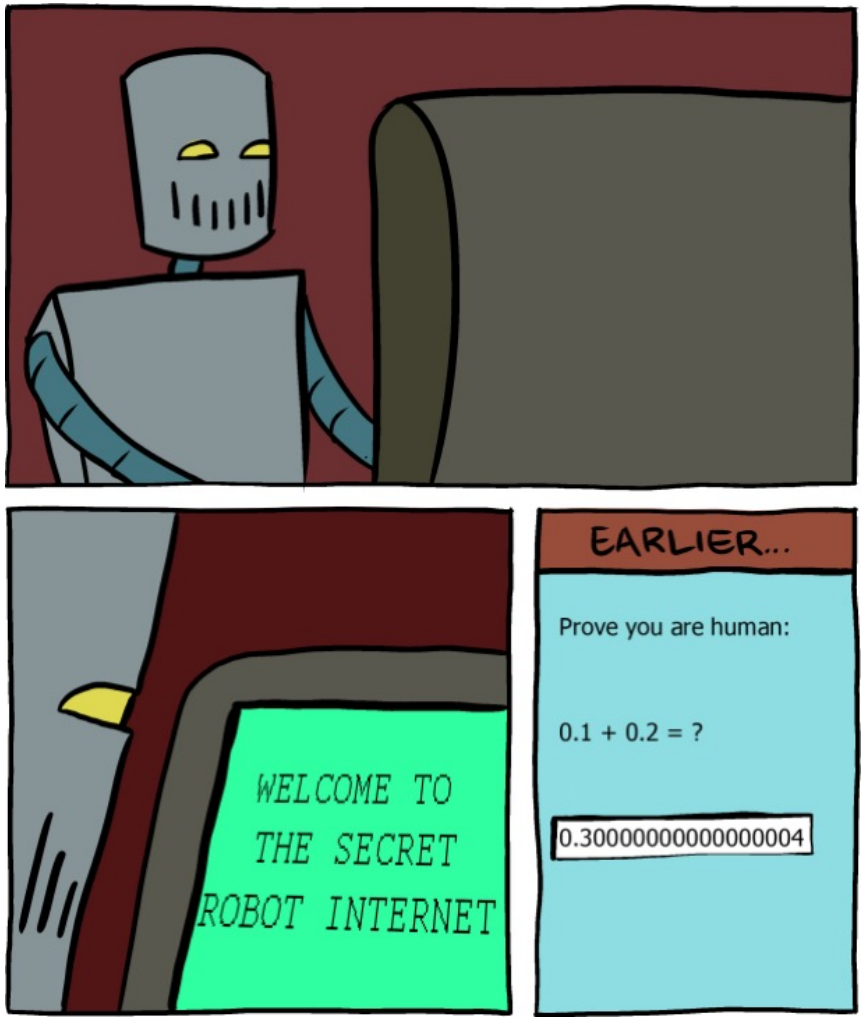
CS107, Lecture 16
assign6 / Floating Point

Reading: B&O 2.4



Plan For Today

- assign6
- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- Floating Point: Special/denormalized values
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

Plan For Today

- **assign6**
- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- Floating Point: Special/denormalized values
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

Assign6: Heap Allocator

- Implement your own implicit and explicit free list allocators
- Many things specified explicitly in the spec, but some design decisions (search policy for free blocks, etc.) up to you!
- Bump allocator provided as implementation example

Assign6: Tips

- Understand heap allocator design *before* you start coding (e.g. how does it find a free block? What does a header look like? Etc.)
- Start early and develop **incrementally** – you do not need to implement the whole allocator before testing!
- Test thoroughly
- Become familiar with the test harness code
- Watch the getting started videos!
- Take advantage of Ed and helper hours

What questions do you have about working on assign6?

Plan For Today

- assign6
- **Representing real numbers and (thought experiment) fixed point**
- Floating Point: Normalized values
- Floating Point: Special/denormalized values
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

How can a computer represent real numbers in addition to integer numbers?

