C/FloatingPoint

[FrontPage] [TitleIndex] [WordIndex]

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at http://www.cs.yale.edu/homes/aspnes/#classes.

Real numbers are represented in C by the **floating point** types `float`, `double`, and `long double`. Just as the integer types can't represent all integers because they fit in a bounded number of bytes, so also the floating-point types can't represent all real numbers. The difference is that the integer types can represent values within their range exactly, while floating-point types almost always give only an approximation to the correct value, albeit across a much larger range. The three floating point types differ in how much space they use (32, 64, or 80 bits on x86 CPUs; possibly different amounts on other machines), and thus how much precision they provide. Most math library routines expect and return `double`s (e.g., `sin` is declared as `double sin(double)`, but there are usually `float` versions as well (`float sinf(float)`).

1. Floating point basics

The core idea of floating-point representations (as opposed to **fixed point representations** as used by, say, `int`s), is that a number x is written as $m*b^e$ where m is a **mantissa** or fractional part, b is a **base**, and **e** is an exponent. On modern computers the base is almost always 2, and for most floating-point representations the mantissa will be scaled to be between 1 and b. This is done by adjusting the exponent, e.g.

$1 = 1*2^0$
$2 = 1*2^1$
$0.375 = 1.5*2^{-2}$

etc.

The mantissa is usually represented in base b, as a binary fraction. So (in a very low-precision format), 1 would be $1.000*2^0$, 2 would be $1.000*2^1$, and 0.375 would be $1.100*2^{-2}$, where the first 1 after the decimal point counts as 1/2, the second as 1/4, etc. Note that for a properly-scaled (or **normalized**) floating-point number in base 2 the digit before the decimal point is always 1. For this reason it is usually dropped (although this requires a special representation for 0).

Negative values are typically handled by adding a **sign bit** that is 0 for positive numbers and 1 for negative numbers.

2. Floating-point constants

Any number that has a decimal point in it will be interpreted by the compiler as a floating-point number. Note that you have to put at least one digit after the decimal point: `2.0`, `3.75`, `-12.6112`. You can specific a floating point number in scientific notation using `e` for the exponent: `6.022e23`.

3. Operators

Floating-point types in C support most of the same arithmetic and relational operators as integer types; `x > y`, `x / y`, `x + y` all make sense when `x` and `y` are `float`s. If you mix two different floating-point types together, the less-precise one will be extended to match the precision of the more-precise one; this also works if you mix integer and floating point types as in `2 / 3.0`. Unlike integer division, floating-point division does not discard the fractional part (although it may produce round-off error: `2.0/3.0` gives `0.6666666666666663`, which is not quite exact). Be careful about accidentally using integer division when you mean to use floating-point division: `2/3` is `0`. Casts can be used to force floating-point division (see below).

Some operators that work on integers will *not* work on floating-point types. These are `%` (use `modf` from the math library if you really need to get a floating-point remainder) and all of the bitwise operators `~`, `<<`, `>>`, `&`, `^`, and `|`.

4. Conversion to and from integer types

Mixed uses of floating-point and integer types will convert the integers to floating-point.

You can convert floating-point numbers to and from integer types explicitly using casts. A typical use might be:

Toggle line numbers

```
 1 /* return the average of a list */
 2 double
 3 average(int n, int a[])
 4 {
 5     int sum = 0;
 6     int i;
 7
 8     for(i = 0; i < n; i++) {
 9         sum += a[i];
10     }
11
12     return (double) sum / n;
13 }
```

If we didn't put in the `(double)` to convert `sum` to a `double`, we'd end up doing integer division, which would truncate the fractional part of our average.

In the other direction, we can write:

Toggle line numbers

```
1     i = (int) f;
```

to convert a `float f` to `int i`. This conversion loses information by throwing away the fractional part of `f`: if `f` was `3.2`, `i` will end up being just `3`.

5. The IEEE-754 floating-point standard

The IEEE-754 floating-point standard is a standard for representing and manipulating floating-point quantities that is followed by all modern computer systems. It defines several standard representations of floating-point numbers, all of which have the following basic pattern (the specific layout here is for 32-bit `float`s):

```
bit  31 30    23 22                    0
     S  EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
```

The bit numbers are counting from the least-significant bit. The first bit is the sign (0 for positive, 1 for negative). The following 8 bits are the exponent in **excess-127** binary notation; this means that the binary pattern 01111111 = 127 represents an exponent of 0, 1000000 = 128, represents 1, 01111110 = 126 represents -1, and so forth. The mantissa fits in the remaining 24 bits, with its leading 1 stripped off as described above.

