CME 212: Lecture 11

Floating Point Representation

Floating point data types are finite precision approximations for real numbers. C++ standard recognizes three fundamental floating point data types: * `float` - single precision floating point type * `double` - double precision floating point type * `long double` - extended precision floating point type

The C++ standard does not specify the representation or even the precision of these three types. The C++ standard only says that: > "The type double provides at least as much precision as float, and the > type long double provides at least as much precision as double. The set > of values of the type float is a subset of the set of values of the type > double; the set of values of the type double is a subset of the set of > values of the type long double.

Floating point type representation may vary from compiler to compiler. Many compilers follow IEEE-754 standard in their floating point types implementation. In most cases * `float` will be 32-bit long and have 7 decimal significant digits in its mantissa, * `double` will be 64-bits long with 15 decimal significant digits in its mantissa. * Most common representations of `long double` type are 80-bit and 128-bit long.

Floating point literals

By default, any literal number with a decimal point is interpreted as `double`. To make compiler interpret a literal as a `float` or `long double`, one needs to postpend it with `F` or `L`, respectively. Floating point number can be written in an exponential form, as well. For example, `1234.5F` can be written as `1.2345e3F`. Literals without decimal point and not written in an exponential form are interpreted as integers. Here are a few examples:

```
auto x = 3.14;   // double
auto y = 3.14F;  // float
auto z = 3.14L;  // long double
auto u = 314L;   // long int
auto v = 314.L;  // long double
auto w = 3.14e2L // long double
```

Quick Review of floating point representation

Floating point numbers are not uniformly distributed over the real line; they have a higher concentration near the origin and are more spread out as their magnitudes grow. A floating point representation looks like $(-1)^{\texttt{sgn}} \left(\sum_{n=0}^{p-1} \texttt{bit}_n \times 2^{-n} \right) \times 2^e$, where we have a leading sign bit, p represents

the *precision* and e is our exponent. When our sign bit takes on unit value, it indicates a negative number. For a single precision floating point the IEEE standard allocates 1 bit for the sign, 23 bits to the mantissa, and the exponent is alloted 8 bits. To calculate the exponent, we actually use a bias term of $2^{k-1} - 1$ (127 for single precision, 1023 for double precision), that is we take our exponent to be $E = exp - \texttt{Bias}$. Written slightly differently, we can represent a decimal number by $(-1)^s 2^{e-127} \times 1.f$ where we remark that there is a "hidden 1" before the mantissa is for *normalized* values; a value is *denormalized* when the exponent field is all zeros, in this case we take $E = 1 - \texttt{Bias}$ and we do not take an implied leading one in front of the mantissa (this allows us to represent the zero value with the all zeros bit pattern).

We can use some low level programming techniques to print out the bit representation of a floating point. This code is *not* portable, as it assumes a particular endianness in printing out the bit representation (that's why our loop counters start from high and go to low).

```
void printBits(float f) {
        printf("Input: %8.5f = ", f);
        char *p = reinterpret_cast<char*>(&f);
        for (int i = sizeof(float)-1; i >= 0; --i) {
                for (int j = 7; j >= 0; --j)
                        // Look at i'th byte of float, pull out bits one at a time.
                        std::cout << (p[i] & (1 << j) ? 1 : 0);
                std::cout << ' ';
        }
        std::cout << std::endl;
}
```

If we make a call to `printBits(1)`, we'll see the output

```
Input:  1.00000 = 00111111 10000000 00000000 00000000
```

where the leading sign bit is zero (since our input is positive valued), and the next eight bits denote our exponent term which in this case is given by $1 + 2 + 4 + 8 + 16 + 32 + 64$ (reading from right to left, and noticing that the most significant bit of the exponent term is off), and all fractional bits are set to zero. In order to evaluate the exponent term properly we have to take into account the implicit bias term: $-1^0 \times 2^(1 + 2 + 4 + 8 + 16 + 32 + 64 - 127) = 1$.

We can print out other integer values like 2 or -2, i.e. `printBits(2)` and `printBits(-2)` yield:

```
Input:  2.00000 = 01000000 00000000 00000000 00000000
Input: -2.00000 = 11000000 00000000 00000000 00000000
```

Notice that we incremented our exponent term by one, which makes sense since the value 2 is twice as large as unit value, and to interpret our exponent bit we see that after applying bias: $2^(128 - 127) = 2$.

What happens with a simple fractional value? E.g. `printBits(2.5)`?

```
Input:  2.50000 = 01000000 00100000 00000000 00000000
```

Notice that our exponent bits are the same as before and now have a single "on" bit in our mantissa, which corresponds to 2^-2: this is because we can represent $2.5 = 2 \times (1 + 2^{-2}) = 2 \times 1.25$ where we recall our implied/hidden 1 as part of the mantissa or fractional component.

