Notes: Fractional Numbers and C++

Riley Taylor

June 2024

Fractional numbers are represented in floating point data structures - where a fraction of the bits assigned to the number is dedicated to significant digits, and another fraction of the number is dedicated to an exponent. This inherently causes accuracy problems for a number of reasons.

One reason is that some fractions in base 10 become very difficult to represent in base 2. 0.1 for example is simple in base 10, but in base 2, it is actually impossible to precisely store in base 2 without infinite space. See: here for more and here.

You can also check out the wikipedia page on floating point arithmetic.

1 Techniques to Convert from Decimal to Binary:

You can figure this out for yourself pretty easily, but it involves writing the integer part of a number as a sum: $\sum x_n 2^n,\ n \in\ \mathbb{Z}_{\geq 0}$ and the fractional part of the number as $\sum x_n 2^n,\ n \in\ \mathbb{Z}_{<0}$.

Notice that you can factor out a 2 (or a $\frac{1}{2}$ for the fractional component) and then proceed to a new number at each step, but you have a single extra term that represents the one's place in that number. For the integer component, if the number is odd, clearly that one's place is a 1, otherwise it would be an even number. For the fractional component, it isn't about odd or even, but literally if the resultant factored number is is greater than or equal to 1.

1.1 Fractional Component: Decimal to Binary Shortcut

Here's an example (note that N might not exist if we wish for an exact representation of the number - if the number cannot be written as the sum of powers of 2, it will not terminate and only become a successively closer approximation). Note also that because we can always choose to split a number into its fractional and integer components, we can choose this fractional approach to also assign $x_0 = 0$, because the actual value for x_0 will be picked up from the integer component conversion.

$$0.128_{10} = x_0 2^0 + x_{-1}2^{-1} + x_{-2}2^{-2} + ... + x_{-N}2^{-N}$$

$\therefore\ x_0 = 0$ because 0.128_{10} is less than 1, so x_{-1} cannot be 1. (of course, by this approach, x_0 will always be 0)

$$\begin{aligned} 0.128_{10} &= \frac{1}{2} * (x_{-1}2^0 + x_{-2}2^{-1} + ... + x_{-N}2^{-N+1}) \\ 0.128_{10} &= \frac{1}{2} * 0.256_{10} \\ 0.256_{10} &= x_{-1}2^0 + x_{-2}2^{-1} + ... + x_{-N}2^{-N+1} \end{aligned}$$

$\therefore\ x_{-1} = 0$ because 0.256_{10} is less than 1, so x_{-1} cannot be 1.

$$\begin{aligned} 0.256_{10} &= \frac{1}{2} * (x_{-2}2^0 + ... + x_{-N}2^{-N+2}) \\ 0.256_{10} &= \frac{1}{2} * 0.512_{10} \\ 0.512_{10} &= x_{-2}2^0 + x_{-3}2^{-1} + ... + x_{-N}2^{-N+2} \end{aligned}$$

$\therefore\ x_{-2} = 0$ because 0.512_{10} is less than 1, so x_{-2} cannot be 1.

$$0.512_{10} = \frac{1}{2} * (x_{-3}2^0 + ... + x_{-N}2^{-N+3})$$
$$0.512_{10} = \frac{1}{2} * 1024_{10}$$
$$1.024_{10} = x_{-3}2^0 + x_{-4}2^{-1} + ... + x_{-N}2^{-N+3}$$

$\therefore$ $x_{-3} = 1$ because 1.024_{10} is greater than 1, so x_{-3} cannot be 0 (recall the geometric series of 2^{-n} for $n = 1, 2, ..$ converges to 1 but never equals or surpasses 1).

$$1.024_{10} = 1 + \frac{1}{2} * (x_{-4}2^0 + ... + x_{-N}2^{-N+4})$$
$$1.024_{10} = 1 + \frac{1}{2} * 0.048_{10}$$

We only care about the 0.048_{10} for the rest of process, as we need to identify x_{-4} next.

$$0.048_{10} = x_{-4}2^0 + ... + x_{-N}2^{-N+4}$$

You can see how this approach works. The pattern continues. You grab the fractional component, factor out $\frac{1}{2}$ and record the 1's digit of the result.

