Directed Rounding Arithmetic Operations (Revision 2)

WG21 Document : N2899=09-0089

Guillaume Melquiond and Sylvain Pion*

2009-06-19

Abstract

This proposal is addressed to both the Core Working Group and the Library Working Group, since it proposes a library-like user interface, but its implementation requires special compiler support.

We propose the addition of new functions to the C++0x standard library that provide floating-point operations (`+`, `-`, `*`, `/`, `sqrt` and `fma`), conversion functions (between floating-point types, and between floating-point types and integral types), as well as and user-defined-literals, with *directed rounding*. These functions correspond to the IEEE-754 specifications, and are necessary to provide efficient support for interval arithmetic and related certified computations. These functions require special compiler support due to their `constexpr` nature.

Changes since N2876

- Replaced `decltype(t+u)` in the return types by `HasPlus::result_type` as the `FloatingPointType` concept does not provide `+` (a concept mirroring `std::common_type` would probably be ideal here).

Changes since N2811

- Added constexpr user-defined-literals that extend the floating-literals with a rounding direction.
- Overflow and underflow can occur in a `directed_cast` to an integral type : make the result unspecified in this case (like `lrint`).
- Made the whole header conditionally supported, and made the implementation define the macro `__STDC_DIRECTED_FP__` in this case.
- Moved all functions to the `std::directed_fp` sub-namespace.
- Renamed `rounded_cast` to `directed_cast`.
- Renamed header <`rounded_math`> to <`directed_fp`>.
- Changed the uses of the `FloatingPointLike`, `IntegralLike` and `ArithmeticLike` concepts to `FloatingPointType`, `IntegralType` and `ArithmeticType`.

*Email : `<Guillaume.Melquiond@inria.fr>`, `<Sylvain.Pion@sophia.inria.fr>`.

- Added the wording changes that reference the new section at the beginning of Chapter 26 [Numerics library].
- Added more motivations (cite Pete Becker's book, and mention compilers default optimization options).

Motivation and context

WG21 has chosen to postpone the consideration of the proposal to add interval arithmetic to the standard library, N2137[1], after C++0x. While a new standard is being developed to provide a language-independent base for interval arithmetic (IEEE-1788), we believe that progress can be made to enhance low-level support for interval arithmetic in C++0x, and make it easier to build an interval arithmetic library on top of C++0x efficiently and portably.

The current CD provides the `fegetround` and `fesetround` functions in the `<cfenv>` header to access and change the current rounding mode. Those functions were imported from C99, and they are at the root of most interval arithmetic implementations. There are however two problems which prevent efficient support for interval arithmetic:

- the CD does not provide the accompanying `FENV_ACCESS` pragma, which instructs the compiler that a block of code cares about the rounding mode. The consequence is that one must use temporary volatile variables, or any other equivalent implementation-dependent method, which is inefficient. This can be considered as an incomplete addition to C++0x compared to C++03. This already leads to embarassments like the one mentioned in Pete Becker's book "The C++ Standard Library Extensions" on TR1, page 242, "If your compiler doesn't support this `pragma`, turn off optimizations". We do not advocate for adding the pragma, but we propose an alternate solution that will allow to turn on optimizations for many cases. We also believe that this proposal removes one more reason why one could still advocate the use of C99 instead of C++.
- building interval constants is very useful, but it is not possible to build `constexpr` interval operations since the restricted form of bodies allowed by `constexpr` functions forbids calling `fesetround`. Extending a general feature such as `constexpr` to support this, does not seem appropriate for this particular use.

Moreover, in practice, compilers also perform optimizations by doing transformations of floating-point expressions which are valid only when rounding is to the nearest (like `(-a)*(-b)` into `a*b`), which is by far the most commonly used rounding mode. Using specific functions to instruct the compilers precisely where the rounding is not the default, helps setting a more efficient default for optimization options (like setting `-fno-rounding-math` by default for GCC).

A different approach

We propose to go away from `fesetround` and the pragma as the low-level primitives used to implement interval arithmetic. Instead of a global state rounding mode, we propose to add functions with directed rounding modes that perform floating-point addition, subtraction, multiplication, division, square root and fma, as well as conversions between floating-point and integral types, and between floating-point types themselves. These functions correspond to the functionality mandated by IEEE-754-1985 (and by IEEE-754-2008 for fma).