Floating point arithmetics is not associative

Due to fixed precision and round-off errors, associativity of addition and multiplication is not preserved in floating point representations. Let's take a look at this simple example:

```
float a = (1e-8F +  1.0F) - 0.99999F;
float b =  1e-8F + (1.0F  - 0.99999F);
std::cout << "a = " << a << "\n";
std::cout << "b = " << b << "\n";
```

On my machine, the output looks like this:

```
a = 1.00136e-05
b = 1.00236e-05
```

How can this be? Realize that if we allocate 23 bits for the mantissa, then this really corresponds to being able to represent 7 signficant decimal digits in a value, since $log(2^{-23})_{10} \approx -7$. When computing `a`, since the mantissa of `float` has 7 significant digits by properties of single precision floating point, the sum of `1e-8F` and `1.0F` will be truncated to `1.0F`. We can see this if we make a call to `printBits(1e-8F + 1.0F)` which prints out:

```
Input:  1.00000 = 00111111 10000000 00000000 00000000
```

Here we have no fractional bits turned on! What happens when we take the difference `1.0F - 0.99999F`? We can print out the bit representation, this time printing our input with greater precision to show off the remainder:

```
Input:  0.000010014 = 00110111 00101000 00000000 00000000
```

Notice here what we end up with: $2^(64+32+8+4+2-127)*(1+2^(-2)+2^(-4)) = 1.001358e-5$, where in particular we have some residual that distinguishes us from $1 - 0.9999 = 1e - 5$.

What happens when we compute `b`? Let's take a look at the binary representation for `1e-8F`:

```
Evaluating 1e-8F
Input:  0.000000010 = 00110010 00101011 11001100 01110111
```

Realize that this floating point approximation actually evaluates to $2^(64+32+4-127) \times (1+2^{-2}+2^{-4}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-17}+2^{-18}+2^{-19}+2^{-21}+2^{-22}+2^{-23})$ which is very close to $1e-8$ but there is some remainder since $1e-8$ cannot be represented exactly in a base 2 number system. When computing `b`, the difference between `1.0F` and `0.99999F` will again be 10-5 evaluated with only 3 correct digits due to finite precision truncation. Adding the literal `1e-8F` will this time change the result as it falls within the mantisa of `b`.

The results for `a` and `b` have the same number of significant digits and they agree to within truncation error, but they provide different representation of the number 1.001 x 10-5. This example illustrates that associativity of addition and multiplication is not a representation invariant of the floating point number.

Comparing floating point numbers

Since during a computation different floating point representations can be used to describe the same real number, one needs to be careful when implementing representation of a real number comparison. The rule of thumb is never assume two floating point numbers are equal. This is somewhat tongue in cheek statement, but take a look at this example:

```
double i = 0.0;
while (i != 1.0)
{
  i += 0.1;
  std::cout << i << "\n";
}
```

With most compilers this will be an infinite loop. The decimal number 0.1 has a binary representation with infinite number of digits 0.00011001100110011... Because of the truncation error, summing this value ten times will not produce exactly one, and the loop condition above will always be true.

When implementing comparison representation, we have to account for the truncation error. For example, if checking if two real numners are equal, we should check if they agree to within some tolerance that represents the truncation error. Let us modify our look like this:

```
double i = 0.0;
while (std::abs(i - 1.0) > 1e-14)
{
  i += 0.1;
  std::cout << i << "\n";
}
```

Now, we are checking if 1 - 10-14 < i < 1 + 10-14. The value of 10-14 is too small to affect our computation, but big enough to account for the truncation error.

Unfortunately, there are no simple rules how to set comparison tolerances. The truncation error may accumulate or cancel out depending on the computation, so the comparison tolerance will be very much computation dependent.

Numerically stable softmax

One more quick example. In machine learning we use a softmax function when performing multi-class classification tasks. This is a function that takes as input a vector of real numbers and outputs a probability distribution by first pushing each component through an exponential term and then normalizing by their sum. However, a naive implementation is at risk of overflow in the exponentials, which renders the output useless. It can be easily shown that subtracting out the maximum value of the vector from each of the inputs to the softmax does not change the resulting output or probability distribution. What it buys us is that by subtracting out the max entry, the entries of the input vector are all nonpositive, which rules out numerical overflow. Further, there is at least one zero element (corresponding to the maximum(s)) which rule out a vanishing denominator. We include this trick just so you can see that different contexts require different types of care to handle or gaurd against numerical issues Stack Overflow.

Numerical limits

Standard template library provides class `numeric_limits` that contains methods and constants that provide information on compiler representation of numeral data types. In file limits.cpp we show example usage of the `numeric_limits` class. This code provides basic information about floating point types implementation. On my machine, using clang compiler the output looks like this:

```
            min     max     mantisa exp min exp max radix
float       1.17549e-38 3.40282e+38 6   -37 38  2
double      2.22507e-308    1.79769e+308    15  -307    308 2
long double 3.3621e-4932    1.18973e+4932   18  -4931   4932    2
```

More information about `numeric_limits` class can be found at cppreference.com.

Reading

- What Every Computer Scientist Should Know About Floating-Point Arithmetic
- Wikipedia article on C data types.