Learning Goals

Understand the design and compromises of the floating point representation, including:

- Fixed point vs. floating point
- How a floating point number is represented in binary
- Issues with floating point imprecision
- Other potential pitfalls using floating point numbers in programs

Real Numbers

- We previously discussed representing integer numbers using two's complement.
- However, this system does not represent real numbers such as 3/5 or 0.25.
- How can we design a representation for real numbers?

Real Numbers

Problem: There are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1

Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001,…

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers*.

Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: $1/6_{10} = 0.16666666\ldots..{}_{10}$

Base 2: $1/10_{10} = 0.0001100110011001100110011\ldots{}_{2}$

Therefore, by representing in base 2, *we will not be able to represent all numbers*, even those we can exactly represent in base 10.

Thought Experiment: Fixed Point

Idea: Like in base 10, let's add binary decimal places to our existing number representation.

5	9	3	4	.	2	1	6
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}

1	0	1	1	.	0	1	1
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

Thought Experiment: Fixed Point

Idea: Like in base 10, let's add binary decimal places to our existing number representation.

1 0 1 1 . 0 1 1

8s 4s 2s 1s 1/2s 1/4s 1/8s

Pros: arithmetic is easy! And we know exactly how much precision we have.

Thought Experiment: Fixed Point

Problem: we must fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

. 0 1 1 0 0 1 1

1/2s 1/4s 1/8s …

1 0 1 1 0 . 1 1

16s 8s 4s 2s 1s 1/2s 1/4s

The Problem with Fixed Point

Problem: We must fix where the decimal point is in our representation. This fixes our **precision**.

$$6.022e23 = \underbrace{11\ldots0}_{79\text{ bits}}.0$$

(base 10) (base 2)

$$6.626e\text{-}34 = 0.\underbrace{0\ldots01}_{111\text{ bits}}$$

To store both these numbers in the same fixed-point representation, the bit width of the type would need to be at least 190 bits wide!

Let's Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible "floating" decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable overflow behavior

Plan For Today

- assign6
- Representing real numbers and (thought experiment) fixed point
- **Floating Point: Normalized values**
- Floating Point: Special/denormalized values
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

Let's Get Real

What would be nice to have in a real number representation?

- ❑ Represent widest range of numbers possible
- ❑ Flexible "floating" decimal point
- ❑ Still be able to compare quickly
- ❑ Represent scientific notation numbers, e.g. 1.2×10^6
- ❑ Have more predictable overflow behavior

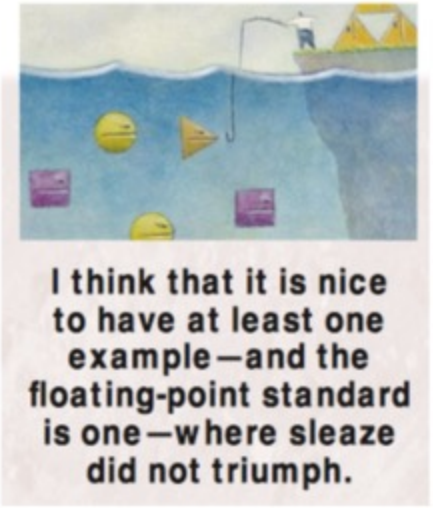
IEEE floating point

IEEE Standard 754

- Established in 1985 as a uniform standard for floating point arithmetic
- Supported by all major systems today
  Hardware: specialized co-processor vs. integrated into main chip

Driven by numerical concerns

- Behavior defined in mathematical terms
- Clear standards for rounding, overflow, underflow
- Support for transcendental functions (roots, trig, exponentials, logs)
- Hard to make fast in hardware
  Numerical analysts predominated over hardware designers in defining standard



— Will Kahan
(chief architect of standard)

IEEE Floating Point

Let's aim to represent numbers of the following scientific-notation-like format:

$$V = (-1)^{s} \times M \times 2^{E}$$

Sign bit, s:
negative (s == 1)
positive (s == 0)

