Document Number: P3488R0
Date: 2024-11-14
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: SG6, EWG
Target: C++26

Floating-Point Excess Precision

ABSTRACT

CWG2752 asks whether a conforming implementation can represent a floating-point literal with excess precision. This issue was opened after GCC implemented excess precision for C++. Notably, GCC also uses excess precision for evaluation at compile-time as shown in this paper. For a holistic answer this paper considers excess precision of constants and in evaluation. Therefore, the main question we need answered is whether literals must be rounded or can be stored with excess precision. The secondary question is the use of excess precision in constant expressions and in compile-time evaluation of floating-point operations. The goal is to find a consensus on what the design intent should be, without breaking performance or correctness requirements of C++ users. This paper lists possible design intent and discusses their implications on potential optimizations.

CONTENTS

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

3 INTRODUCTION

This paper tries to take a holistic approach at the questions around excess precision in C++. As such it is not constrained to resolving only the issue described in CWG2752.

The following issues are considered:

- CWG2752: can the value of a floating-point literal be stored with excess precision?
- A library clause, especially a macro inherited from C, should not add constraints to the core language. We need to ensure that the library wording simply allows reflecting on implementation choices of the core wording.
- Can floating-point expressions use higher intermediate precision (and range) at compile-time? Or, in other words, does `FLT_EVAL_METHOD` apply only to runtime evaluation?
- The language allows greater intermediate precision and range without constraints, but `FLT_EVAL_METHOD` constrains it to `double` and `long double`. This makes evaluating `std::float16_t` and `std::bfloat16_t` in intermediate precision of `std::float32_t` impossible. An implementation would have to use `double` and then evaluate `float` in `double` precision and range.

(This is an extended/modified copy of CWG2752.)

Consider:

```
int main()
{
  constexpr auto x = 3.14f;
  assert( x == 3.14f );          // can fail?
  static_assert( x == 3.14f );   // can fail?
}
```

Can a conforming implementation represent a floating-point literal with excess precision, causing the comparisons to fail?

Subclause 5.13.4 [lex.fcon] paragraph 3 specifies:

C++ [lex.fcon]

3 If the scaled value is not in the range of representable values for its type, the program is ill-formed. Otherwise, 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.

This phrasing leaves little leeway for excess precision. In contrast, C23 specifies:

ISO/IEC 9899:2024 6.4.4.3 Floating constants

6 The values of floating constants may be represented in greater range and precision than that required by the type (determined by the suffix); the types are not changed thereby. See 5.2.5.3.3 regarding evaluation formats.[1]

Subclause 7.1 [expr.pre] paragraph 6 uses very similar wording to allow excess precision for floating-point computations (including their operands):

C++ [expr.pre]

6 The values of the floating-point operands and the results of floating-point expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.[2]

Taken together, that means that `314.f / 100.f` can be computed and represented more precisely than `3.14f`, which is hard to justify. The footnote appears to imply that `(float)3.14f` is required to yield a value with `float` precision, but that conversion (eventually) ends up at 9.4.1 [dcl.init.general] bullet 16.9:

C++ [dcl.init.general]

[...] Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. [...]

---

1 Hexadecimal floating constants can be used to obtain exact values in the semantic type that are independent of the evaluation format. Casts produce values in the semantic type, though depend on the rounding mode and may raise the inexact floating-point exception.

2 The cast and assignment operators must still perform their specific conversions as described in 7.6.1.4 [expr.type.conv], 7.6.3 [expr.cast], 7.6.1.9 [expr.static.cast] and 7.6.19 [expr.ass].

If values produced from literals were permitted to carry excess precision, this phrasing does not seem to convery permission to discard excess precision when converting from a `float` value to type `float` ("[...] is the value [...]"), apparently requiring that the target object's value also carry the excess precision.

However, if initialization is intended to drop excess precision, then an overloaded operator returning `float` can never behave like a built-in operation with excess precision, because returning a value means initializing the return value.

The C++ standard library inherits the `FLT_EVAL_METHOD` macro from the C standard library. C23 specifies it as follows:

ISO/IEC 9899:2024 5.2.5.3.3 Characteristics of floating types `<float.h>`

26 The values of floating type yielded by operators subject to the usual arithmetic conversions, including the values yielded by the implicit conversion of operands, and the values of floating constants are evaluated to a format whose range and precision may be greater than required by the type. Such a format is called an evaluation format. In all cases, assignment and cast operators yield values in the format of the type. The extent to which evaluation formats are used is characterized by the value of `FLT_EVAL_METHOD`:

- -1 indeterminable;
- 0 evaluate all operations and constants just to the range and precision of the type;
- 1 evaluate operations and constants of type `float` and `double` to the range and precision of the `double` type, evaluate `long double` operations and constants to the range and precision of the `long double` type;
- 2 evaluate all operations and constants to the range and precision of the `long double` type.

