Extensible Math Functions for C++

Document #:	D0000R0
Date:	2026-03-30
Project:	Programming Language C++
Audience:	LEWG SG6

Reply-to:	Stéphane Gros-Lemesre <stephane.groslemesre@gmail.com>

Abstract

This paper proposes a direction for making standard library mathematical functions extensible via a new `std::math` sub-namespace, using `std::math::sqrt` as a representative example. No wording is proposed at this stage. The goal is to gauge committee appetite for the direction before committing to a full proposal covering the entire `<cmath>` surface.

1 Motivation

1.1 The Problem

The C++ standard library provides mathematical functions such as `sqrt` and `abs` in the `std` namespace. These functions are not customization points: calling `std::sqrt(x)` directly bypasses any user-defined overload, even if one exists for the type of `x`.

The idiomatic workaround is to enable ADL via a `using` declaration:

```
using std::sqrt;
return sqrt(x);
```

This allows a user-defined `sqrt` in the same namespace as `x` to be found via ADL, while falling back to `std::sqrt` for built-in types. However, this idiom has a critical limitation: it requires a statement, and is therefore unavailable in contexts that only accept expressions.

Constructor member initializer lists:

```
MyType(A aSq, B b, C c)
    : a(sqrt(aSq)),   // std::sqrt not found through conversion
      b(b),
      c(abs(c))     // std::abs not found through conversion
{}
```

The same applies to default member initializers:

```
struct Foo {
    double x = sqrt(v);  // std::sqrt not found through conversion
};
```

Requires expressions:

```
template<class T>
concept numeric = requires(T x) {
    { abs(x) };  // requirement fails
};
```

There are other, less obvious situations where the same limitation applies, such as constant expressions (array size from a new-type) and template arguments.

In all of these cases, the programmer faces the same three unsatisfactory choices:

Option 1: Call `std::` explicitly

```
template<typename A, typename B, typename C>
MyType(A a, B b, C c)
    : a(std::sqrt(a)),  // silently breaks extensibility
      b(b),
      c(std::abs(c))
{}
```

This compiles and works for built-in types, but silently cuts off any user-defined overload.

Option 2: Restructure to allow a statement

For member initializer lists, this means moving initialization to the constructor body:

```
MyType(A a, B b, C c)
    : b(b)
{
    using namespace std;
    this->a = sqrt(a);  // requires a to be default-constructible
    this->c = abs(c);   // loses const and reference member support
}
```

This restores ADL but members can no longer be `const` or references, and all members must be default-constructible. Not all contexts can be restructured this way.

Option 3: Write boilerplate helper functions

```
template<class T> auto my_sqrt(T&& x) {
    using std::sqrt;
    return sqrt(std::forward<T>(x));
}

MyType(A a, B b, C c)
    : a(my_sqrt(a)),
      b(b),
      c(my_abs(c))
{}
```

This restores extensibility but requires every author of generic code to write and maintain their own dispatch layer; one wrapper per math function, 45+ to be exhaustive. The ownership of these wrappers is unclear. Custom numeric-type authors should probably embed them in their math function implementations, but authors of generic code using such functions cannot rely on this being provided and may need to implement them redundantly to support a wider range of types. This makes for poor separation of concern.

The existence of multiple widely-used libraries implementing exactly this machinery (see Existing Practice below) is evidence that there are real use-cases and that the status quo is encouraging unnecessary duplication.

1.2 Existing Practice

Multiple independent, widely-used C++ libraries have been confronted with this problem and have each arrived at their own workaround.

mp-units, **Eigen**, and **Boost.Units** have all independently converged on the same core mechanism: bring `std::sqrt` into scope and let ADL do the work.

```
using std::sqrt;
return sqrt(x);
```

The surrounding machinery differs:

— mp-units adds an explicit member function check. [mp-units]

— Eigen wraps the dispatch in a traits struct with SIMD specialisations and relies on macros to minimize the duplication between different functions. [eigen-math]

```
#define EIGEN_MATHFUNC_IMPL(func, scalar) \
  Eigen::internal::func##_impl<typename \
  Eigen::internal::global_math_functions_filtering_base<scalar>::type>
```

— Boost.Units applies the pattern to the inner value of a quantity type. [boost-units-sqrt]

But the fundamental approach is identical in all three. This independent convergence on the same pattern is strong evidence both that the need is real and that the solution is well-understood, making this a natural candidate for standardization.

