Unified floating point rounding proposal

Doc. No: P4231R0
Contact: Hans Boehm <hboehm@google.com>
Additional authors: Guy Davidson <gd@6it.dev>, Jan Schultke <janschultke@gmail.com>
Audience: LEWG, SG6
Date: May 11, 2026
Target: C++29

Abstract

This is an attempt to unify P2746R7, which introduced a library to provide dynamic floating point rounding mode control, with P3864, which proposed a facility for correctly rounded arithmetic, mostly motivated by reproducibility concerns. P2746 provided a more general facility, but left IEC 60559 conformance optional.
This paper presents a modified version of P2746. that follows P3864 in requiring IEC 60559 conformance where the type has the correct format.

Introduction

Please see P2746 and P3864 for detailed discussions of the two distinct motivations driving this proposal.
Here we focus on the details of the proposed API, borrowing heavily from both papers. The overall structure of the API is more similar to P2746, since SG6 previously went through a number of alternatives in connection with that proposal, arriving at the rounding-mode-encapsulated-by-struct-of-templates approach.

A disadvantage of this P2746 approach is that it is not C-friendly. However, we have detected no interest in these proposals from WG14, and C has alternate, mostly not-C++-friendly alternatives for addressing some of these issues. If this becomes an issue, it seems easy for the user or a later standard to wrap this in less clean C++ syntax that is C-friendly..

A unified proposal

We thus propose just the addition of the struct described below, with a quite modest number of member functions.

A correctly-rounded arithmetic example

In this approach, assuming `numeric_limits<double>::is_iec559` is true, we can compute a correctly rounded version of -0.1 - (x + y) with

```
constexpr rounded cr();

cr.sub(cr.make<double>(“-0.1”), cr.add(x, y));
```

The call to make here forces IEEE round-to-nearest for the constant. Depending on the reading of the current standard, this might be optional, and can perhaps be abbreviated as simply

```
cr.sub(-0.1, cr.add(x, y));
```

But the reasoning behind that simplification is unpleasantly complex:

[lex.fcon] p3 states:

“… the value of a floating-point-literal is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.”

It is not completely clear to us whether this requires round-to-nearest except in the equidistant case, or allows more implementation freedom. In the former case, it becomes optional:

Since we know that -0.1 is a binary repeating fraction, it cannot be half-way between the two closest representable double values, and thus we cannot encounter the “implementation-defined” case. Similar reasoning applies to the vast majority, but not all, literals.

A directed rounding example

To compute an upper bound on the true result of -0.1 - (x + y) we may write:

```
constexpr rounded round_up(round_toward_infinity);
constexpr rounded round_down(round_toward_neg_infinity);

round_up.sub(round_up.make<double>(“-0.1”), round_down.add(x, y));
```

Note that in this case, the use of “make” is not optional, since the literal would otherwise be incorrectly rounded to the nearest `double`.

The actual proposal

We propose the addition of the following struct. Actual wording is still left as future work (some of which can be taken from p3864). This is basically identical to P2746, except that we constrain operations to be IEEE conformant whenever is_iec_559 is `true`.

```
struct rounded {
   float_round_style round_style;  // Exposition only.

   // Construct a rounded instance with the given round_style.
   constexpr rounded(float_round_style rs = round_to_nearest);

   // For convenience; yields numeric_limits<F>::is_iec559
   // Changed: Discuss naming.
   template<floating_point F> is_iec_559();

   // The following functions should ideally all be constexpr.
   // The general SG6 sentiment was to postpone that until there
   // is demonstrated demand for that, since it involves significant
   // implementation effort.

   // The following member functions of rounded(rs) use rounding mode rs.

   template<floating_point F> constexpr F add(F x, F y) const;

   template<floating_point F> constexpr F sub(F x, F y) const;

   template<floating_point F> constexpr F mul(F x, F y) const;

   template<floating_point F> constexpr F div(F x, F y) const;

   // Compute x * y + addend.
   // If conforms_to_iec_60559() yields true, then the result is
   // the correctly rounded value of that entire expression, i.e.
   // only a single rounding is performed.
   template<floating_point F> F fma(F x, F y, F addend) const;

   // Round to a different floating point type.
   // Conversion is expected to be exact if every value representable by G
   // can be exactly represented by F.
   template<floating_point F, floating_point G> F cast(G x) const;

   // Convert a string s representing a constant to the floating
   // point value it represents. S may be a signed floating-point
   // constant consisting of an optional minus sign, followed by a
   // sequence of decimal digits, and an optional decimal point
   // anywhere before, after, or in the middle of that sequence of
   // digits. It is implementation-defined which other
```

```
    // constant strings representing constant floating-point
    // expressions are supported. Throws std::format_error
    // if the argument is not supported.
    // Note: This is the primary mechanism for expressing constants.
    // The from_chars API would be inconvenient.
    template<floating_point F> constexpr F make(string_view s) const;

    // Respects the rounding mode, in that round_toward_infinity
    // produces a decimal string whose exact value is a number no less
    // than x, and correspondingly for round_toward_neg_infinity and
    // round_toward_zero.
    // Conversion to decimal character sequences of at most cr_decimal_dig
    // digits are correctly rounded. For higher precision values, any
    digits
    // beyond this are zero to ensure reproducibility on implementations
    // with the same value of cr_decimal_dig.
    template<floating_point F>
    to_chars_result to_chars(char* first, char* last, F value,
                             chars_format fmt, int precision) const;

    // The number of decimal destination digits for which to_chars
    guarantees
    // correct rounding. Should correspond to C's CR_DECIMAL_DIG when
    possible,
    // but can be 0.
    static constexpr int cr_decimals_dig;

    template<floating_point F> F sqrt(F x) const;

    // Round x to an integer, according to rounding mode rs,
    // returning an R. R must satisfy the integral or floating_point
    // concepts. Overflows handled via floating-point exceptions.
    template<typename R, floating_point F> R rint(F x) const;
    template<typename R, floating_point F> R nearbyint(F x) const;

    // We may eventually want to add assert_exact as another rounding mode
    here?
};
```

