Deprecate and Replace Fenv Rounding Modes

Doc. No: P2746R0
Contact: Hans Boehm hboehm@google.com)
Audience: SG6, LEWG
Date: Dec. 15, 2022
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 an initial draft proposal for feedback on the idea and the approaches taken by the replacement proposal.

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`.

As far as we can tell, `fesetround()`, in spite of its long history, has very few real use cases, and those are either not fully correct, or not portable across common implementations. There are many reasons for this:

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..
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 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)`?)

4. As Jim Thomas points out, you could try to get some idea of floating point errors by trying with different rounding modes. But, by the same argument as (3), 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.
5. 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?)
6. 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: "All floating constants of the same source form(79) shall convert to the same internal format with the same value."
7. 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, basically no 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.

As far as we can tell, these facilities are thus very rarely used in practice. The canonical use case seems to be interval arithmetic, 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, and `<fenv.h>` seems to be a particularly bad match. 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.

Although this facility does not appear to be very useful, 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 desetround(), 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.

Prior discussions

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 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, and separate correctly-rounded types. (Both choices are debatable, and not very strongly held positions for us; see below.)

* 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 suggest 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 addition, recent versions of the IEEE floating point arithmetic standard basically require correct rounding. Thus, as suggested in previous discussions, we propose to introduce a mechanism for generating correctly rounded IEEE-conforming floating point results. These would then also allow the explicit specification of rounding modes.

The design of this facility raises several questions that do not have clear (to us) answers. A major aim of this proposal is to solicit opinions on these issues:

1. We could provide appropriately named free functions on the existing floating point types as in 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. We thus lean towards introducing new types of correctly rounded floating point numbers.
2. By the same argument, we propose to avoid implicit conversions between correctly-rounded and regular floating-point values.
3. It is unclear to us whether the resulting arithmetic functions should be free functions or member functions of the correctly-rounded types. We somewhat arbitrarily chose the latter for now.
4. We originally suggested passing the rounding mode as a nominally runtime parameter, as with `memory_order` arguments to atomic operations. Richard Smith pointed out that this has proven to be a bit of a compile time issue for atomics. For simplicity, we thus

initially propose purely static template rounding-mode arguments. In the (very rare?) cases in which dynamic arguments are necessary, the user could define the dynamic version in terms of a `switch` statement.

We thus, very tentatively, end up with a class along the following lines: (Actual wording and syntax checking is left as future work)

```
template<floating_point F>
class correctly_rounded {
  explicit correctly_rounded(F plain_value);
  // Other expected constructors, destructor, assignment;
  F to_plain() const;
  …
  correctly_rounded<F> operator+(correctly_rounded<F> y) const;
  template<float_round_style r = round_to_nearest>
    correctly_rounded<F> add(correctly_rounded<F> y) const;

  correctly_rounded<F> operator*(correctly_rounded<F> y) const;
  template<float_round_style r = round_to_nearest>
    correctly_rounded<F> multiply(correctly_rounded<F> y) const;
  …

  template<float_round_style r = round_to_nearest>
  correctly_rounded<F> sqrt() const;
  …
}

correctly_rounded<double> operator “” _d_round_to_nearest(const char
*);
correctly_rounded<double> operator “” _d_round_toward_infinity(const
char *);
…
// We may want to add assert_exact as another rounding mode here?
```

Semantics:

We propose to require that:

1. All provided operations on correctly-rounded numbers are fully conforming to the corresponding IEEE type.
2. All provided operations are truly correctly rounded, even in cases in which that might be difficult to implement. We do not add functions to correctly-rounded until we are convinced that practically useful implementations with this property are feasible. This is already the case for many more functions than are listed here, at least for

`correctly_rounded<float>`. See for example, Lim and Nagarakatte, "High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations".

3. In the presence of `fesetenv()`, `fesetenv()` will not affect `correctly_rounded<>` operations.

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 `fesetenv()` will need to set the control register before every sequence of `correctly_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 `fesetenv()`. If `fesetenv()` is unsupported, we expect that the control register will normally indicate to-nearest rounding, and be adjusted only when a different mode is required.