— **nholthaus/units** takes a different approach: it provides `units::math::sqrt` which calls std::sqrt directly on the underlying scalar value, which means it does not enable ADL resolution and thus does not extend to custom underlying types. [nholthaus-units]

1.3 Existing Code

A large body of existing generic C++ code calls `std::sqrt` directly, either out of habit or because the author was unaware of the ADL idiom. This code silently fails to work with user-defined types that provide their own `sqrt` and restricts the composability of generic code. There is no practical way for users of such libraries to fix this without modifying the library itself.

1.4 Teachability

The `using std::sqrt; sqrt(x);` idiom is specialist knowledge. The C++ Core Guidelines [CppCoreGuidelines] discuss ADL in the context of operators and `swap`, but do not provide explicit guidance on its application to math functions. An intermediate C++ programmer following the guidelines would have no reason to know this pattern is necessary. Calling `std::sqrt(x)` looks correct and compiles cleanly, but breaks extensibility for custom types.

The following exchange from [nholthaus-issue39] is instructive. A user reports that generic algorithms using ADL-found math functions do not work with unit types, and the library author responds:

> *"Honestly, I guess I just never use ADL because I pretty much exclusively use fully qualified namespaces in my code, and I didn't put thought into it."*

This should not be surprising. The responsibility of making custom types work intuitively should belong to their authors, not to their users or third-party. It takes conscious effort for a generic library author to anticipate the needs for wrappers of hypothetical custom types and explicitly provide support for them. Arithmetic operators require no such effort: they are defined by the type author and compose transparently in generic code. Math functions should be no different.

In the current situation, it is easy to do the wrong thing, and difficult to do the right one.

1.5 Why This Belongs to the Standard Library?

The core problem outlined here is essentially jurisdictional: the naive solution of overloading the math functions in the `std` namespace is undefined behaviour and the alternative solutions are less satisfactory. This proposal

cannot be implemented by users and the bridge between the `std` namespace and user-defined namespace must therefore be built from within the Standard Library.

Without this facility, each library has to implement their own partial bridge, resulting in duplicated code and no universal solution.

2 Proposed Direction

We propose the introduction of a `std::math` sub-namespace containing ADL-aware wrappers for `<cmath>` functions. For concision, we will be using `sqrt` as the representative example in this section.

The sub-namespace is introduced to guarantee that existing code is unaffected by this change. A compatibility layer for `std::sqrt` is also proposed further down.

2.1 `std::math::sqrt` as a Customization Point Object

Rather than a plain function template, `std::math::sqrt` is proposed as a Customization Point Object (CPO) — a `constexpr` global function object whose `operator()` performs the dispatch. This design, established by the Ranges library (`std::ranges::begin`, `std::ranges::swap` etc.), offers these two advantages:

- The CPO cannot be found by ADL on user types, since it is an object rather than a function. Calling `std::math::sqrt(x)` always invokes the CPO's `operator()` explicitly, preventing accidental interception.
- The `operator()` can be constrained, making the CPO SFINAE-friendly and allowing it to be used correctly inside `requires` expressions and concepts.