These functions, especially as they would be `constexpr`, require specific compiler support.

On hardwares that support a static rounding direction in the floating-point operation instructions or that support several floating-point control registers, the new functions make it easier for

[1]`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2137.pdf`

the compiler to generate those instructions than currently, since using the `fesetround` interface around a block of floating-point operations requires some (possibly inter-procedural) flow analysis.

Fundamentally, there is no reason why C++0x provides built-in operators `+`, `-`, `*`, `/` which allow to build `constexpr` functions with the default rounding mode, while it would not be possible with its directed rounding cousins.

Note on optimization: on hardware that needs to set a register holding the rounding mode, using `fesetround` in a block of code can be more efficient. In practice however, this kind of computation is typically needed for adding vectors of intervals, and it is our belief that, in our approach, a quality compiler should be able to move the rounding mode change instructions outside of the loops.

Syntax and design choices

There are 4 rounding modes specified by IEEE-754-1985: to the nearest (default), towards zero, towards plus infinity and towards minus infinity.

We propose that the new functions use a constant rounding mode. That is, it would not be possible to pass it as an argument, and even less be a run time entity.

Moreover, C++0x currently has 2 ways of refering to rounding modes:

- `fesetround` takes an argument of type `int` with possible values `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO` and `FE_UPWARD`.
- the `floating_round_style` enum which has 4 values plus a fifth `round_indeterminate`.

We prefer the latter alternative for type checking and consistency reasons. Moreover, we suggest to pass the rounding mode as explicit template argument, to make sure it is a constant, while still not hardcoding it in the function name like `add_up` or `mul_down`. Passing `round_indeterminate` is not useful, and is not allowed.

An example of use would then be:

```
double d     = add<round_toward_infinity>(a, b);
float  pi_up = directed_cast<round_toward_infinity, float>(3.14159265358979323);
int    i     = directed_cast<round_toward_infinity, int>(17./3.);
```

Side note: some languages which support more mathematical symbols for operators, like Fortress[2], provide a compact syntax like `a` $\overset{+}{\triangle}$ `b`.

Directed rounding floating-literals

The current floating-literals do not even provide round-to-nearest. It is for example a challenge to define an accurate constant for π in a portable way, since one needs to give the exact decimal representation corresponding to a representable floating-point.

Moreover, in order to build interval constants, one would benefit from user-defined-literals with directed rounding.

We suggest the addition of constexpr user-defined-literals that extend the floating-literals with a rounding direction. These also need to be compiler-supported in order to be constexpr.

Example:

```
constexpr double pi_up   = 3.14159265358979323846264338327 9_round_toward_infinity;
constexpr double pi_down = 3.141592653589793238462643383279_round_toward_neg_infinity;
constexpr long double pi =
   3.14159265358979323846264338327950288419716939937510L_round_to_nearest;
```

[2]`http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`

Promotion rules

Ideally, the `add` function should behave just like the built-in floating-point `operator+`, concerning the types of the operands, following the promotion rules. And similarly for the other functions. We propose to use constrained templates, `auto` and `decltype` to achieve this.

Side effects

Division by zero, or taking the square root of a negative number, can produce side effects like raising floating-point exceptions and setting `errno`. Obviously, this cannot work for `constexpr` functions at compile-time. We therefore propose that the evaluation of these functions follow the IEEE-754 "default non-stop exception handling" mode (producing NaNs and Inf, but no floating-point exception nor `errno` change), and with the caveat that the compile-time evaluation is done without raising the corresponding status flags, especially not the inexact flag.

A compiler may choose to issue or not a warning in such cases, when the other status flags would be raised.

Namespace

Such simple and common function names may have the possibility to conflict with other uses. We have therefore put the functions under the `std::directed_fp` namespace.

Header file

The closest related headers are <`cmath`> and <`numeric`>. However, neither seem perfectly appropriate to provide these functions. Moreover, since these functions are compiler-supported, it makes sense to put them in a separate header file, so as to ease the work of library vendors. This header could then be provided by compiler vendors.

