x);
constexpr float atanhf(float x);
constexpr long double atanhl(long double x);

constexpr floating-point-type cosh(floating-point-type x);
constexpr float coshf(float x);
constexpr long double coshl(long double x);

constexpr floating-point-type sinh(floating-point-type x);
constexpr float sinhf(float x);
constexpr long double sinhl(long double x);

constexpr floating-point-type tanh(floating-point-type x);
constexpr float tanhf(float x);
constexpr long double tanhl(long double x);

constexpr floating-point-type exp(floating-point-type x);
constexpr float expf(float x);
constexpr long double expl(long double x);

constexpr floating-point-type exp2(floating-point-type x);
constexpr float exp2f(float x);
constexpr long double exp2l(long double x);

constexpr floating-point-type expm1(floating-point-type x);
constexpr float expm1f(float x);
constexpr long double expm1l(long double x);

constexpr floating-point-type frexp(floating-point-type value, int* exp);
constexpr float frexpf(float value, int* exp);
constexpr long double frexpl(long double value, int* exp);

constexpr floating-point-type ilogb(floating-point-type x);
constexpr int ilogbf(float x);
constexpr int ilogbl(long double x);

constexpr floating-point-type ldexp(floating-point-type x, int exp);
constexpr float ldexpf(float x, int exp);
constexpr long double ldexpl(long double x, int exp);

constexpr floating-point-type log(floating-point-type x);
constexpr float logf(float x);
constexpr long double logl(long double x);

constexpr floating-point-type log10(floating-point-type x);
constexpr float log10f(float x);
constexpr long double log10l(long double x);

constexpr floating-point-type log1p(floating-point-type x);
constexpr float log1pf(float x);
constexpr long double log1pl(long double x);

constexpr floating-point-type log2(floating-point-type x);
constexpr float log2f(float x);
constexpr long double log2l(long double x);

constexpr floating-point-type logb(floating-point-type x);
constexpr float logbf(float x);
constexpr long double logbl(long double x);

constexpr floating-point-type modf(floating-point-type value, floating-point-type* iptr);
constexpr float modff(float value, float* iptr);
constexpr long double modfl(long double value, long double* iptr);
constexpr floating-point-type scalbn(floating-point-type x, int n);
constexpr float scalbnf(float x, int n);
constexpr long double scalbnl(long double x, int n);

constexpr floating-point-type scalbln(floating-point-type x, long int n);
constexpr float scalblnf(float x, long int n);
constexpr long double scalblnl(long double x, long int n);

constexpr floating-point-type cbrt(floating-point-type x);
constexpr float cbrtf(float x);
constexpr long double cbrtl(long double x);

// [c.math.abs], absolute values
...

constexpr floating-point-type hypot(floating-point-type x, floating-point-type y);
constexpr float hypotf(float x, float y);
constexpr long double hypotl(long double x, long double y);

// [c.math.hypot3], three-dimensional hypotenuse
constexpr floating-point-type hypot(floating-point-type x, floating-point-type y, floating-point-type z);

constexpr floating-point-type pow(floating-point-type x, floating-point-type y);
constexpr float powf(float x, float y);
constexpr long double powl(long double x, long double y);

constexpr floating-point-type sqrt(floating-point-type x);
constexpr float sqrtf(float x);
constexpr long double sqrtl(long double x);

constexpr floating-point-type erf(floating-point-type x);
constexpr float erff(float x);
constexpr long double erfl(long double x);

constexpr floating-point-type erfc(floating-point-type x);
constexpr float erfcf(float x);
constexpr long double erfcl(long double x);

constexpr floating-point-type lgamma(floating-point-type x);
constexpr float lgammaf(float x);
constexpr long double lgammal(long double x);

constexpr floating-point-type tgamma(floating-point-type x);
constexpr float tgammaf(float x);
constexpr long double tgammal(long double x);
...
```

F. Modifications to “Three-dimensional hypotenuse” [c.math.hypot3]

```
constexpr floating-point-type hypot(floating-point-type x, floating-point-type y, floating-point-type z);
```