Understanding and Using Floating Point Numbers

by Jeff Bezanson

Numbers are surely the most prevalent kind of data in computer programs. They are so fundamental that people don't spend much time talking about them—surely everybody knows how to use numbers in their programs. Well, one of the wonderful things about programming is that nearly everywhere you look, you find more than meets the eye.

Most programmers have heard or observed one strange thing or another about floating point numbers. For example, we often discover that floating point numbers that look the same do not necessarily satisfy C's "==" test. New programmers are usually taught never to use == for floating point numbers for this reason. Occasionally we run into other exceptional cases, for instance mathematically sound formulae which, when implemented using floating point, produce seemingly random or disappointingly inaccurate results.

What is going on? In truth there is much that is counterintuitive about floating point numbers, and it helps to know a thing or two if you want to use them productively in your programs.

Floating point numbers provide a kind of illusion; they look like "real" numbers, with decimals and possibly very large or small magnitudes. In reality, a 4-byte floating point number, for example, can actually hold *fewer* distinct values than a 4-byte integer. The reason for this is, of course, that the internal representation of floating point numbers is not straightforward. Bits representing an integer are interpreted literally as a binary number, while bits in a floating point number have a more complicated interpretation. I will talk about this interpretation at length, but first I'd like to discuss some conceptual differences between integers and floats.

I. Accuracy vs. Precision

Accuracy and precision are two concepts in measurement that nicely capture the different properties of ints and floats (on any system, independent of the particular floating point representation used). "Accuracy" refers to how close a measurement is to the true value, whereas "precision" has to do with how much information you have about a quantity, how uniquely you have it pinned down.

Integer arithmetic enjoys the property of complete accuracy: if I have the integer "2", it is exactly 2, on the dot, and nobody can dispute that. Furthermore, if I add 1 to it, I know I will get exactly 3. Whatever operations I do with integers, provided there is no overflow, I will always get a number that matches the correct answer bit-for-bit. However, integers lack precision. Dividing both 5 and 4 by 2, for instance, will both yield 2. Integers are unable to keep track of the fractional part, so the information that I had a slightly bigger number than 4 (namely, 5) is lost in the process. Integers are too "chunky" to represent these finer gradations; using them is like building with bricks. If you want to make a cube, you know the bricks can represent it perfectly, but a sphere wouldn't come out quite as well (naturally, if the sphere is significantly larger than each brick, you might be able to get close enough).

Floating point numbers are the exact opposite of integers with respect to accuracy and precision. They have good precision, since they never deliberately discard information representing your numbers. If you had enough bits, you could reverse any FP calculation to get the original number, just like how, with enough bits, you could represent an arbitrarily large integer. But floating point numbers have poor accuracy. If ints are like bricks, then floats might be thought of as silly putty. You have enough control to mold them into complex curved shapes, but they leave something to be desired when it comes to forming a specified target shape. Imagine trying to make a perfect cube out of silly putty—you'll never get those corners as sharp as they should be. In fact, in many cases there is literally *no hope* of a floating point answer's matching the correct answer bit for bit.

Now how can I make an outlandish claim like that? Floating point numbers are inherently different from integers in that not every fraction can be represented exactly in binary, whereas any integer can. Binary arithmetic is not to blame; the same restriction applies in any base system. Consider the number 1/3. No finite decimal digit representation (e.g. 0.333333) can ever be equal to 1/3; we can never have enough 3's. Thus it is more than likely that the computed result you need *cannot* be represented accurately by a finite floating point variable—you're going to be wrong by at least a little bit no matter what you do. We know that floats are still useful, though; we just have to prevent those little roundoff errors from stepping on our toes. To that end, and also for general edification, it's nice to know how floats actually work.

Next: Floating Point Representation

YES!
It's a
Charger
रुपये
5
RUPEES
2009