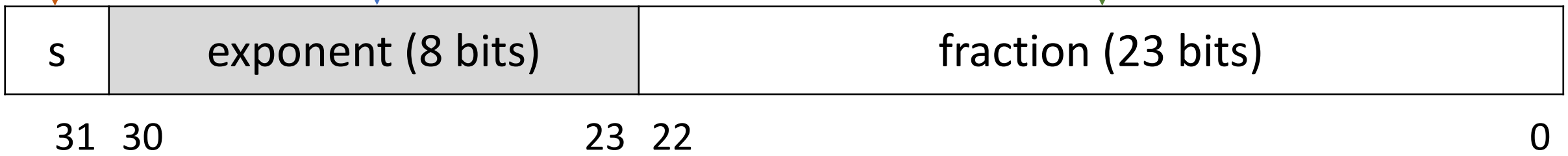
Mantissa, M :
Significant digits, also called significand

Exponent, E:
Scales value by (possibly negative) power of 2

IEEE Floating Point

Let's aim to represent numbers of the following scientific-notation-like format:

$$V = (-1)^{s} \times M \times 2^{E}$$



⚠ Quirky! Exponent and fraction are not encoded as M and E in 2's complement!

Exponent

exponent (Binary)	E (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

special

normalized

denormalized

Exponent: Normalized values

exponent (Binary)	E (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

- Based on this table, how do we compute an exponent from a binary value?
- Why would this be a good idea? (hint: what if we wanted to compare two floats with >, <, =?)

Exponent: Normalized values

exponent (Binary)	E (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

- Not 2's complement
- E = exponent – **bias**, where `float` **bias** = $2^{8-1} - 1 = 127$
- Exponents are sequentially represented starting from 000…1 (most negative) to 111…10 (most positive).
- Bit-level comparison is fast!

Fraction

$$x * 2^y$$

- We could just encode whatever x is in the fraction field. But there's a trick we can use to make the most out of the bits we have.

An Interesting Observation

In Base 10:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

$0.624 \times 10^5 = 6.24 \times 10^4$

We tend to adjust the exponent until we get down to one place to the left of the decimal point.

In Base 2:

$10.1 \times 2^5 = 1.01 \times 2^6$

$1011.1 \times 2^5 = 1.0111 \times 2^8$

$0.110 \times 2^5 = 1.10 \times 2^4$

Observation: in base 2, this means there is *always* a 1 to the left of the decimal point!

Fraction

$$x * 2^{y}$$

- We can adjust this value to fit the format described previously. Then, x will always be in the format **1.XXXXXXXXX…**
- Therefore, in the fraction portion, we can encode just what is *to the right* of the decimal point! This means we get one more digit for precision.

Value encoded = 1._[FRACTION BINARY DIGITS]_

Practice #1

$$V = (-1)^s \times M \times 2^E$$

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

1. Is this number:
   A. Greater than 0?
   B. Less than 0?

2. Is this number:
   A. Less than -1?
   B. Between -1 and 1?
   C. Greater than 1?

3. Bonus: What is the number?

Practice #1

$$V = (-1)^s \times M \times 2^E$$

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

1. Is this number:
   A. Greater than 0?
   B. Less than 0?

2. Is this number:
   A. Less than -1?
   B. Between -1 and 1?
   C. Greater than 1?

3. Bonus: What is the number?

$$(-1)^0 \times 1.0 \times 2^{-1} = 0.5$$

Let's Get Real

What would be nice to have in a real number representation?

- ✓ Represent widest range of numbers possible
- ✓ Flexible "floating" decimal point
- ✓ Still be able to compare quickly
- ❑ Represent scientific notation numbers, e.g. 1.2×10^6
- ❑ Have more predictable overflow behavior

Plan For Today

- assign6
- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- **Floating Point: Special/denormalized values**
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

All zeros: Zero + denormalized floats

Zero (+0, -0)

s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	all zeros

Denormalized floats:

s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	any nonzero

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has **no leading zero**: $M = 0.[\text{fraction bits}]$

$$V = (-1)^s \times M \times 2^E$$

Why would we want so much precision for tiny numbers?

