WG21/N2088 J16/06-0158 2006-09-07

Nick Maclaren
University of Cambridge Computing Service,
New Museums Site, Pembroke Street,
Cambridge CB2 3QH, England.
Email: nmm1@cam.ac.uk
Tel.: +44 1223 334761
Fax: +44 1223 334679

IEEE 754R Support and Threading (and Decimal)

1.0 The Questions

This is an attempt to repeat my questions that I asked about Decimal Arithmetic, because it was so clearly misunderstood in Berlin, and has important consequences, especially for threading. **Exactly** the same questions apply when attempting to support IEEE 754 flags for any other reason.

I am not arguing against the decimal class proposal; I am agnostic about whether floating-point should use any base between 2 and 256; I am asking some questions about C++'s intent, and whether it implies support for IEEE 754 flags.

Specifically, I am asking the following questions, which could affect the **core** language in quite major ways.

Which (possibly all) of the following are intended:

Q 1.1 for the Decimal module to enable programs that do financial calculations to get the "right" (i.e. legally-required) answers?

Q 1.2 to enable full support for IEEE 754R (i.e. including the flags), whether in binary or decimal, assuming that it is standardised?

Q 2.1 to integrate the Decimal proposal with the current numeric classes, with implicit conversions to other classes?

Q 2.2 eventually to allow (or even require) the generic floating-point type to be a decimal class?

If the answer to any of these is "yes", **then** we had better start worrying about the long-term issues, **now**, especially in the context of the threading model.

To the readers of cpp-threads: **please** read at least section 1.3.

1.1 The Problem

The requirement on financial calculations is that they follow a set of conventions to the letter, using decimal **fixed-point** numbers with a tightly specified precision. They often include arcane rounding rules, and need to raise an exception (in some sense) whenever those rules are broken. Those conventions are usually written by lawyers, and may or may not make any mathematical sense.

IEEE 754 and IEEE 754R have similar properties, but are designed for use by specialist numerical computationalists; the rules are equally arcane and some of them are as mathematically dubious. That is not the point – IEEE 754 rules are what they are, and their believers have won the political battle (for now). See later.

In both cases, the 'solution' provided by IEEE 754 is to use its flags; the consequences of that are not commonly realised – see **1.2**. In particular, correct financial and robust scientific codes need two incompatible settings of the flag mask, and there is no mode that even makes sense for both.

In order to support either, there are two consequences:

- The implementation must support semi-global, updatable contexts (without prejudicing exactly how that is done) to support the exception flags (and possibly rounding and exception modes). These are critical in order to emulate decimal fixed-point reliably. These are very nasty indeed to handle in threaded code, and there has been heated debate in IEEE 754R on this very topic.
- The *as if* rule is essentially cancelled for floating-point types — in IEEE 754, `A+0.0` is not the same as `A`, and the same applies to almost every other mathematical equivalence. This essentially forbids any optimisation of floating-point expressions, and makes the current specification of when constructors are called **very** problematic, both of which will make many people very unhappy.

It is important to realise that neither of these problems is an issue with the current proposal, but immediately become one if the conditions I describe in the previous section are met. Obviously, if they are likely to be met, at least the threading model needs to consider the problems.

1.2 Values and Flags

As Professor Kahan points out in "How Java's Floating-Point Hurts Everyone Everywhere", IEEE 754's values are not reliable error indicators on their own, and **must** be combined with the flags. Also those flags must be checked frequently, which in turn means that the compiler must ensure that the flags are raised where the code implies that they are.

Let us start with the scientific use.

One example is that, for ordinary, scalar floating-point (i.e. not interval arithmetic and not complex number cut-planes), the sign of an infinity is meaningful only if the divide-by-zero exception was **not** raised during its creation. For ordinary, normalised numbers, consider:

```
double x;  // Assume it is provided with a value
x = 1.0/x;
```

If `x` is infinite and the divide-by-zero exception **has** been raised, its sign **isn't** meaningful and it should be regarded as a sort of NaN. If, however, the divide-by-zero exception has **not** been raised, its sign **is** meaningful and it stands for a number too large to represent.