The proposed implementation:

```
namespace std::math {

namespace __sqrt {

    // Poison pill: prevents unqualified ADL calls from accidentally
    // finding std::math::sqrt during concept checking
    void sqrt(auto) = delete;

    template<class T>
    concept has_member_sqrt = requires(T&& x)
    {
        std::forward<T>(x).sqrt();
    };

    template<class T>
    concept has_adl_sqrt = !has_member_sqrt<T> && requires(T&& x)
    {
        sqrt(std::forward<T>(x)); // ADL only; poison pill blocks std::math::sqrt
    };

    template<class T>
    concept has_std_sqrt = !has_member_sqrt<T> && !has_adl_sqrt<T> && requires(T&& x)
    {
        std::sqrt(std::forward<T>(x));
    };

    struct __fn
    {
        template<class T>
```

```
        requires has_member_sqrt<T>
              || has_adl_sqrt<T>
              || has_std_sqrt<T>
        constexpr auto operator()(T&& x) const
        {
            if constexpr (has_member_sqrt<T>)
                return std::forward<T>(x).sqrt();       // member customization
            else if constexpr (has_adl_sqrt<T>)
                return sqrt(std::forward<T>(x));        // ADL
            else
                return std::sqrt(std::forward<T>(x));   // std fallback
        }
    };

} // namespace __sqrt

inline namespace __cpo {
    inline constexpr __sqrt::__fn sqrt{};
}

} // namespace std::math
```

The dispatch priority is explicitly:

1. Member function `x.sqrt()` — unambiguous, not subject to ADL conflicts
2. Free function found via ADL in the type's own namespace
3. `std::sqrt` as the final fallback for all legacy types

The explicit separation of ADL and `std::sqrt` lookup into distinct steps also prevents ambiguity for types with multiple implicit conversions: ADL is checked first, and `std::sqrt` only participates if no ADL candidate is found.

The three concepts `has_member_sqrt`, `has_adl_sqrt`, and `has_std_sqrt` make the CPO properly SFINAE-friendly. A `requires` expression such as:

```
template<class T>
concept has_sqrt = requires(T x) {
    std::math::sqrt(x);
};
```

correctly evaluates to `false` for types that support none of the three dispatch paths, rather than always appearing satisfied as an unconstrained function template would.

2.2 `std::math` namespace

In the code illustration above, the CPO is defined in a new sub-namespace of `std`: `std::math`. This is not absolutely necessary to enable the extensibility of math functions, but offers several advantages: - Reusing the same names as the functions the CPOs refer to such as `abs`, `sqrt`, `sin`, `pow`, `log`, etc. without ambiguity. - Clear separation of behaviours: `std` namespace functions are unchanged, working code leveraging them is unaffected; `std::math` offers a new, different contract where ADL resolution allows for user-defined extension. - Providing a more focused name space that can be leveraged for autocompletion, separate from the numerous prefixed C-compatible versions.

This also seems to follow recent precedents such as `std::ranges::sort` alongside `std::sort`.

If introducing this namespace is considered undesirable, it would be possible to introduce the CPO directly in `std` instead, likely with a distinct prefixed name (`std::ext_sqrt`). Additionally, types defining a conversion to

primitive types such as `float` or `double` would not be candidate for ADL resolution, as it would risk changing the behaviour of existing code.

2.3 Preserving Legacy Behaviour

With this fixture in its own namespace, the legacy behaviour would be preserved safely, but to extend the benefits of this approach to code calling `std::sqrt` directly, we additionally propose an opt-in compatibility forwarding layer in `namespace std`. This compatibility layer prioritizes maintaining the legacy behaviour, ensuring legacy code doesn't change behaviour while allowing custom types where safely possible :

```
namespace std {

// Opt-in trait for the compatibility forwarding layer.
// Users specialise this for their own types.
template<class T>
inline constexpr bool is_math_extensible = false;

// Forward to the extensible layer for opted-in types
// outside the legacy domain.
template<class T>
constexpr auto sqrt(T&& x) -> decltype(math::sqrt(std::forward<T>(x)))
    requires is_math_extensible<std::remove_cvref_t<T>>
{
    return math::sqrt(std::forward<T>(x));
}

} // namespace std
```

A typical use case is a custom numeric type used with an existing library that calls `std::sqrt` directly and cannot be modified:

```
// Generic library using std::sqrt directly
namespace someLib
{
    template<typename T>
    auto someFunction(T&& someValue)
    {
        // some code...
        return std::sqrt(std::forward<T>(someValue));
    }
}

// User's custom type, with its own sqrt in its namespace
namespace mylib
{
    struct Scalar { ... };
    Scalar sqrt(Scalar x) { ... }
}

// Opt in to the compatibility layer
template<>
inline constexpr bool std::is_math_extensible<mylib::Scalar> = true;

// Now where std::sqrt(mylib::Scalar{}) is used in the library, it is
// forwarded to mylib::sqrt via std::math::sqrt
```

```
someLib::someFunction(mylib::Scalar{5.2});  // now works
```

The two paths have deliberately different behaviour:

— `std::sqrt` behaviour is unchanged for all existing types. User-defined types that opt-in via `is_math_extensible` are forwarded to the extensible layer.
— `std::math::sqrt` is **customization first**: member and ADL customizations are preferred, with `std::sqrt` as the fallback. No opt-in is required.

