int main () {
    float x = 0.0;
    while (x <= 1.0) {
        printf("x = %f\n", x);
        x = x + 0.1;
    }
    printf("the final value of x = %f\n", x);

double y = 0.2;
    while (y < 2.0) {
        printf("y = %1f\n", y);
        y = y + 0.2;
    }
    printf("the final value of y = %1f\n", y);
}
Announcements

● Assignment 9
  ○ Deadline extended to 23:59:59 Monday, Apr. 1
  ○ Do not submit it last minute!

● Make-up office hours
  ○ 9:30am on Monday, Apr. 1
Binary Encodings

CMPT 125
Mar. 29
Lecture 32

Today:

- Integer Encodings
- Floating Point Representation
- Endian-ness
Positional Value (Review)

The value of binary digits are positional

- just like decimal, except 2-fold instead of 10-fold

Decimal:

\[
739 = 700 + 30 + 9
\]

\[
\begin{array}{c}
10^0 \\
10^1 \\
10^2 \\
\end{array}
\]

Binary:

\[
1001 \ 0111_2
\]

\[
\begin{array}{c}
2^7 \\
2^6 \\
2^5 \\
2^4 \\
\end{array}
\]

\[
= 2^7 + 2^4 + 2^2 + 2^1 + 2^0
\]
Divisibility By 2

- Construct an FSM that accepts all binary strings divisible by 2
- Intuition: everything that ends with 0 should be accepted
- Adding 0
  - Multiply by 2: result is divisible by 2
  - $2n$ is divisible by 2 for any $n$
- Adding 1
  - multiply by 2, and add 1: not divisible by 2
  - $2n+1$ always has remainder 1 when divided by 2
Fixed Width Encodings

Simple data types are usually fixed in width

- usually multiples of 8 bits
- E.g., char (8 bits), int (often 32 bits), long (often 64 bits)

Puts a limit on the range of possible numbers

- for $k$ bits, gives a max of $2^k$ possibilities
- E.g., int: $[-2^{31}, 2^{31}-1]$

Step outside the range and you lose precision

- E.g., $x = 2147483647$; $x++$; results in an overflow
- You can also lose precision due to round-off errors

From Stack Overflow:
Q. What’s the maximum value for an int32? I can never remember that number. I need a memory rule.
A. It's $2,147,483,647$. Easiest way to memorize it is via a tattoo. (4923 upvotes)
Quick Estimates

- $2^{10} = 1024$, approximately $1000 \ (10^{3})$
- $2^{20}$ is about $1000 \times 1000$, or $1$ million ($10^{6}$)
- $2^{30}$ is about $1$ billion ($10^{9}$)
- $2^{31}$ is about $2$ billion ($2 \times 10^{9}$)
- The last bit is used for the sign
  - So largest positive number is about $2$ billion

- Another couple of facts:
  - 32-bit operating systems can access 4GB of RAM
  - 64-bit operating systems can access $4 \times 2^{32}$ GB of RAM
Non-Integer Arithmetic

Two common decimal numbers:

\[
\frac{1}{3} = 0.3333333\ldots \quad \frac{2}{3} = 0.6666666\ldots
\]

It’s easy if you have an infinite amount of paper!

But what if you have a fixed width of digits?

- you have to *truncate* and *round*.

These are the *significant digits* of the number

- also known as the *significand*
Scientific Notation (Review)

A convention to express numbers by their significand and their magnitude (exponent)

- E.g., $6.022 \times 10^{23}$ atoms/mol = $N_A$ (Avogadro’s Const.)
- E.g., $2.99792458 \times 10^8$ m/s = $c$ (speed of light)
- E.g., $1.073741824 \times 10^9$ bytes = 1 gigabyte

Common usage is to place one significant digit before the radix (decimal point)

- E.g., $\frac{1}{3} = 3.333333 \times 10^{-1}$

The same conventions are used for binary.
Floating Point Encoding

A float is composed of 32 bits:

- 1 bit for the sign — 0 → positive, 1 → negative
- 23 bits for the significand — $1.b_{22}b_{21}...b_{1}b_{0}$
  - approximately 7 decimal digits of precision
- 8 bits for the exponent — ranges from [-126,127]

Range of representations:

- $\pm 1.b_{22}b_{21}...b_{1}b_{0} \times 2^{\text{exp}}$, where:
  - the largest number $\approx 2^{128} \approx 3.40 \times 10^{38}$
  - the smallest number $\approx 2^{-126} \approx 1.17 \times 10^{-38}$
- There is a “special” representation for 0
- . . . and a handful of other special cases

Sign-magnitude representation of negatives
Example: -0.625

Decimal fraction: -\(\frac{5}{8}\)
- sign bit? = 1 (negative)
- significand and exponent? \(\frac{5}{8} = 0.001\), so \(\frac{5}{8} = 0.101\)

An exact number! No rounding!
- \(-0.625 = -1.01000000000000000000000 \times 2^{-1}\)

Format:

```
1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

+ bias ( = 127)
Example: 0.1

Decimal fraction: 1/10

- sign bit? = 0
- significand and exponent? — try long division

\[
\begin{array}{c}
\text{1010} \\
\text{1.0000000000000000000000000000000000...} \\
\text{1010} \\
\text{11001100110011001100110011001100110011001100...} \\
\end{array}
\]

- repeating decimal — truncate and round
- exponent = -4
Special Exponents

- Exponent has 8 bits
  - but ranges from [-126, 127]?  
  - 8 bits has $2^8 = 256$ possible values  
  - -126 to 127 has $127 - (-126) + 1 = 254$ values

- How to represent 0?
  - 1.(anything) times $2^{(anything)}$ is never going to be zero

- Two exponents reserved for special cases
  - All zero exponent bits $\rightarrow$ sign*0.(significand)*$2^{-126}$
  - All one exponent bits $\rightarrow$ infinity, NaN
  - NaN: “not a number”, for e.g. when dividing by 0
A Note about Endian-ness

A multiple-byte quantity, like `int` or `float`, is stored across a contiguous sequence of addresses in memory or in a file.

- two possible memory / file layouts

```
int var = -100000;
```

---

**“Little-Endian”**

```
var:
 FF
 FE
 79
 60
```

- increasing address

**“Big-Endian”**

```
var:
 FF
 FE
 79
 60
```

- increasing address