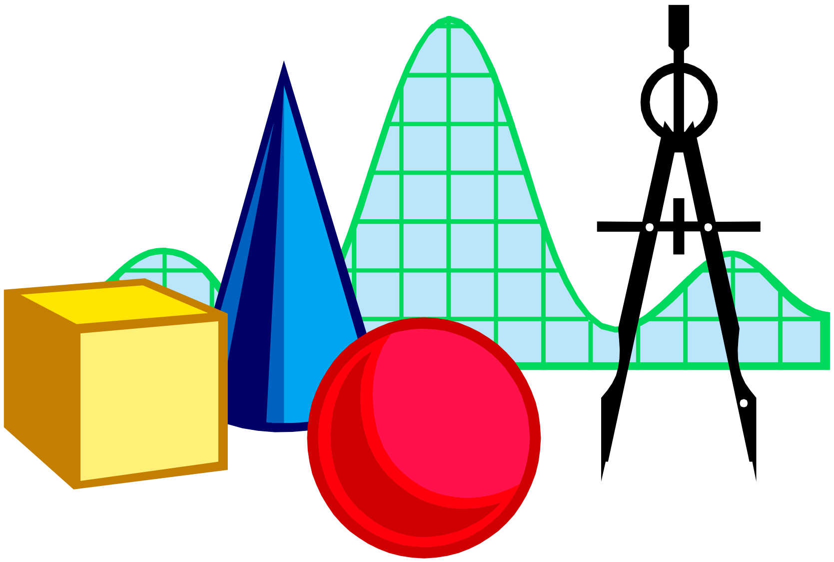
Floating-point arithmetic

- Floating-point computations are vital to many applications. However, it's pretty hard to implement a floating-point system.
- Today we'll look at the IEEE 754 floating-point arithmetic standard.
  - Floating-point numbers have their own binary representation.
  - Rounding numbers is essential, but leads to roundoff errors.
  - The standard includes some special values for special situations.
- We'll use the rest of the time for exam questions and answers.



Floating-point representation

- IEEE numbers are stored in a kind of scientific notation.

$$\pm \text{ fraction} \times 2^{\text{exponent}}$$

- We can represent floating-point numbers with three binary fields.

sign	exponent	fraction

- IEEE 754 defines two formats which differ only in the length of the fields.
  - Single precision numbers have one sign bit, an 8-bit exponent, and a 23-bit fraction, for a total of 32 bits.
  - Double precision numbers have a sign bit, an 11-bit exponent field and a 52-bit fraction field, for a total of 64 bits.
- This is a simplified overview of the IEEE format; the representation of the fraction and exponent fields is fairly complex.

Signed magnitude

- IEEE numbers use a signed magnitude format.
- This makes operations like multiplication fairly easy to implement.
  - To multiply two numbers, first multiply their magnitudes.
  - If the numbers have the same sign, the result is positive. Otherwise the result is negative.
- However, one of the drawbacks we mentioned about signed magnitude is that there are *two* zeroes—a positive one and a negative one!
- It turns out that 0-x and -x are *not* the same!

```
float x = 0.0;
printf( "%f\n", 0.0-x );
printf( "%f\n", -x    );
```

- IEEE hardware and software have to take this into account.

Finiteness

- Most modern machines store data in 32-bit chunks. This is only enough to represent about 4 billion (2^{32}) different values.
  - For signed integers, we can represent all the numbers between about –2 billion and +2 billion.
  - But there are an *infinite* number of reals, and we can only represent *some* of the ones between roughly -2^{128} to $+2^{128}$.
- This limitation causes enormous headaches in doing arithmetic.

