Deprecate and Replace Fenv Rounding Modes

Doc. No: P2746R4
Contact: Hans Boehm (hboehm@google.com)
Audience: SG6, LEWG
Date: Feb. 14, 2024
Target: C++26

Abstract

We argue that floating point rounding modes as specified by `fesetround()` are largely unusable, at least in C++. Furthermore, this implicit argument to floating point operations, which is almost never used, and largely opaque to the compiler, causes both language design and compilation problems. We thus make the somewhat drastic proposal to deprecate it, in spite of its long history, and to replace it with a much better-behaved facility. This is still a preliminary proposal for feedback on the shape of the replacement.

[First-time readers: Skip to “Introduction”.]

History

Issaquah, Feb. 2023, SG6

1. We would still like more feedback on use cases. So far, the only known ones are (1) bunding true results, e.g for interval arithmetic, and (2) perturbing the results to get a heuristic idea of numerical stability. Further instances of these were brought up during the meeting.
2. There was unanimous agreement that we should strive to replace `fesetround()`. There were no supporters of the current API. Most of us thought we should replace it with a library with explicit rounding mode arguments. There was some support for a C23-style statically scoped mechanism, together with some optimism that this may result in implementations. There was a volunteer to develop such a proposal. We do not plan to pursue this approach in this proposal, but welcome the opportunity to compare the two.
3. There was agreement that any library should follow the simplest possible route and not introduce a separate type for correctly rounded floats, but instead simply provide free functions on the existing types. This applies in spite of the danger of overlooking operations that could affect rounding.
4. There was also consensus that we should use nominally runtime rounding mode arguments, in spite of the valid concern that this would increase compile time. Given the scarcity of actual use cases for the current mechanism, we did not feel this is likely to turn into a significant issue.

Feb 28, 2023 C floating point committee meeting (virtual)

Also no real attachment to `fesetround()`. General preference for the C23 statically scoped `#pragma` solution. Fairly strong preference for keeping floating point exception environment dynamic. (FP exceptions were not discussed in SG6.)

Prior email discussions pointed out that the CORE-MATH project is developing libraries that (1) produce correctly rounded results for all rounding modes, and (2) generally do not actually adjust round modes. The general approach seems to be to initially carry enough extra precision for the rounding mode not to matter, and then perform one final operation that combines say, two halves of the result using the current dynamic rounding mode. This allows a C23 implementation to call a single math function to produce correctly rounded results depending on the current dynamic rounding mode. This proposal would probably remove the need for such strong guarantees, but such a library could still be used to implement correctly-rounded functions discussed here.

Varna, June 2023, SG6 (includes results of further investigation by author)

1. *It was pointed out that it is common to disable subnormals/denormals/gradual underflow on x86, and changing this locally around cr_ library calls is probably infeasible.* Further investigation showed:

   X86 implementations other than x87 (SSE and later) commonly need software assist for gradual underflow, potentially making performance unacceptable when dealing with many near-zero values. (Gcc provides the -mdaz-ftz (denormals are zero, flush to zero) flag for this purpose. It appears to be implemented entirely by including code that sets a hardware flag; constants are unaffected.)

   The situation on ARM is similar. However, RISC-V consistently requires subnormal support. The machine architects I talked to felt that there are now known implementation techniques that would keep any overhead acceptable. The fact that flush-to-zero is not consistently supported means that we cannot reasonably treat this as part of the rounding mode, even if we wanted to do that.

   Flush-to-zero or the like seems to be most commonly enabled by -ffast-math.

   This mode is ***not*** IEC 60559 conformant. The C standard also says “IEC 60559 arithmetic (with default exception handling) always treats subnormal numbers as nonzero”. Clang does not support it as an explicit flag, though -ffast-math may have a similar effect. (This seems to depend on installation details.) As one might expect, it is highly controversial. It is also brittle since it can apparently affect code not expecting to run in this mode. (See e.g. https://moyix.blogspot.com/2022/09/someones-been-messing-with-my-subnormals.html, which I found through a clang discussion.)

It is hard to make compiler-provided information about 60559 conformance accurate, given that behavior with respect to subnormals is controlled by a runtime flag. Gcc sets __STDC_IEC_559__ with -mdaz-ftz, but not with -ffast-math.

My current conclusion is that this is a bit of a mess, which we should largely try to dodge. A standard-conforming implementation should not claim to be 60559 compliant unless it can turn off flush-to-zero for those operations.