The normal Fortran solution is to say that the IEEE 754 exceptions overflow, divide-by-zero and invalid are errors, and to terminate the program if they are not trapped. Inexact

and underflow are quietly ignored. That works very well for scientific calculations but, as described below, would not do so for financial ones.

Now let's consider finanancial calculations.

Some more details are given later, but the executive summary is that such calculations **must** trap the inexact exception on addition, subtraction and related operations, in order to turn it into fixed-point overflow. Consider adding 8.22 and 3.41 with a 3 digit mantissa (3 is used here for brevity; the possible values are 7, 16 and 34). In floating-point this is 1.16×10^1 and raises the inexact flag, but in fixed-point, it is 11.63 and thus overflows.

Unfortunately, that is not a primitive provided by IEEE 754R, and is flatly incompatible with scientific use. C++ could easily provide it, but I hope that the consequences are obvious.

Lastly, let's consider initialisation.

We have had several discussions on ordering issues to do with initialisation, and this one is very nasty. If we support IEEE 754 exception flags, or use separate modes for scientific and financial calculations, then they are global (at least to a thread), and therefore the location of an initialisation becomes visible to the program in a way that it wasn't before.

Now, this can be resolved by the implementation carefully preserving the modes and flags during out-of-order initialisation, restoring them afterwards and raising the initialiser-generated flags when the declaration is actually 'executed'. Should C++ require that? It is a significant performance penalty.

1.3 Threading Issues

Consider an Intel-like set of container and range classes, such as Arch Robison described in Redmond. Let's consider just a simple `vector` container class that takes a type and size, and a simple `range` method that takes a start and ending point and delivers a sub-container. For example, let us assume that `function` is an elemental function (in the Fortran sense):

```
vector<double,N> x, y, z;
double d;
x = function(y + 1.0/z);
```

If `function` looks at the flags, it is relying on every element of the containers `x, y` and `z` being handled in the same thread throughout the operations of division and addition, and in the function call. In this case that is not unreasonable, but consider:

```
vector<double,N> x, y, z;
double d;
x = y+x;
x += 1.0/z.range(0,N/2);
x = function(y + 1.0/z);
```

If we don't preserve the element to thread mapping for all three statements, the function will receive the wrong set of flags. That, I assert, is not acceptable.

If we do preserve it, that forces a fixed mapping from elements to threads, which is extremely inefficient, and will need a lot of specification. I am not sure that it is possible.

- Should we require the thread binding to be explicit, and make it the user's problem to handle this? That has both a major performance impact and forces even naïve users to get involved with some very hairy issues.
- Should flags be associated with the container or the statement? That would imply some synchronisation, and a fair amount of new mechanism and specification.
- Or should C++ turn away from IEEE 754 flags? That has serious consequences, at least for emulating decimal fixed-point, as described above. It would also seriously annoy the IEEE 754R camp.

  Of course, the default solution is to flannel, and to say nothing about this issue.

2.0. Myths of Decimal Floating-Point

The following comments are intended to correct a few myths — they should be read only by people who doubt my statements.

2.1. Decimal Floating-Point Is Suitable For Financial Calculations

As mentioned above, it isn't, because the requirement is for decimal fixed-point with a tightly specified precision, and decimal floating-point breaks some of the required rules. The lack of associativity of addition is perhaps the main trap (though there are others); incidentally, this also means that `(A+B)-A` is not necessarily `B`, even when no infinities, NaNs, overflow or underflow is involved, which is definitely unacceptable in accounting circles. As described above, 8.22 and 3.41 with a 3 digit mantissa overflows in fixed-point, but the floating-point hardware will deliver 1.16×10^1 and raise the inexact flag.

One theory is that every financial programmer will used Decimal-128, but I don't believe it. If C++ does not even **permit** reliable fixed-point overflow detection (what I called "Cobol" semantics), it is not something that I want to be associated with. But, as I say above, that means trapping the inexact exception and turning it into overflow, which is unacceptable for scientific code.