Limits of the IEEE representation

- Some integers simply cannot be represented in IEEE format.

```
int x   = 33554431;
float y = 33554431;
printf( "%d\n", x );
printf( "%f\n", y );
```

- Some simple decimal numbers cannot be represented exactly in binary. Recall one of the questions from Homework 1, for example:

$$0.10_{10} = 0.0001100110011\ldots_2$$

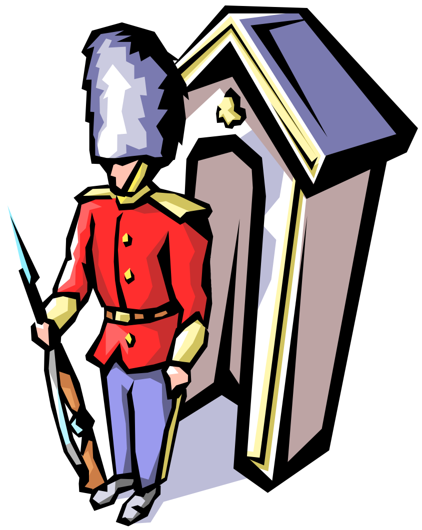
- In addition to overflow, now we have to worry about underflow, where the magnitude of a number is too *small* to represent. For example, what happens if we divide the smallest positive number, 2^{-126}, by two?

0.10

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!
- Professor Skeel wrote a short article about this.

[Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.]

Guarding against rounding errors



- With a limited number of representable numbers, it's very possible that some results will have to be rounded, and rounding errors will occur.
- Seemingly small roundoff errors can quickly accumulate, especially with multiplications and exponentiations.
- To help minimize rounding problems, IEEE 754 requires implementations to use guard digits—additional bits that increase the internal precision of operations.

Extreme errors

- Rounding errors in addition can still occur if one argument is significantly smaller than the other, since we can never have enough precision.
- An extreme example is something like the following.

$$(1.5 \times 10^{38}) + (1.0 \times 10^{0}) = 1.5 \times 10^{38}$$

  The number 1.0×10^{0} is much smaller than 1.5×10^{38}, and it basically gets rounded out of existence.
- This has some nasty implications. The order in which you do additions can affect the result, so (x + y) + z is not always the same as x + (y + z)!

```
float x = -1.5e38;
float y =  1.5e38;
printf( “%f\n”, (x + y) + 1.0 );
printf( “%f\n”, x + (y + 1.0) );
```

Converting between precisions

- Another loss-of-precision problem occurs when double-precision numbers are converted, or cast, to single precision.
- In some languages like C the conversions occur automatically, which can yield some unexpected results.

```
float x = 3.0 / 7.0;
if ( x == 3.0 / 7.0 )
   printf("Equal\n");
else
   printf("Not equal\n");
```

- Here 3.0/7.0 is a double-precision value that is automatically converted to single-precision format in the first line.
- In the second line, "x" is converted back to double precision, but it is no longer equal to its original value.

NaN

- Sometimes it's easier to continue a computation even if an error occurs.
- A special "not a number" value NaN can represent undefined results such as 0/0 or the square root of a negative number.

```
printf( "%f\n", 0.0/0.0 );
```

- NaNs are propagated, so any operation on a NaN will also yield a NaN.
- A NaN is never equal to anything—not even another NaN!

```
float x = 0.0 / 0.0;
if ( x == x )
  printf("Equal\n");
else
  printf("Not equal\n");
```

Infinity

- Infinity is defined as an alternative to overflow, which would otherwise usually lead to an exception or the wrong answer.
- Positive and negative infinities are included so the sign of the answer can be preserved, even if its magnitude can't.
- Here are some fun and interesting cases.

```
printf( "%f\n",  0.0/0.0 );     /*  NaN */
printf( "%f\n",  1.0/0.0 );     /*  Inf */
printf( "%f\n", -1.0/0.0 );     /* -Inf */
```

Summary

- The IEEE 754 floating-point standard has lots of interesting features.
  - Numbers can be represented in single precision or double precision.
  - Guard bits help to increase internal precision.
  - There are some special values like NaN and infinity.
- Having a finite number of bits is a big problem because we have to throw a lot of arithmetic principles out the window.
  - $+0$ is not the same as -0.
  - $0-x$ is not the same as $-x$.
  - $(x + y) + z$ is not the same as $x + (y + z)$.
  - x is not always the same as x.
- Implementing and programming floating-point correctly are *both* hard.