We therefore propose a new header file <`directed_fp`>, and a new section to describe it.

Conditionally supported functionality

The functionality is strongly linked to IEEE-754, which is conditionally supported in C++. For this reason, the whole header is conditionally supported, and a compiler macro helps detecting whether the functionality is available or not.

Acknowledgements

We thank Marc Glisse, Lawrence Crowl, Lee Winter, Daniel Krügler, and members of the LWG for their useful comments.

Proposed wording

Compared to the latest working draft, N2857, do the following changes:

Add the following line in 16.8/2 "Predefined macro names" [cpp.predefined]:

`__STDC_DIRECTED_FP__`

`__STDC_DIRECTED_FP__` is defined iff the header <`directed_fp`> is supported by the implementation.

Change the sentence 26.1/2 [numerics.general]:

The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, ~~and~~ facilities included from the ISO C library, **and directed rounding functions,** as summarized in Table 92.

Add the following line to Table 92 – Numerics library summary:

26.9	Directed rounding functions	`<directed_fp>`

Add the following section:

26.9 Directed rounding functions

1 The header `<directed_fp>` provides constexpr functions over floating-point, as well as conversions with integral types. These functions allow a rounding mode to be passed as template argument.

2 The header `<directed_fp>` is conditionally supported. The implementation defines the macro `__STDC_DIRECTED_FP__` iff it supports the header.

26.9.1 Header `<directed_fp>` synopsis

```
namespace std {
 namespace directed_fp {
  template < float_round_style r, FloatingPointType T, ArithmeticType U >
  requires True<(r != round_indeterminate)>
  constexpr T directed_cast(U u);

  template < float_round_style r, IntegralType T, FloatingPointType U >
  requires True<(r != round_indeterminate)>
  constexpr T directed_cast(U u);

  template < float_round_style r, FloatingPointType T, FloatingPointType U >
  requires True<(r != round_indeterminate)> && HasPlus<T, U>
  constexpr HasPlus<T, U>::result_type add(T t, U u);

  template < float_round_style r, FloatingPointType T, FloatingPointType U >
  requires True<(r != round_indeterminate)> && HasPlus<T, U>
  constexpr HasPlus<T, U>::result_type sub(T t, U u);

  template < float_round_style r, FloatingPointType T, FloatingPointType U >
  requires True<(r != round_indeterminate)> && HasPlus<T, U>
  constexpr HasPlus<T, U>::result_type mul(T t, U u);

  template < float_round_style r, FloatingPointType T, FloatingPointType U >
  requires True<(r != round_indeterminate)> && HasPlus<T, U>
  constexpr HasPlus<T, U>::result_type div(T t, U u);

  template < float_round_style r, FloatingPointType T >
  requires True<(r != round_indeterminate)>
  constexpr T sqrt(T t);

  template < float_round_style r, FloatingPointType T, FloatingPointType U,
             FloatingPointType V >
  requires True<(r != round_indeterminate)> && HasPlus<T, U>
```

```
          && HasPlus<HasPlus<T, U>::result_type, V>
  constexpr HasPlus<HasPlus<T, U>::result_type, V>::result_type fma(T t, U u, V v);

  constexpr float       operator "" f_round_to_nearest(const char* s);
  constexpr float       operator "" F_round_to_nearest(const char* s);
  constexpr double      operator ""  _round_to_nearest(const char* s);
  constexpr long double operator "" l_round_to_nearest(const char* s);
  constexpr long double operator "" L_round_to_nearest(const char* s);

  constexpr float       operator "" f_round_toward_zero(const char* s);
  constexpr float       operator "" F_round_toward_zero(const char* s);
  constexpr double      operator ""  _round_toward_zero(const char* s);
  constexpr long double operator "" l_round_toward_zero(const char* s);
  constexpr long double operator "" L_round_toward_zero(const char* s);

  constexpr float       operator "" f_round_toward_infinity(const char* s);
  constexpr float       operator "" F_round_toward_infinity(const char* s);
  constexpr double      operator ""  _round_toward_infinity(const char* s);
  constexpr long double operator "" l_round_toward_infinity(const char* s);
  constexpr long double operator "" L_round_toward_infinity(const char* s);

  constexpr float       operator "" f_round_toward_neg_infinity(const char* s);
  constexpr float       operator "" F_round_toward_neg_infinity(const char* s);
  constexpr double      operator ""  _round_toward_neg_infinity(const char* s);
  constexpr long double operator "" l_round_toward_neg_infinity(const char* s);
  constexpr long double operator "" L_round_toward_neg_infinity(const char* s);
 }
}
```