All other negative values for `FLT_EVAL_METHOD` characterize implementation-defined behavior. The value of `FLT_EVAL_METHOD` does not characterize values returned by function calls (see 6.8.7.5, F.6).

Taken together, a conforming C++ implementation cannot define `FLT_EVAL_METHOD` to 1 or 2, because literals (= "constants") cannot be represented with excess precision in C++.

3.1 Annex H of C23

Annex H of C23 "specifies extension types for programming language C that have the arithmetic interchange and extended floating-point formats specified in ISO/IEC 60559".

This annex modifies `FLT_EVAL_METHOD` and is relevant with regard to discussion around evaluation of e.g. `std::float16_t` operations:

ISO/IEC 9899:2024 H.3 Characteristics in `<float.h>`

2 If `FLT_RADIX` is `2`, the value of `FLT_EVAL_METHOD` (5.2.5.3.3) characterizes the use of evaluation formats for standard floating types and for binary floating types:

- `-1` indeterminable;

0 evaluate all operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `float`, to the range and precision of `float`; evaluate all other operations and constants to the range and precision of the semantic type;

1 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `double`, to the range and precision of `double`; evaluate all other operations and constants to the range and precision of the semantic type;

2 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `long double`, to the range and precision of `long double`; evaluate all other operations and constants to the range and precision of the semantic type;

N where `_Float`N is a supported interchange floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `_Float`N, to the range and precision of `_Float`N; evaluate all other operations and constants to the range and precision of the semantic type;

$N+1$ where `_Float`N`x` is a supported extended floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `_Float`N`x`, to the range and precision of `_Float`N`x`; evaluate all other operations and constants to the range and precision of the semantic type.

3.2 Relevance of this issue

This issue should be irrelevant for all environments where `FLT_EVAL_METHOD` is `0`. An example environment where `FLT_EVAL_METHOD` is non-zero is GCC compiling with `-m32` or `-mfpmath=387`. With GCC 13 or later and one of the mentioned compiler flags and e.g. `-std=c++23` the above code example fails both the `static_assert` and the runtime `assert`[3].

An example that exhibits different behavior for constant propagation / expressions can also be constructed[4]:

```
constexpr float a = 0x1.000003p0f; // this rounds to nearest
static_assert(a ==  0x1.000004p0f); // as expected

constexpr float b = 0x2.000005p0f; // this rounds to nearest
static_assert(b ==  0x2.000004p0f); // as expected

constexpr float b0 = 0x1.000002p0f + 0x1.000003p0f;
// -> without intermediate rounding: 0x2.000005p0f
//           -> subsequent rounding: 0x2.000004p0f (A)
// ->    with intermediate rounding: 0x2.000006p0f (B')
//           -> subsequent rounding: 0x2.000008p0f (B)
```

3 https://compiler-explorer.com/z/vrYoT5cer
4 https://compiler-explorer.com/z/5KGoebo75

```
static_assert(b0 != 0x2.000004p0f); // (A)
static_assert(b0 == 0x2.000006p0f); // (B')
static_assert(b0 == 0x2.000008p0f); // (B)

constexpr float b1 = 0x1.000002p0f + a;
// same constants as 'b0' except rounding for 'a' is required
// -> 0x2.000006p0f -> subsequent rounding: 0x2.000008p0f
static_assert(b1 == 0x2.000008p0f);

constexpr float b2 = 0x1.000002p0f + a - 1.f;
// 0x2.000006p0f - 1 -> 0x1.000006p0f (C)
// 0x2.000006p0f rounds to 0x2.000008p0f -> subtract 1 -> 0x1.000008p0f (D)

static_assert(b2 != 0x1.000006p0f); // (C)
static_assert(b2 == 0x1.000008p0f); // (D)

constexpr float third = 1 / 3.f;
constexpr float five_third = 5 * third;
constexpr float five_third_ = 5 * (1 / 3.f);
static_assert(five_third == five_third_); // (E)
```

All of these static assertions hold on GCC, Clang, and MSVC as far as I tested them, except when compiling with GCC 13 (and up) and the `-m32` flag (targeting 32-bit x86). There, the assertions marked `(A)`, `(B')` `(B)`, `(C)`, `(D)`, and `(E)` fail. This is due to `FLT_EVAL_METHOD == 2` which GCC interprets as allowing / requesting constants in `long double` precision.