All zeros: Zero + denormalized floats

Zero (+0, -0)

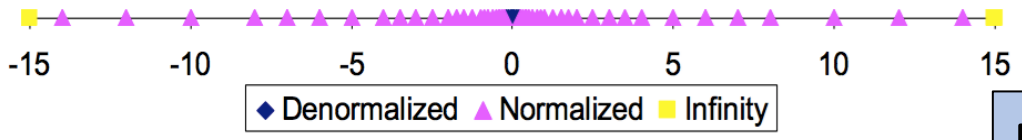
s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	all zeros

Denormalized floats:

s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	any nonzero

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has **no leading zero**: $M = 0.[\text{fraction bits}]$

$$V = (-1)^s \times M \times 2^E$$



Denormalized values enable gradual **underflow** (too-small-to-represent floats).

All ones: Infinity and NaN

Infinity (`+inf`, `-inf`)

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	**`all zeros`**

Why would we want to represent infinity?

Not a number (**NaN**):

Computation result that is an invalid mathematical real number.

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	**`any nonzero`**

What kind of mathematical computation would result in a **non-real** number? (hint: square root)

All ones: Infinity and NaN

Infinity (`+inf`, `-inf`)

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	**`all zeros`**

Floats have built-in handling of overflow: infinity + anything = infinity.

Not a number (**`NaN`**):

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	**`any nonzero`**

Computation result that is an invalid mathematical real number.

Examples: `sqrt(x)` (i.e., $\sqrt[2]{x}$), where x is negative, $\frac{\infty}{\infty}$, $\infty + (-\infty)$, etc.

Let's Get Real

What would be nice to have in a real number representation?

- ✓ Represent widest range of numbers possible
- ✓ Flexible "floating" decimal point
- ✓ Still be able to compare quickly
- ✓ Represent scientific notation numbers, e.g. 1.2×10^6
- ✓ Have more predictable overflow behavior

Number Ranges

- 32-bit integer (type **int**):
  › -2,147,483,648 to 2147483647
- 64-bit integer (type **long**):
  › –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- 32-bit floating point (type **float**):
  - ~1.2 x10^{-38} to ~3.4 x10^{38}
  - Not all numbers in the range can be represented (not even all integers in the range can be represented!)
  - Gaps can get quite large! (larger the exponent, larger the gap between successive fraction values)
- 64-bit floating point (type **double**):
  - ~2.2 x10^{-308} to ~1.8 x10^{308}

Skipping Numbers

- We said that it's not possible to represent *all* real numbers using a fixed-width representation. What does this look like?

Float Converter

- https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floats and Graphics

- https://www.shadertoy.com/view/4tVyDK

Plan For Today

- assign6
- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- Floating Point: Special/denormalized values
- **Floating Point Arithmetic**

```
cp -r /afs/ir/class/cs107/samples/lectures/lect16 .
```

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.

Nick's Official Guide To Making Money

Demo: Float Arithmetic

bank.c

Try it yourself:

```
./bank 100 1          # deposit
./bank 100 -1         # withdraw
./bank 100000000 -1 # make bank
./bank 16777216 1    # lose bank
```

Introducing "Minifloat"

For a more compact example representation, we will use an 8 bit "minifloat" with a 4 bit exponent, 3 bit fraction and bias of 7 (note: minifloat is just for example purposes, and is not a real datatype).

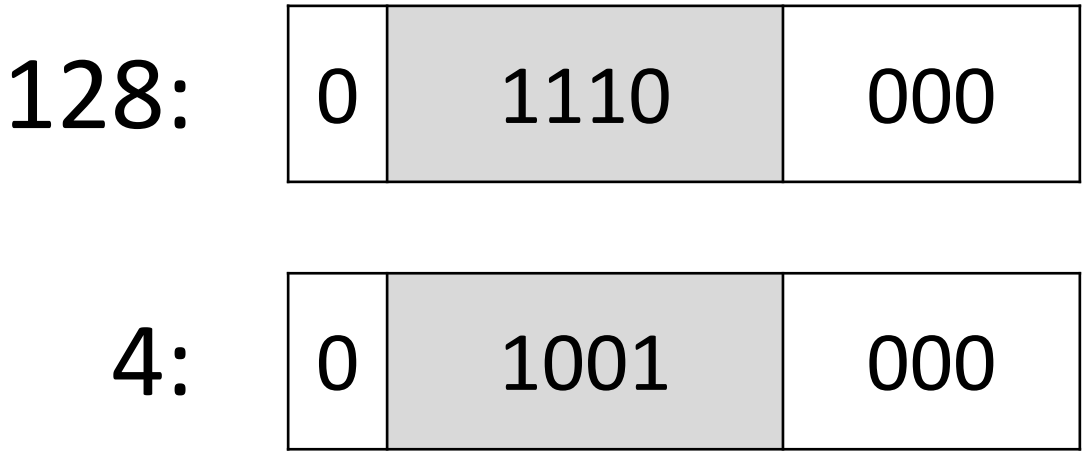
Floating Point Arithmetic

In minifloat, with a balance of $128, a deposit of $4 **would not be recorded** at Nick's Bank. Why not?