1 The compile-time evaluation of these functions ignores any floating-point status flags raised, if any.

2 The run-time evaluation of these functions shall not raise any floating-point exception, nor change `errno`.

```
template < float_round_style r, FloatingPointType T, ArithmeticType U >
requires True<(r != round_indeterminate)>
constexpr T directed_cast(U u);
```

Returns: The conversion of `u` to the type `T` rounded according to `r`.

```
template < float_round_style r, IntegralType T, FloatingPointType U >
requires True<(r != round_indeterminate)>
constexpr T directed_cast(U u);
```

Returns: The conversion of `u` to the type `T` rounded according to `r`. If no such value of type `T` exists, the result is unspecified.

```
template < float_round_style r, FloatingPointType T, FloatingPointType U >
requires True<(r != round_indeterminate)> && HasPlus<T, U>
constexpr HasPlus<T, U>::result_type add(T t, U u);
```

Returns: The addition of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointType T, FloatingPointType U >
requires True<(r != round_indeterminate)> && HasPlus<T, U>
constexpr HasPlus<T, U>::result_type sub(T t, U u);
```

Returns: The subtraction of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointType T, FloatingPointType U >
requires True<(r != round_indeterminate)> && HasPlus<T, U>
constexpr HasPlus<T, U>::result_type mul(T t, U u);
```

Returns: The multiplication of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointType T, FloatingPointType U >
requires True<(r != round_indeterminate)> && HasPlus<T, U>
constexpr HasPlus<T, U>::result_type div(T t, U u);
```

Returns: The division of `t` and `u` rounded according to `r`.

```
template < float_round_style r, FloatingPointType T >
requires True<(r != round_indeterminate)>
constexpr T sqrt(T t);
```

Returns: The square root of `t` rounded according to `r`.

```
template < float_round_style r, FloatingPointType T, FloatingPointType U,
           FloatingPointType V >
requires True<(r != round_indeterminate)> && HasPlus<T, U>
         && HasPlus<HasPlus<T, U>::result_type, V>
constexpr HasPlus<HasPlus<T, U>::result_type, V>::result_type fma(T t, U u, V v);
```

Returns: The fused multiply-and-add of `t`, `u`, and `v`, rounded according to `r`.

```
constexpr float       operator "" f_round_to_nearest(const char* s);
constexpr float       operator "" F_round_to_nearest(const char* s);
constexpr double      operator ""  _round_to_nearest(const char* s);
constexpr long double operator "" l_round_to_nearest(const char* s);
constexpr long double operator "" L_round_to_nearest(const char* s);

constexpr float       operator "" f_round_toward_zero(const char* s);
constexpr float       operator "" F_round_toward_zero(const char* s);
constexpr double      operator ""  _round_toward_zero(const char* s);
constexpr long double operator "" l_round_toward_zero(const char* s);
constexpr long double operator "" L_round_toward_zero(const char* s);

constexpr float       operator "" f_round_toward_infinity(const char* s);
constexpr float       operator "" F_round_toward_infinity(const char* s);
constexpr double      operator ""  _round_toward_infinity(const char* s);
constexpr long double operator "" l_round_toward_infinity(const char* s);
constexpr long double operator "" L_round_toward_infinity(const char* s);

constexpr float       operator "" f_round_toward_neg_infinity(const char* s);
constexpr float       operator "" F_round_toward_neg_infinity(const char* s);
constexpr double      operator ""  _round_toward_neg_infinity(const char* s);
constexpr long double operator "" l_round_toward_neg_infinity(const char* s);
constexpr long double operator "" L_round_toward_neg_infinity(const char* s);
```

Returns: The floating-point value corresponding to the string `s`, exactly like the basic floating-literals [2.14.4, lex.fcon], but with the ability to specify a rounding direction.