A proof of concept compiling under GCC, Clang, and MSVC is available at: https://godbolt.org/z/cxYTozPrc

2.4 Alternative implementation

If [P2806R3] (do expressions) and [P2826R2] (replacement functions) are accepted for C++29, the implementation of `std::math::sqrt` reduces significantly.

The do expression makes the ADL idiom available in expression contexts, and replacement functions provide expression-equivalence guarantees. This would allow the entire CPO to be expressed as a single declaration, without explicit dispatch concepts or a poison pill. The direction proposed here would compose naturally with these future language improvements.

For illustration purposes, the implementation could become as simple as:

```
namespace std::math
{
  static constexpr struct
  {
    using operator()(auto&& x) = (do { using std::sqrt; do_return sqrt(std::forward<decltype(x)>(x)); });
  } sqrt;
}
```

eliminating layers of inlining context.

3 Scope of This Proposal

This paper deliberately addresses only `std::math::sqrt` as a proof of concept. The full set of `<cmath>` functions that would eventually need `std::math::` equivalents includes and is not limited to:

Category	Functions
Power / root	`sqrt`, `cbrt`, `pow`, `hypot`
Exponential / logarithmic	`exp`, `exp2`, `expm1`, `log`, `log2`, `log10`, `log1p`
Trigonometric	`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
Hyperbolic	`sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
Rounding	`ceil`, `floor`, `trunc`, `round`, `nearbyint`, `rint`
Absolute value / sign	`abs`, `fabs`, `copysign`, `signbit`
Floating point	`fmod`, `remainder`, `fma`, `fdim`, `fmax`, `fmin`
Classification	`isfinite`, `isinf`, `isnan`, `isnormal`, `fpclassify`

The following are explicitly out of scope for this paper at this stage:

— A complete `std::math` namespace covering all `<cmath>` functions
— Any changes to the existing `std::sqrt` observable behaviour for types already handled by existing `std::sqrt` overloads.

4 Suggested Polls

1. Would allowing a mechanism for user-defined extension of mathematical functions be desirable?
2. Would this group prefer to isolate such a mechanism in its own sub-namespace (e.g. `std::math`)?
3. Is the compatibility layer in namespace `std` desirable (preserving existing behaviour, while allowing extension for opted-in types only)?
4. Would the committee encourage more work on this topic, especially expanding the scope to more functions from `<cmath>`?

5 References

[boost-units-sqrt] Boost Developers. Boost.Units: sqrt Implementation.
https://github.com/boostorg/units/blob/develop/include/boost/units/cmath.hpp

[CppCoreGuidelines] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines.
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

[eigen-math] Eigen Developers. Eigen: MathFunctions.h Implementation.
https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/Core/MathFunctions.h

[mp-units] Mateusz Pusz. mp-units: Math Dispatch Implementation.
https://github.com/mpusz/mp-units/blob/b0e72810b983841b260d570b241c52586aa78999/src/core/include/mp-units/framework/representation_concepts.h#L227-L262

[nholthaus-issue39] Nick Holthaus. nholthaus/units Issue #39: ADL and Math Functions.
https://github.com/nholthaus/units/issues/39

[nholthaus-units] Nick Holthaus. nholthaus/units Library.
https://github.com/nholthaus/units

[P2806R3] Barry Revzin, Bruno Cardoso Lopez, Zach Laine, Michael Park. 2025-01-12. do expressions.
https://wg21.link/p2806r3

[P2826R2] Gašper Ažman. 2024-03-18. Replacement functions.
https://wg21.link/p2826r2