4 A PLAN ON HOW TO REACH A CONCLUSION

Three steps:

1. SG6 documents possible design intent and their implications. The group then makes a recommendation on how the issue should be resolved. Irrespective of whether a consensus is reached, the paper then progresses to EWG.

2. EWG does what it does. Most importantly EWG is the group that has the final say in how this issue is resolved.

3. CWG.

5 CHOOSE A DESIGN INTENT

This section only explains the options. In other words, we want to be able to choose one of these and say "this is the design intent". A discussion of the options follows in the next section.

5.1 Strictest: disallow all excess precision

- `FLT_EVAL_METHOD` must always be `0`.

expr.pre must disallow greater precision / range in floating-point expressions.

- Any evaluation of a floating-point expression applies a single rounding to the precision of the floating-point type after each operation.

→ Discussion

5.2 Compatible: do exactly the same as C

lex.fcon must allow representing floating-point constants in greater range and precision.

- Evaluation of constant expressions and compile-time evaluation of expressions may use excess precision.
- Intermediate rounding in runtime and compile-time evaluation is reflected by `FLT_EVAL_METHOD`.

→ Discussion

5.3 Like C but only for runtime evaluation

- The value of a floating-point literal is always rounded to the precision of its type (status quo of [lex.fcon]).
- Evaluation of floating-point expressions at compile-time is not allowed to use excess precision.
- `FLT_EVAL_METHOD` only reflects on runtime evaluation of floating-point expressions.
- Floating-point evaluation at runtime can use (arbitrary) greater precision and range and is only required to round to the precision and range of the floating-point type on cast and assignment. The intermediate precision is exposed to the program via `FLT_EVAL_METHOD`.
- We should consider adding a note to [expr.pre] saying that while excess precision in evaluation is allowed, it is only allowed for performance reasons and it is preferred that intermediate precision and range match the floating-point type.

→ Discussion

6 DISCUSSION

A general observation: A simplification where the implementation were free to use excess precision at runtime as it deems best would lead to suprising results: Consider two floating-point values `a` and `b` where `std::isfinite(b)` is statically known to be `true`. With arbitrary excess precision the optimizer would then be allowed to replace `a + b - b` with `a`.

A general consequence of excess precision is that floating-point evaluation leads to double rounding and thus potentially worse errors. Where the second rounding occurs is not fully reproducible and can potentially change via unrelated code changes in the translation unit[5].

Without excess precision `std::float16_t` and `std::bfloat16_t` can either use a soft-float implementation or dedicated hardware is required. Using `float` (binary32) instructions is impossible with the current possible values for `FLT_EVAL_METHOD`. An implementation that wants to evaluate `std::float16_t` / `std::bfloat16_t` in higher intermediate precision needs to set `FLT_EVAL_METHOD` to 1 or 2 (or 32?).

6.1 STRICTEST: DISALLOW ALL EXCESS PRECISION

I believe [expr.pre] p6 is fairly clear that it was never the design intent to exclude all excess precision.

Implications of disallowing all excess precision:

- Floating-point contraction into FMAs is non-conforming.
- The x87 FPU cannot be used with a single "precision control" value, because double rounding is not correct (e.g. FPU configured to 80-bit with subsequent rounding to 64/32-bit). This implies that the compiler would have to set the x87 floating-point control word (FPCW) using the FLDCW instruction whenever it needs to execute floating-point operations (with different precision).
- This is likely an ABI break and unacceptable for existing implementations.

6.2 COMPATIBLE: DO EXACTLY THE SAME AS C

It might have been the original intent to do the same as C, but [lex.fcon] p3 suggests otherwise.

Implications of adopting this as resolution:

- `float x = 3.14f;` can require 8, 12, 16, or even more bytes to be stored in the resulting binary. (This is the status quo of GCC since version 13.)
- `float x = 3.14f; assert(x == 3.14f);` is allowed to fail depending on implementation, target, and compiler flags. (This is the status quo of GCC since version 13.)