2. *The conversion specification was too simple. We cannot require exact conversions based on just size, for example bfloat16 vs. IEEE binary16.* The description below was changed to reflect this.
3. *We should consider leveraging the existing C++ `from_chars`.* We did not follow up on this. There appear to be two problems. First, since the function takes a reference to the result as a parameter, rather than returning the result, it is not a convenient way to build up constant expressions. Second, the existing floating point version is not `constexpr`, and is general enough that it may be difficult to make it so.
4. *Whatever mechanism is used to generate constants needs to be* `constexpr`*. It might be better to leave `constexpr` off other functions to facilitate implementation.* We left `constexpr` qualification on `cr_const` (now `make`) and on functions that we expect would be needed to implement `cr_const,` while removing it from the remainder. We weakened the minimum requirements for `cr_const` to make it more easily implementable as a library function, given constexpr basic arithmetic functions.
5. *We should consider whether the `cr_` naming for newly introduced functions is sufficiently consistent with the planned C functions with similar names.* Addressed by using a `cr` namespace instead, and later by switching to the `rounded` struct..

Kona 2023, SG6

1. It was suggested that ARM Neon doesn't support denormals at all. However, this appears to be contradicted by the ARM documentation, which claims: “Support is provided in AArch64 NEON for double-precision floating-point and full IEEE754 operation including rounding modes, denormalized numbers, and NaN handling.”
2. We should decide whether the emphasis here is on “correct rounding” or just directed rounding. Based on the few usages we've found, I think it's mostly the latter. When the user wants to achieve a specific outcome, rather than random perturbation, it's generally to bound the true results, as in interval arithmetic. Changed name appropriately.
3. Several alternate API styles were suggested. We converted the proposal to use one suggested by Davis Herring. I don't have a strong preference, and this feels perhaps slightly inconsistent with saturated arithmetic, but it reads better to me than any saturated-arithmetic-style suffix I could think of.
4. Two other operations were suggested: multiply-add (fma) and conversion to strings. The latter would preserve the bounding property all the way to the final displayed result, glibc currently applies the rounding mode during output for this purpose.

R1 changes: Some corrections to explanatory text. Reflected some of Jim Thomas' WG14 comments. Focussed on the free function approach favored by SG6, and outlined a more specific API.

R2 changes: Attempted to minimally remedy the BSI observation that `rint()` and friends were missing. This is probably one of the more common uses of `fenv` rounding modes.

R3 Changes: Integrate Varna comments. Replace `cr_` names with a `cr` namespace to avoid possible conflicts with the C `cr_` reserved names. Address/dodge the flush-to-zero issue, which I don't think can be handled well. Fixed the `cr::cast()` spec. . Avoid constexpr where it would unnecessarily complicate minimal implementations. Added the missing rounding mode argument to cr::make. Changed that from `consteval` to `constexpr` to better match the new name. And it's no doubt useful as a runtime function.

R4 Changes: Added `fma()` and `to_chars()`. Switched to Davis' suggested `struct`-based API formulation. Be clearer that our first, and only absolutely guaranteed, goal is to provide a way to bound the true result. Our secondary goal is to expose the rounding guarantees provided by the IEEE standard, when a suitable hardware implementation is in use.

Introduction

Currently floating point rounding modes are specified either dynamically, through the floating-point environment via `fesetround()` or, in C, statically, via `#pragma STDC FENV_ROUND`.

Searching through some large repositories of mostly nonnumerical code (e.g. on `cs.android.com`) we found many `fesetround()` implementation and test code, and some code to work around the existence of rounding modes, but very few real use cases. It appears to be common to encapsulate actual uses in a very small number of libraries. Earlier SG6 discussion, and the limited support in C++ (see below) also suggests that real uses are relatively rare, but more data would be helpful.

The use cases we have identified:

1. Interval arithmetic (e.g. the CGAL library) that's used to bound the true result, or other uses where it is necessary to get an accurate upper or lower bound on the true result But such code appears to be uncommon. As we discuss below, `<fenv.h>` seems to be a particularly bad match for this. That's especially true in C++, since it inherited the facility from C, but did not keep up with recent improvements to the C standard. But even in C, its utility seems questionable.

2. As a way to specify the additional rounding mode input to the `rint()` function family. In retrospect, this seems like an (ancient) historical design mistake. The client code I managed to find generally seemed to ignore the dependency on the rounding mode, and

it seems far more difficult to call these functions correctly than it would be with a rounding mode parameter.

3. As Jim Thomas points out, you could try to get some idea of floating point errors by trying with different rounding modes. (See http://people.eecs.berkeley.edu/~wkahan/Mindless.pdf) But, by the same argument as (3) below, that seems to be at best a little better than randomly perturbing the results. And the whole approach can break either because the program logic relies on rounding mode, or if the code itself sets the rounding mode. I believe it fails completely for (x - y) - (y - x).