A simpler way to see this:

$$\begin{aligned}
0.128_{10} &= 1/2 * 0.256_{10} && \rightarrow x_0 = 0 \text{ from } 0.128 < 1 \\
0.256_{10} &= 1/2 * 0.512_{10} && \rightarrow x_{-1} = 0 \text{ from } 0.256 < 1 \\
0.512_{10} &= 1/2 * 1.024_{10} && \rightarrow x_{-2} = 0 \text{ from } 0.512 < 1 \\
1.024_{10} &= 1 + 1/2 * 0.048_{10} && \rightarrow x_{-3} = 1 \text{ from } 1.024 \geq 1 \\
0.048_{10} &= 1/2 * 0.096_{10} && \rightarrow x_{-4} = 0 \text{ from } 0.048 < 1 \\
0.096_{10} &= \ldots && \rightarrow x_{-5} = 0 \\
0.192_{10} &= \ldots && \rightarrow x_{-6} = 0 \\
0.384_{10} &= \ldots && \rightarrow x_{-7} = 0 \\
0.768_{10} &= \ldots && \rightarrow x_{-8} = 0 \\
1.536_{10} &= \ldots && \rightarrow x_{-9} = 1 \\
1.072_{10} &= \ldots && \rightarrow x_{-10} = 1
\end{aligned}$$

Writing it out, we notice we have 0.0010000011_2 as our binary approximation (for 10 digits past the one's place) for our original number, 0.128_{10}. Obviously it's not exact, we ended the approximation early. But if you expand our approximation into base 10, you get about 0.12793_{10}, which is certainly already doing a decent job approximating 0.128_{10}.

1.2 Integer Component: Decimal to Binary Conversion Shortcut

The approach for converting the integer portion of the decimal number to binary is even easier and functions along similar logic. Instead, however, you record whether the number for the current step is even (meaning 0) or odd (meaning 1), and you work from x_N down towards x_0. Here's how it works:

$$93_{10} = x_7 2^7 + x_6 2^6 + ... + x_1 2^1 + x_0 2^0$$

Note that 93_{10} is odd, so x_0 must be 1.

$$93_{10} = x_7 2^7 + x_6 2^6 + ... + x_1 2^1 + 1$$
$$93_{10} = 2 * (x_7 2^6 + x_6 2^5 + ... + x_1 2^0) + 1$$
$$93_{10} = 2 * 46_{10} + 1$$
$$46_{10} = x_7 2^6 + x_6 2^5 + ... + x_1 2^0$$

Note that 46_{10} is even, so x_1 must be 0.

$$
\begin{aligned}
46_{10} &= x_7 2^6 + x_6 2^5 + ... + 0 \\
46_{10} &= 2 * (x_7 2^5 + x_6 2^4 + ... + x_2 2^0) \\
46_{10} &= 2 * (23_{10}) \\
23_{10} &= x_7 2^5 + x_6 2^4 + ... + x_2 2^0
\end{aligned}
$$

And we find out that $x_2 = 1$, because 23_{10} is odd.

We can write out this pattern a bit more simply:

$$
\begin{aligned}
93_{10} &= 2 * (46_{10}) + 1 &&\rightarrow\ x_0 = 1 \\
46_{10} &= 2 * (23_{10}) + 0 &&\rightarrow\ x_1 = 0 \\
23_{10} &= 2 * (11_{10}) + 1 &&\rightarrow\ x_2 = 1 \\
11_{10} &= 2 * (5_{10}) + 1 &&\rightarrow\ x_3 = 1 \\
5_{10} &= 2 * (2_{10}) + 1 &&\rightarrow\ x_4 = 1 \\
2_{10} &= 2 * (1_{10}) + 0 &&\rightarrow\ x_5 = 0 \\
1_{10} &= 2 * (0_{10}) + 1 &&\rightarrow\ x_6 = 1 \\
0_{10} &= 0 &&\rightarrow\ x_{n \geq 7} = 0
\end{aligned}
$$

But we write this out in as $x_6 x_5 x_4 x_3 x_2 x_1 x_0$ so the order is a little backwards, you have to read it from bottom to top. But we see that 93_{10} in base 2 is 0101 1101. Checking, we can verify: $2^6 + 2^4 + 2^3 + 2^2 + 2^0 = 64 + 16 + 8 + 4 + 1 = 93$.

Pretty useful.

2 Precision in Floating Points

Precision describes the number of digits that are correct. You can review a numerical analysis textbook to see the details. But essentially, **float** only guarantees a precision of about 7 digits. **double** guarantees a precision of around 15 digits.

But you can see how adding a small number to a large number can create problems - 123456789012345+ 0.123456 will lose the 0.123456 portion completely for a **double**.

Note that there is a lot of depth here in numerical analysis. Precision can be lost and compounded. Be explicit with your operations if precision matters - even equality checks can be wrong due to precision errors, and it's better to check if two numbers are equal by verifying that the distance between the numbers is sufficiently small.

3 Declaring Floating Point Numbers

Please see the accompanying **main.cpp**. You can specify **float** by appending you literal with an 'f', and likewise for **long double** with an 'L'. The default for a literal which has a decimal is a **double**.

4 Infinity and NAN

C++ returns Infinity and NAN in some cases. Infinity (positive or negative) for dividing a floating point number by 0, and NAN for dividing 0.0 by 0.0. Other cases too, but these are ways it handles specific undefined or infinite results.