I wanted to clear up the question posed in class regarding integers (*int*) and floating point numbers (*float*). While it is true that ints and floats take up the same amount of space in memory, the way that they are stored in memory sets the two apart.

Observe:

Integers and floats are both 4 bytes in length. 4 bytes = 32 bits, or 32 "1's or 0's."

Integers are stored as pure binary representations of a number. For instance, if I wished to store the number 428 in a variable of type *int*, C would convert my decimal (base 10) number to binary (base 2). 428 in binary looks like this: 110101100. Now, we've got to store this in the 32 bit (4 byte) space in memory which was created when we initialized our *int*. Storing 428 gives us a block of memory which looks like this:

Unsigned integer - binary representation of 428

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	0

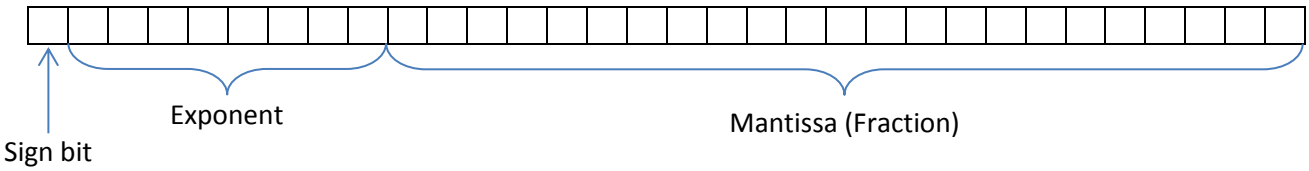
If you type this number in your handy binary to decimal converter, you'll get 428.

A note about signed vs. unsigned: I'm treating this block of memory as if it were unsigned. Were it a signed integer, the most significant bit (the one that is circled below) would represent whether the number was negative or positive. If that number was a 1, the other 31 digits would represent a negative number. (All the other digits would be flipped and we'd add a 1 as well, due to storing things in two's complement. If you're not familiar, I encourage you to read the Wikipedia article about two's complement.) However, 428 would be the same in both signed and unsigned, as the most significant bit is 0, meaning it is positive.

Signed integer - binary representation of 428

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0	0

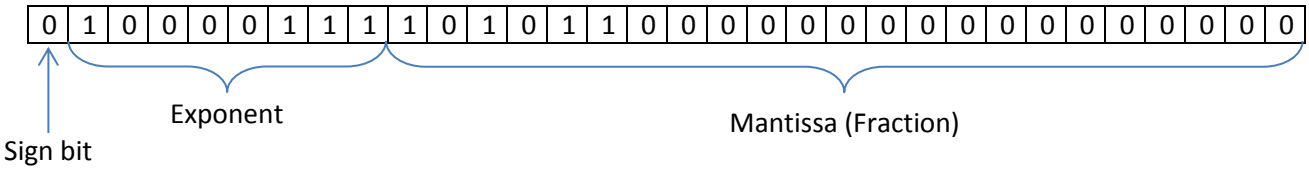
Now, what about *floats*? When we initialize a variable of type *float*, we get the same size memory block as we did when we created an *int* - 32 bits (4 bytes). However, the difference comes from the way we divide our memory up to store the number. IEEE standard 754 gives the definition for this. The memory block for our *float* is divided as follows:



To represent a number, we must combine all three elements of this "data structure" in memory to get our representation. The first component, the sign bit, is easy - a zero means a positive number, a one means a negative. The second component is the exponent to which we will raise the number 2. The third number is the fraction (technically "mantissa") by which we multiply 2 raised to this exponent. The formula for calculation looks like this:

$$Number = (-1)^{sign} * 2^{exponent} * mantissa$$

To represent 428, our number from the above example, our memory block would look like this:



To get 428 out of this, we first convert the exponent from binary to decimal:

$$10000111 = 135$$

Simple, right? Not so fast. The IEEE standard also defines an offset for the exponent block, specifically 127. That means, in order to get the power to which we raise 2, we must account for the offset like so:

$$135 - 127 = 8$$

Converting the mantissa to decimal is relatively complex, and it's done using this formula:

$$fraction = 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}$$

In this formula, b_n represents the binary number stored in the mantissa section of memory, with the subscript *n* representing the index of a specific digit of that binary number. The index decreases from left to right, and is zero based, so it spans from 22-0. For converting 428, we get values for *b* like this:

$b_{22} = 1$, $b_{21} = 0$, $b_{20} = 1$, and so forth.

So, our sum will look this:

$$fraction = 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} = 1.671875$$

At this point, we can finally calculate the decimal using this formula:

$$Number = (-1)^0 * 2^8 * 1.671875 = 428$$

Now, this is relatively straight forward example. Exceptions to some of the general rules used in this example occur, such as when calculating numbers that are less than 2, greater than -2, and a few others. However, those situations are outside the scope of this brief document.

Keep in mind that floating point is imprecise at representing certain decimal numbers. For example, if we tried to represent 428.428, the best we could do is (rounded) 428.42801 or (not rounded) 428.42798. If we were to use a double (64-bits or 8bytes in length), we could represent 428.428 properly without having to round (and this is why you should use a *double* instead of a *float* these days, since the price difference between 4 and 8 bytes of memory is insignificant).

The bottom line for you as computer scientists - understand that things are stored in memory in different ways. While the implementation behind storage formats is not necessary for everyday programming, the limits of your data types are important. Storing data too large or too precise for a data type can cause big problems. Always be aware of the size of your data, the precision required for its storage, and always, always, always test your programs thoroughly and completely with a great variety of possible cases.

If you're interested in knowing more about floats, doubles, storage formats, etc., check out these links:

Floating point conversion calculator - http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

Another floating point calculator - http://www.h-schmidt.net/FloatConverter/

Tutorial on floating point binary numbers - http://kipirvine.com/asm/workbook/floating_tut.htm

Wikipedia article on IEEE floating point standard - http://en.wikipedia.org/wiki/IEEE_754-2008

Wikipedia article about two's complement - http://en.wikipedia.org/wiki/Two%27s_complement