Although this facility appears to have very limited use, it does seem to cause a disproportionate amount of trouble, notably in the context of `constexpr` math functions. Effectively all floating point functions have an implicit rounding mode argument, set by `fesetround()`, whose value is not predictable by the compiler, confounding optimizations expected by users. In C++, it is unclear whether math libraries should respect rounding modes, or even do anything reasonable in a nonstandard rounding mode.

This approach results in many problems complicating use:

1. Implementations can't be counted on to implement it correctly. Without the `FENV_ACCESS` pragma, compilers perform optimizations that do not preserve rounding behavior. That `fenv.h` pragma is explicitly not required to be supported in C++, and hence C++ provides no guarantee that `fesetround()` behaves reasonably. (See the first note in cfenv.syn]).
2. In practice, it seems to be inconsistent, and somewhat unpredictable, what calls to standard math functions, and even more so, user-defined functions, do when invoked with non-standard rounding modes. (This is based on looking at some implementations. I didn't study this systematically.)
3. If you really need to control rounding to guarantee specific properties of the result, rounding modes cannot, in general, just be specified for a region of code; they need to be carefully applied to each operation. What does it mean to set the rounding mode for `cos(a + b)?` It does make sense to specify a single rounding mode if every operation in the block is monotone increasing in every argument, or for very carefully designed code (e.g. https://hal.inria.fr/hal-03721525.) But this is not the general case.
4. Compilers seem to commonly ignore rounding modes for compile-time evaluations, making it very hard for the programmer to predict what they are actually getting. (This is my reading anyway, though the current C standard seems to say otherwise?)
5. If we really wanted to use rounding modes to bound results, constants would also have to be rounded according to the current rounding mode. This is explicitly disallowed, even by C, before C23: "All floating constants of the same source form(79) shall convert to the same internal format with the same value." C23 requires that static rounding modes affect constants. But this raises new issues:

   a. “Constant” macros can change meaning depending on the rounding mode context in which they are used. If the macros include arithmetic operations like subtraction or division, that change in meaning may be very different from any expectation.
   b. The same syntactic type, like char[(int)1.99999999999999999999] denotes different types, depending on the rounding mode.
   c. A “negative constant” like -0.1 is rounded in the opposite direction of the one requested, since the minus sign is not part of the constant.
6. Especially in C++, it is unclear what the rounding modes mean for operations that are not correctly rounded to start with. Although IEEE requires correct rounding, few commonly-used standard library implementations conform.

C’s `FENV_ROUND` pragma is, in our opinion, an improvement, but not enough to actually make it very useful. Most of the above points still apply in some form.

Discussions preceding N2746R0

A prior email discussion of d1381r1 suggested introducing correctly rounded math functions, with an explicit rounding argument for the rare occasions on which explicit rounding modes are useful. I think that's a much better replacement, even if the set of such functions is minimal. This is largely the approach we pursue here.

Matthias Kretz points out that there have been prior rounding-related proposals to WG21:

* N2899 Directed Rounding Arithmetic Operations (Revision 2) by G. Melquiond,
S. Pion (2009-06-19) (older revisions: N2876 and N2811). This proposes free functions with explicit rounding modes, and constant suffixes, roughly along the lines we propose here. The major differences are the fact that we choose nominally run-time parameters instead of template arguments for the rounding modes.
* P0105R1 Rounding and Overflow in C++ by Lawrence Crowl (2017-02-05) discusses some current narrower rounding issues, for both integers and floating point, and suggests some templatized free functions to explicitly control rounding and overflow handling.

Replacement proposal

We propose to deprecate the `fesetround()` and `fegetround()` functions, moving any mention of them, and the associated Note 1 describing any use as implementation-defined, to Appendix D.

Given the long history of this facility, even in spite of its sparse use, we do not expect this to be acceptable without a replacement facility. Given the infrequent use of the current facility, our inclination is to strive for a minimalist, but extensible, replacement facility. The rest of this document focuses on such a replacement.

It is not entirely clear to us what the various rounding modes should mean in the absence of correctly rounded arithmetic. In our experience, the most common rigorous and occasionally essential use of specified rounding modes is to bound true results, and we take that as our primary goal.

Recent versions of the IEEE floating point arithmetic standard basically require correct rounding. Thus, as suggested in previous discussions, we also propose to expose this guarantee to C++ programmers using IEEE conforming hardware arithmetic, in conjunction with explicitly specified rounding modes..