Currently we do not provide literals corresponding to directed rounding. This somewhat reduces the problem that -0.1 rounds in an unexpected direction.

```
-rounded(round_toward_infinity).make<float>("0.1")
```

yields -0.1 rounded downward, but hopefully makes that less surprising, but

```
rounded(round_toward_infinity).make<float>(“-0.1”)
```
rounds fully as expected.

We require that `rounded(...).make()` works for simple fixed point constants, optionally preceded by a minus sign. This should be easy to implement. It is in fact usually possible to guarantee correctly rounded evaluation of more complex expressions, and that would probably be ideal, though perhaps a bit challenging to specify and implement. It is not clear that this would be used enough to be worth the implementation cost, so we leave support for a wider variety of expressions implementation-defined.

Our expectation is that other math functions in the same style may eventually be added, if demand warrants. We do not add `rounded::` functions until we are convinced that practically useful implementations of the correctly rounded functions are feasible. This is already the case for many more functions than are listed here, especially for `float` operations. See for example, Lim and Nagarakatte, “High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations”.

Semantics

We propose to require that:

1. If `numeric_limits<F>::is_iec559` is `true`, then all provided `rounded` operations (except possibly `to_chars`) provide correctly rounded IEC559-conforming results. `round_to_nearest` resolves ties to even.
2. At a minimum, `round_toward_infinity`, `round_toward_neg_infinity` and `round_toward_zero` should bound the true results, and round-to-nearest make a strong effort to provide the nearest result
3. In the presence of (eventually deprecated) `fesetround()`, `fesetround()` calls will not affect `rounded::` operations.
4. Floating-point exceptions will be handled as they are now, i.e. via `fegetexcept()`.

Implementation Discussion

Our requirements on `is_iec559` = `false` types are rather lax, making those implementations strictly easier. (It is not completely clear we want to support that case at all, but it can be useful for directed rounding.)

We thus focus on `is_iec559` = `true`. If we assume that the underlying hardware conforms to IECC 60559, then it is generally easy to implement the basic operations as short sequences of assembler code. If the hard flushes to zero, this needs to be augmented by software to handle that case correctly. If the rounding mode can only be supplied via a control register, then the assembly code will have to temporarily change it, and then restore the original mode. So long as the implementation continues to support `fesetround()`, then this code will have to handle prior non-default rounding modes and properly restore them. All of this may not be optimal, but is straightforward.

The greater difficulty is in allowing compile-time "`constexpr`" evaluation. This can be addressed in two ways:

1. We can implement these operations as compiler intrinsics. The compiler can then perform compile-time evaluation by executing those same assembly operations at compile-time. This also potentially allows the compiler to coalesce updates to the floating-point control register.

2. We can implement the operations in a library as:

```
if consteval {
        < perform correctly rounded operation in software >
} else {
        < assembly code for runtime operation >
}
```

   The Berkeley SoftFloat implementation should almost work for compile-time evaluation, after some minor restructuring to eliminate non-constexpr-friendly constructs, notably goto statements.Some preliminary investigation suggests that all relevant bit manipulation and low-level floating-point functions are already constexpr in C++23.

There is prior work on correctly-rounded base conversion, such as that required by make() and to_chars().
See, for example, David M. Gay, "Correctly Rounded Binary-Decimal and Decimal-Binary Conversions". MPFR appears to provide implementations.

The implementation of `make()` reduces to assembling the sequence of digits, converting it to an unbounded integer, left-shifting by the number of bits required to generate the correct mantissa length, and finally dividing it by the appropriate power of 10. The remainder can then be used to determine the correctly rounded last bit. Thus the key to making this a constexpr function appears to be the implementation of unbounded integer division in a constexpr function. As far as I can tell, that appears entirely feasible.

The `to_chars()` function is intended to be implementable by relying on C's `printf()`, either by actually invoking it from `to_chars()` after temporarily setting the correct rounding mode, or by invoking the same routines used to implement `printf()`.

The `rint()` and `nearbyint()` functions are also implementable by specializing the template to versions that call the corresponding C function after temporarily setting the correct rounding mode.

We hope to soon provide an "existence-proof" implementation that supports runtime evaluation with fully usable and deterministic semantics, but provides only a low-quality non-IEEE implementation for compile time evaluation.

Questions

The rounding-mode-encapsulated-by-struct-of-templates was agreed upon in the context of explicitly specified rounding modes, which is accepted to be a niche use. Is that still the right approach for correctly rounded arithmetic, which may be more broadly used?

What is the current requirement on literal rounding?