Certain numbers have a special representation. Because 0 cannot be represented in the standard form (there is no 1 before the decimal point), it is given the special representation `0 00000000 00000000000000000000000`. (There is also a -0 = `1 00000000 00000000000000000000000`, which looks equal to +0 but prints differently.) Numbers with exponents of 11111111 = 255 = 2^{128} represent non-numeric quantities such as "not a number" (`NaN`), returned by operations like (`0.0/0.0`) and positive or negative infinity. A table of some typical floating-point numbers (generated by the program float.c) is given below:

```
         0 =                       0 = 0 00000000 00000000000000000000000
        -0 =                      -0 = 1 00000000 00000000000000000000000
     0.125 =                   0.125 = 0 01111100 00000000000000000000000
      0.25 =                    0.25 = 0 01111101 00000000000000000000000
       0.5 =                     0.5 = 0 01111110 00000000000000000000000
         1 =                       1 = 0 01111111 00000000000000000000000
         2 =                       2 = 0 10000000 00000000000000000000000
         4 =                       4 = 0 10000001 00000000000000000000000
         8 =                       8 = 0 10000010 00000000000000000000000
     0.375 =                   0.375 = 0 01111101 10000000000000000000000
      0.75 =                    0.75 = 0 01111110 10000000000000000000000
       1.5 =                     1.5 = 0 01111111 10000000000000000000000
         3 =                       3 = 0 10000000 10000000000000000000000
         6 =                       6 = 0 10000001 10000000000000000000000
       0.1 =      0.10000000149011612 = 0 01111011 10011001100110011001101
       0.2 =      0.20000000298023224 = 0 01111100 10011001100110011001101
       0.4 =      0.40000000596046448 = 0 01111101 10011001100110011001101
       0.8 =      0.80000001192092896 = 0 01111110 10011001100110011001101
     1e+12 =             999999995904 = 0 10100110 11010001101010010100101
     1e+24 =   1.0000000138484279e+24 = 0 11001110 10100111100001000011100
     1e+36 =   9.9999996169031625e+35 = 0 11110110 10000001001011111001110
       inf =                     inf = 0 11111111 00000000000000000000000
      -inf =                    -inf = 1 11111111 00000000000000000000000
       nan =                     nan = 0 11111111 10000000000000000000000
```

What this means in practice is that a 32-bit floating-point value (e.g. a `float`) can represent any number between `1.17549435e-38` and `3.40282347e+38`, where the `e` separates the (base 10) exponent. Operations that would create a smaller value will underflow to 0 (slowly—IEEE 754

allows "denormalized" floating point numbers with reduced precision for very small values) and operations that would create a larger value will produce `inf` or `-inf` instead.

For a 64-bit `double`, the size of both the exponent and mantissa are larger; this gives a range from `1.7976931348623157e+308` to `2.2250738585072014e-308`, with similar behavior on underflow and overflow.

Intel processors internally use an even larger 80-bit floating-point format for all operations. Unless you declare your variables as `long double`, this should not be visible to you from C except that some operations that might otherwise produce overflow errors will not do so, provided all the variables involved sit in registers (typically the case only for local variables and function parameters).

6. Error

In general, floating-point numbers are not exact: they are likely to contain **round-off error** because of the truncation of the mantissa to a fixed number of bits. This is particularly noticeable for large values (e.g. `1e+12` in the table above), but can also be seen in fractions with values that aren't powers of 2 in the denominator (e.g. `0.1`). Round-off error is often invisible with the default float output formats, since they produce fewer digits than are stored internally, but can accumulate over time, particularly if you subtract floating-point quantities with values that are close (this wipes out the mantissa without wiping out the error, making the error much larger relative to the number that remains).

The easiest way to avoid accumulating error is to use high-precision floating-point numbers (this means using `double` instead of `float`). On modern CPUs there is little or no time penalty for doing so, although storing `double`s instead of `float`s will take twice as much space in memory.