The design of this facility requires us to resolve several questions that do not have 100% clear answers. We make the following calls, based on SG6 discussions:

1. We provide an API similar in spirit to the appropriately named free functions on the existing floating point types from N2899. This is simplest, but seems error-prone to use, in that it is easy to accidentally apply a non-correctly-rounded conversion, or even arithmetic operation as part of a computation that is intended to use e.g. a directed rounding mode. Given the low observed usage rate of the current API, it was not felt that we should complicate the proposal to address this issue.
2. N2899 suggests passing the rounding mode as a template argument. This avoids some compile time overhead of optimizing out the runtime parameter in the usual case. SG6 felt that, unlike the `<atomic>` API, where it is a known issue, all indications are that rounding mode APIs are very rarely used, so this should not be a major issue. Passing it as a regular argument is a bit simpler, and makes it easier to port this facility to C, should that be desired at some point.
3. In response to a suggestion by Davis Herring, we actually encapsulate the “free functions” in a `struct` holding the rounding mode. This appears to be a slightly simpler syntax, and allows easy programmer-introduced abbreviations. The `struct` still does not hold an actual number; it merely encapsulates the rounding mode for syntactic convenience.

In general, based on both SG6 discussions, and our concern that this proposal will not be as widely used as many others, and hence may have trouble motivating implementers, we try to keep this proposal as small and focussed as possible. For the same reason, we limit the guarantees made to the programmer in this version to the minimal useful set. We do try to avoid losing functionality relative to current `fesetround` implementations.

If this API becomes more popular than expected, future proposals will will have to fill in the holes and possibly strengthen guarantees.

We thus propose just the following struct, with a quite modest number of member functions. Actual wording and syntax checking is still left as future work:

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

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

   // Do these functions fully conform to IEC 60559?
   // This should be false if subnormals are not supported.
   // Since that may not be known until runtime, it is not constexpr.
   // rounded() is useful for enforcing strict IEEE evaluation rules when
   this
   // yields true, or when the arithmetic is somehow known to have similar
   // properties, but probably not otherwise.
   template<floating_point F>
   static bool conforms_to_iec_60559();

   // 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);

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

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

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

   //
   template<floating_point F>
   F fma(F x, F y, F addend);

   // 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);

    // 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 s);

    // 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.
    // Additional precision guarantees are implementation-defined.
    // It is recommended practice that round_to_nearest produces the
    closest
    // decimal representable number obeying the formatting constraint,
    // resolving ties toward even.
    template<floating point F>
    to_chars_result
    to_chars(char* first, char* last, F value, chars_format fmt, int
    precision);

    template<floating_point F>
    F sqrt(F x, float_round_style r = round_to_nearest);

    // 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);

    // 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 `rounded::conforms_to_iec_60559()` yields true, then all provided `rounded` operations (except possibly `to_chars`) provide correctly rounded IEEE-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.
3. In the presence of a deprecated `fesetround()`, `fesetround()` calls will not affect `rounded::` operations.
4. Floating-point exceptions will be handled as they are now, i.e. via `fegetexcept()`.

We should also provide implementation advice recommending that implementations that conform to (IEC 60559 except for subnormals) follow IEEE rounding rules in all other respects.

A note on implementation:

For machine architectures that expect the rounding mode to be specified in floating point control register rather than in each instruction, we expect that implementations supporting `fesetround()` will need to set the control register before every sequence of `rounded` operations, and reset it to the original value when done. Normal floating point operations will continue to behave as they do now, even in the presence of `fesetround()`. If `fesetround()` is unsupported, we expect that the control register will normally indicate to-nearest rounding, and be adjusted only when a different mode is required.

Questions

When should `to_chars` produce correctly rounded results? The current `to_chars` spec seems strange to me, in that it has a strong round-trip requirement if no precision is specified, but probably no such requirement if a large precision is specified. (It is not 100% clear if any of that wording applies if a precision is specified.) For the sake of easy implementability, I'm tempted to only require the stated bound properties, even if `rounded::conforms_to_iec_60559()` yields true, and leave the rest implementation specified, with recommended practice suggesting that the implementation do the right thing.

Does `round_to_nearest` in the current standard imply anything about ties? [charconv.to.chars]p2 seems to imply that, but I couldn't find such a statement.

Are we in sufficient agreement about the API that it makes sense to draft actual wording?

Can we make floating point exception retrieval guaranteed reliable, at least for the `rounded::` functions, without mandating support for the `FENV_ACCESS` pragma? Should we add variants that return a pair?