Both IBM's General Decimal Arithmetic specification and IEEE 754R deal with the differences between fixed-point and floating-point by defining mandatory modes and exceptions, which have the consequences mentioned above. Again, this is **not** an argument against including a decimal floating-point, but it **is** an argument for deciding on whether its intent implies the support for modes and flags.

The current Decimal proposal is **not** adequate for financial calculations as it stands, which is why I am trying to find out what its intent is and whether there is a next stage in its agenda.

3.2. Decimal Floating-Point is Numerically Worse than Binary

Back in the 1970s, there were a lot of claims that IBM 360/370 arithmetic was inaccurate because it used base 16, rather than base 2; those of us with experience of other bases

doubted it. A few of us did some investigations (I used AUGMENT to run in rounded mode on a System/370) and we proved that it was the truncation, not the base. Decimal might be a couple of bits less accurate compared to binary, but no more, and it doesn't make the error accumulation worse (in general).

There is one serious numeric trap that it does introduce, however. Many (perhaps most) numeric codes calculate the mid-point of a range using one of the obvious formulae; both `(A+B)/2.0` and `0.5*A+0.5*B` will always give a result in the range `[A,B]` for any faithful binary floating-point arithmetic or fixed-point in any base. This is **not** true for decimal floating-point (or any base other than 2). Note that this will not show up with simple (low digit count) test cases in IEEE 754R decimal floating-point, so is very likely to pass naïve testing.

So binary is fractionally better, numerically, but not enough to get excited about.

3.3. Serial Performance Is No Longer A Problem on Modern CPUs

For "mainframes" (i.e. almost all of what are now called "servers"), almost all commercial codes and most other codes run on "general purpose" systems, that is true. However, it is not true for either high-performance computing or embedded uses (probably including graphics); while the former may be forced into using decimal floating-point by the cost-effectiveness of commodity hardware, I doubt that the latter will accept it.

The reasons are chip area and power consumption, rather than the limit of performance. A large server chip can tolerate the area and power consumption of a complicated, optimised decimal floating-point, but it is a real problem for highly parallel designs and even more so for micro-miniaturised embedded designs. Such systems already use simplified forms of IEEE 754 for those reasons, and IEEE 754R decimal floating-point is more complicated yet again.

This is obviously **not** an argument against including a decimal floating-point class, or even for allowing it to be the "ordinary" floating-point, but it **is** an argument against requiring a C++ implementation to use it.

3.4. Decimal Makes Using Floating-Point Easier and More Reliable

It is claimed that naïve users have trouble with binary floating-point, which is true, but it is claimed that decimal floating-point makes numeric programming easier and more reliable, which is not. What it will mean is that naïve users will get further before having to be told to go back and rewrite their code **not** using assumptions that aren't true. As Mark Twain (and Josh Billings) said:

It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so.

Those of us who have to teach practical programming at a graduate level curse the amount of "unteaching" that we have to do, just to correct the fallacies that are taught at

school and in undergraduate courses. At those levels, many things have to be simplified, but far too much is over-simplified without considering the later consequences. Here are two examples (in addition to the very serious trap described above) that I can personally witness **will** be made worse by decimal floating-point:

- Programmers will not realise that addition and multiplication are not associative (e.g. `A+(B+C)` may not be `(A+B)+C` and `A+(B-A)` may not be `B`), even when infinities, NaNs, overflow or underflow are not involved. Note that addition **is** associative in fixed-point for any base and, as with the mid-point calculation, this will not show up with naïve testing in IEEE 754R decimal floating-point, but will in real use.

I could give many more examples along the same vein, but let's not bother.

- Programmers will be told that conversion is precise, and will misunderstand that to believe that the constants that they input are. No, π is not exactly 3.141593 or 3.14159265358979, and no physical measurements are precise; you might be amazed at how many graduate physicists learn that only as a result of querying why 0.1 is not printed out exactly on a computer.

I don't want to make too much of this, as I don't think that it's a big issue — it is just that the claim that decimal floating-point will make numeric programming easier and more reliable is wrong. It will make trivial numeric programming easier, at the cost of both making it harder to learn how to write reliable non-trivial numeric code and reducing the reliability of the average real (i.e. not classroom exercise) numeric program.