Note that a consequence of the internal structure of IEEE 754 floating-point numbers is that small integers and fractions with small numerators and power-of-2 denominators can be represented *exactly*—indeed, the IEEE 754 standard carefully defines floating-point operations so that arithmetic on such exact integers will give the same answers as integer arithmetic would (except, of course, for division that produces a remainder). This fact can sometimes be exploited to get higher precision on integer values than is available from the standard integer types; for example, a `double` can represent any integer between -2^{53} and 2^{53} exactly, which is a much wider range than the values from `2^-31^ to 2^31^-1 that fit in a 32-bit` int `or` long. `(A 64-bit` long long `does better.) So` double` should be considered for applications where large precise integers are needed (such as calculating the net worth in pennies of a billionaire.)

One consequence of round-off error is that it is very difficult to test floating-point numbers for equality, unless you are sure you have an exact value as described above. It is generally not the case, for example, that `(0.1+0.1+0.1) == 0.3` in C. This can produce odd results if you try writing something like `for(f = 0.0; f <= 0.3; f += 0.1)`: it will be hard to predict in advance whether the loop body will be executed with `f = 0.3` or not. (Even more hilarity ensues if you write `for(f = 0.0; f != 0.3; f += 0.1)`, which after not quite hitting `0.3` exactly keeps looping for much longer than I am willing to wait to see it stop, but which I suspect will eventually converge to some constant value of `f` large enough that adding `0.1` to it has no effect.) Most of the time when you are tempted to test floats for equality, you are better off testing if one lies within a small distance from the other, e.g. by testing `fabs(x-y) <= fabs(EPSILON * y)`, where `EPSILON` is usually some application-dependent

tolerance. This isn't quite the same as equality (for example, it isn't transitive), but it usually closer to what you want.

7. Reading and writing floating-point numbers

Any numeric constant in a C program that contains a decimal point is treated as a `double` by default. You can also use `e` or `E` to add a base-10 exponent (see the table for some examples of this.) If you want to insist that a constant value is a `float` for some reason, you can append `F` on the end, as in `1.0F`.

For I/O, floating-point values are most easily read and written using `scanf` (and its relatives `fscanf` and `sscanf`) and `printf`. For `printf`, there is an elaborate variety of floating-point format codes; the easiest way to find out what these do is experiment with them. For `scanf`, pretty much the only two codes you need are `"%lf"`, which reads a `double` value into a `double *`, and `"%f"`, which reads a `float` value into a `float *`. Both these formats are exactly the same in `printf`, since a `float` is promoted to a `double` before being passed as an argument to `printf` (or any other function that doesn't declare the type of its arguments). But you have to be careful with the arguments to `scanf` or you will get odd results as only 4 bytes of your 8-byte `double` are filled in, or—even worse—8 bytes of your 4-byte `float` are.

8. Non-finite numbers in C

The values `nan`, `inf`, and `-inf` can't be written in this form as floating-point constants in a C program, but `printf` will generate them and `scanf` seems to recognize them. With some machines and compilers you may be able to use the macros `INFINITY` and `NAN` from `<math.h>` to generate infinite quantities. The macros `isinf` and `isnan` can be used to detect such quantities if they occur.

9. The math library

(See also KernighanRitchie Appendix B4.)

Many mathematical functions on floating-point values are not linked into C programs by default, but can be obtained by linking in the math library. Examples would be the trigonometric functions `sin`, `cos`, and `tan` (plus more exotic ones), `sqrt` for taking square roots, `pow` for exponentiation, `log` and `exp` for base-e logs and exponents, and `fmod` for when you really want to write `x%y` but one or both variables is a `double`. The standard math library functions all take `double`s as arguments and return `double` values; most implementations also provide some extra functions with similar names (e.g., `sinf`) that use `float`s instead, for applications where space or speed is more important than accuracy.

There are two parts to using the math library. The first is to include the line

Toggle line numbers

```
1 #include <math.h>
2
```

somewhere at the top of your source file. This tells the preprocessor to paste in the declarations of the math library functions found in `/usr/include/math.h`.

The second step is to link to the math library when you compile. This is done by passing the flag `-lm` to `gcc` *after* your C program source file(s). A typical command might be:

```
gcc -o program program.c -lm
```

If you don't do this, you will get errors from the compiler about missing functions. The reason is that the math library is not linked in by default, since for many system programs it's not needed.

CategoryProgrammingNotes

2014-06-17 11:57