128:	0	1110	000
4:	0	1001	000

Let's step through the calculations to add these two numbers (note: this is just for understanding; real float calculations are more efficient).

Floating Point Arithmetic



To add real numbers, we must align their binary points:

```
    128.00
+     4.00
-----------
    132.00
```

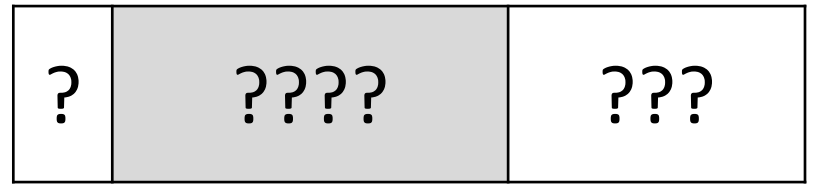
What does 132.00 look like as a minifloat?

Floating Point Arithmetic

Step 1: convert from base 10 to binary

What is 132 in binary? **1 0 0 0 0 1 0 0**

132:



Floating Point Arithmetic

Step 2: find how many places we need to shift **left** to put the number in 1.xxx format. This fills in the exponent component.

$0b10000100 = 0b1.0000100 \times 2^7$

7 + bias of 7 = 14 for minifloat exponent

132:

?	1110	???

Floating Point Arithmetic

Step 3: take as many digits to the right of the binary decimal point as we can for the fractional component, rounding if needed.

$0b10000100 = 0b1.0000100 \times 2^7$

132:

?	1110	000

Floating Point Arithmetic

Step 4: if the sign is positive, the sign bit is 0. Otherwise, it's 1.

+132

Sign bit is 0.

132:

0	1110	000

Floating Point Arithmetic

The binary minifloat representation for 132 thus equals the following:

0	1110	000

This is the **same** as the binary representation for 128 that we had before!

We didn't have enough bits to differentiate between 128 and 132.

Floating Point Arithmetic

Another way to corroborate this: the *next-largest minifloat* that can be represented after 128 is **144**. 132 isn't representable!

144:

0	1110	001

$= 1.125 \times 2^7$

Key Idea: the smallest float hop increase we can take is incrementing the fractional component by 1.

Floating Point Arithmetic

Is this just overflowing? It turns out it's more subtle.

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints 3.14
```

Floating point arithmetic is not associative. The order of operations matters!

- The first line loses precision when first adding 3.14 and 1e20, as we have seen.
- The second line first evaluates 1e20 – 1e20 = 0, and then adds 3.14

Demo: Float Equality

`float_equality.c`

Floating Point Arithmetic

Float arithmetic is an issue with most languages, not just C!

- http://geocar.sdf1.org/numbers.html

Let's Get Real

What would be nice to have in a real number representation?

- ✓ Represent widest range of numbers possible
- ✓ Flexible "floating" decimal point
- ✓ Still be able to compare quickly
- ✓ Represent scientific notation numbers, e.g. 1.2×10^6
- ✓ Have more predictable overflow behavior

Floats Summary

- IEEE Floating Point is a carefully-thought-out standard. It's complicated but engineered for their goals.
- Floats have an extremely wide range but cannot represent every number in that range.
- Some approximation and rounding may occur! This means you don't want to use floats e.g. for currency.
- Associativity does not hold for numbers far apart in the range
- Equality comparison operations are often unwise.

Recap

- **Recap:** Generics with Function Pointers
- Representing real numbers
- Fixed Point
- **Break:** Announcements
- Floating Point
- Floating Point Arithmetic

Next time: assembly language