5 e.g. because of register allocation

6.3 LIKE C BUT ONLY FOR RUNTIME EVALUATION

- The intent here appears to be that we want to prescribe reproducible floating-point behavior.
- However, since that has potentially dramatic consequences on runtime performance, this restriction is only a recomendation for runtime evaluation. We thus acknowledge the existence of hardware where reproducible floating-point behavior comes at unreasonable performance cost. Because of these cases — and only for these — [expr.pre] allows excess precision in evaluation, which should be reflected by non-zero `FLT_EVAL_METHOD`.
- We should consider a new type trait along the lines of

```
template <floating-point T>
struct evaluation_type {
  using type = see below;
};

template <floating-point T>
using evaluation_type_t = typename evaluation_type<T>::type;
```

  Where e.g. `evaluation_type_t<float16_t>` could be `float`. This would supersede the use of the `FLT_EVAL_METHOD`. Implementations could then reasonably set `FLT_EVAL_METHOD` to `-1` and rely solely on the traits for reflection of floating-point evaluation behavior.

7 FLOATING-POINT CONTRACTION

Floating-point contraction is the transformation of `a * b + c` into `std::fma(a, b, c)`. This effectively increases the intermediate precision and range of the multiplication result. Thus, floating-point contraction is related to this discussion. [expr.pre] p6 appears to allow floating-point contraction.

ISO/IEC 60559:2020 specifies

ISO/IEC 60559:2020 10.4 Literal meaning and value-changing optimizations

A language standard should also define, and require implementations to provide, attributes that allow and disallow value-changing optimizations, separately or collectively, for a block. These optimizations might include, but are not limited to:

- Applying the associative or distributive laws.
- Synthesis of a **fusedMultiplyAdd** operation from a **multiplication** and an **addition**.
- Synthesis of a *formatOf* operation from an operation and a conversion of the result of the operation.
- Use of wider intermediate results in expression evaluation.

The fourth item is what this paper has been discussing so far.

The second item is considered a different optimization in the 60559 standard. Therefore, we should also consider floating-point contraction separately from `FLT_EVAL_METHOD`. It is unclear what the original intent for floating-point contraction for C++ had been. Existing practice is to default to floating-point contraction as an optimization independent of `FLT_EVAL_METHOD`. Therefore, I suggest we ensure the wording matches existing practice.

Note that the 60559 wording talks about "attributes that allow and disallow value-changing optimizations". C++ does not provide such attributes. However, implementations typically provide them (e.g. as compiler flags treating one complete translation unit as a "block", but also as vendor attributes that can be applied to functions). This appears to follow the guidance in 60559 which says that if a language standard doesn't define something it is implementation defined.

Consequently, I'd be wary of making floating-point contraction non-conforming. Rather we want to keep it as a conforming optimization and (for now) continue to trust the implementations to provide the necessary "attributes" to control floating-point contraction. Adding such an "attribute" to C++ itself (and possibly adding a trait to determine whether floating-point contraction should be expected) is material for another paper.

7.1 Guaranteed opt-out of floating-point contraction

It appears that accoding to the footnote of [expr.pre] p6 the expression `a * b + c` can be transformed into an FMA, whereas `auto(a * b) + c` cannot. Likewise `auto ab = a * b; ab + c` would not lead to floating-point contraction.

It is unclear whether a simple floating-point wrapper class would inhibit floating-point contraction:

```
class Float
{
  float x;

public:
  Float(float xx) : x(xx) {}

  friend Float operator+(Float a, Float b) { return a.x. + b.x; }
  friend Float operator*(Float a, Float b) { return a.x. * b.x; }
};

Float test(Float a, Float b, Float c)
{ return a * b + c; } // is contraction allowed or not?
```

The copy constructor of `Float` implicitly assigns to the data member `x`. But there is no assignment or cast expression. The return statements in the binary operators of `Float` call the `Float(float)` constructor which copies the `float` into `xx` and subsequently into `x`. Both copies are neither using a cast not assignment expression. Consequently this wrapper class would still allow floating-point contraction, correct?

With a minor change to the `Float(float)` constructor to

```
Float(float xx) : x(float(xx)) {}
```

floating-point contractions would be inhibited.

I believe we need to clarify whether this matches the intent and at least add a note in the wording to explain this subtlety.

8 WORDING

TBD. But here's at least a sketch if we agree on adopting 5.3:

1. Clarify [expr.pre] that it only provides this freedom for runtime evaluation.
2. Clarify [expr.pre] that floating-point contraction is a conforming transformation for runtime evaluation (but not required)
3. Add the above `Float` class example to [expr.pre]?
4. Stop inheriting `FLT_EVAL_METHOD` verbatim from C. We need to write our own wording that clarifies `FLT_EVAL_METHOD` only applies to runtime evaluation and not to constants. Also we need to consider adopting and adjusting the wording from Annex H, which is important for `std::float16_t` and `std::bfloat16_t`.