ECE 222 System Programming Concepts
Lecture 2 – Bits, Bytes, and Data Types

There are a few basic data types in C: int, float, double, char. We can qualify them as "unsigned", and change their ranges using "short" and "long". All of these data types give us places to store values (variables). But what is the difference between an int and a float? That's easy: One stores whole numbers, the other stores real numbers. How? Or, how about some tougher questions: What's the difference between a float and a double? What's the difference between a short int and an unsigned char? In order to answer these questions, we have to understand the bit models that underlie the data types.

A bit is a binary valued variable. Typical values are 1 and 0, or true and false. On a computing chip (processor, memory chip, etc.), they are represented by high and low voltages, where a high value is typically 1-5V and a low value is typically 0.0-0.5V. Everything in computing is based upon combinations of bits. Everything stored in a computing chip is based upon using a fixed number of bits to model (or represent) the thing of interest.

A byte is 8 bits: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

For our discussion, bits can be one or zero (1 or 0):

For representing whole numbers, the "place values" of bits are as follows:

[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
127	64	32	16	8	4	2	1

A bit value of 0 indicates a place value of 0, a bit value of 1 indicates a place value as given in the picture. The total value of the number represented is found by adding up the place values of all the bits.

For example:

0	0	0	1	0	0	1	1	
0	0	0	16	0	0	2	1	(sum = 19)

Given 8 bits, it is possible to store numbers in the range $0 \ldots \sum_{i=0}^{7} 2^i = 2^8 - 1 = 255$.

The bits with the lowest powers are called the least significant bits, or lowest order bits, because they represent the smallest portions of the number. The bits with the highest powers are called the most significant bits, or higher order bits. It is common practice to list bits from highest to lowest, left to right, following the same convention used to write base 10 numbers.

Binary arithmetic is done similarly to base 10 arithmetic. For example:

```
7          00000111
```

```
+4       +  00000100
--       -----------
11          00001011
```

When the sum in a column is greater than 9 (base 10) or 1 (base 2), then the sum carries over to the next digit.

In the case where signed numbers are desired, a common practice is to allocate the highest order bit to be the "sign bit". This is called the sign-magnitude model.

[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]
s	2^6	2^5	2^4	2^3	2^2	2^1	2^0
+-	64	32	16	8	4	2	1

This provides a range of values from -127…127.

Notice that there are two possible bit values for zero: 00000000 is "positive zero", while 10000000 is "negative zero". This does not make much sense. Even more importantly, allowing negative numbers makes arithmetic computations more complicated. If we have zero or two negative numbers, we perform addition, as above. If we have one negative number, we must instead perform a subtraction.

The two's complement model was invented to overcome both these problems. In two's complement notation, positive numbers are represented as already shown. A negative number is found by the following steps:

(a) write bits for positive version of number
(b) invert all bits
(c) add one

For example, to represent -7, one would:

```
00000111  +7
11111000  invert
11111001  add one
```

When a two's complement number has a 1 in the highest bit, it indicates the number is negative. To find the value, we perform the same steps:

```
11111001  -[something]
00000110  invert
00000111  add one  => -7
```

Two's complement modeling solves the "double zero" problem:

```
00000000  0
11111111  invert
```

```
00000000  add one
```

Note that we only have 8 bits, so the carry out of the highest order bit is lost there. It has no place to go. What happened to "negative zero"? Remember that when a two's complement number starts with 1, we assume it is a negative number, and proceed to find the value:

```
10000000  -[something]
01111111  invert
10000000  add one   => -128
```

This is called the "weird number". When following the two's complement conversion steps, it comes out the same as it started. However, at the beginning, we only look at the highest bit to determine the number is negative. After converting, we see that the value is 128, and so the complete value is -128. Thus, in two's complement, the full range of numbers using 8 bits is -128…+127.

Two's complement also makes addition and subtraction easier to implement, because regardless of the signs of the two numbers, they can always be added. For example:

```
        11111111   (carry bits)
   7    00000111
+(-5)   11111011   00000101 invert 11111010 add1 11111011
-----   --------
   2    00000010
```

Note that the final carry bit was again thrown away. However, it must be checked. For the operation to be valid, the highest two carry bits must be either 11 or 00. If they are either 10 or 01, then the result is an arithmetic overflow. Arithmetic overflow is when the process of a calculation results in a number outside the range of values that can be represented by the available bits. For example:

```
        10000000   (carry bits)
 (-127)  10000001  01111111 invert 10000000 add1 10000001
+(-126)  10000010  01111110 invert 10000001 add1 10000010
  -----  --------
  -253   00000011
```

The result appears to be 00000011, or 3. But in fact an overflow has occurred. With 8 bits, we cannot represent -253 (it is outside our allowed range of -128…+127). We can see that an overflow has occurred by looking at the highest two carry bits, which are 10.

In the C language, the data type qualifier "signed" is the default; "unsigned" must be used to allocate the highest order bit to values. For example,

```
char a1;
```

unsigned char a2;

gives us a 1-byte variable named a1 to store numbers ranged 0…255 (magnitude only numbers), and a 1-byte variable named a2 to stored numbers ranged -128…127 (two's compliment numbers).

Storing such small numbers is not always enough. To have larger numbers, we must use a different data type. The int data type uses 4 bytes to store a whole number. It can also be qualified as "unsigned".

[What range does it allow? Have class figure it out.]
[Answer: 32 bits. Unsigned, 0…4billion. Signed, -2billion….+2billion.]

In the C language, data types can often be qualified using "long", and sometimes "long long", which tend to double the amount of bytes used to store a value.

The sizeof() function reports how many bytes a data type, or a variable, is using.

For storing real numbers, a different system must be used. The most common approach is floating point, in which the general idea is to represent the number using scientific notation (a base and an exponent). For example:

$123.456 = 1.23456 * 10^2$

In a computer we have only bits, so the representation uses an implied (known) base of 2 and a leading mantissa value of 1:

$1.f * 2^e$

Thus, what is stored in the bits are values for e and f. For a concrete example, we look at the IEEE floating point standard for computing. It assumes 4 bytes are used:

total bits	1	8 23
	[s]	[exp] [fraction]
bit	31	30…23 22……....0

The 8 bits for the exponent represent -127…+128. The bits in the fraction represent powers of one over two the N. For example, bit 22 represents ½, bit 21 represents ¼, bit 20 represents 1/8, etc.

To find all the bit values, the following steps are performed. In order to understand them, we will work an example at the same time. We will encode -118.625.

(a) get sign bit	sign bit = 1	
(b) write # in binary (without sign)	1110110.101	(118.625)

(c) normalize, moving radix (decimal pt) to just after the first 1 digit — 1.110110101×2^6

(d) f = right of the radix, zero padded — f = 110 1101 0100 0000 0000 0000

(e) e = given exponent, biased +127 — e = 6+127 = 133 = 1000 0101

The exponent is biased so that the range of powers is half negative (representing fractional numbers less than 1.0) and half positive (representing large numbers).

For our example, the final 32 bit number is 1 1000 0101 110 1101 0100 0000 0000 0000. Note that the spaces are for our convenience only, they are not really there.

In the C language, the float data type uses 4 bytes to store a real number, while the double data type uses 8 bytes to store a real number. The double still uses 1 bit for sign, but it uses a lot more bits for the exponent and fraction, so that it can represent a wider range of really small numbers (fractions) and large numbers.

To summarize, what is 1100 0010 1110 1101 0100 0000 0000 0000 ?

Depends!

It could be 4 unsigned char (magnitude-only) in a row: 194 237 64 0
It could be 4 char in a row (two's complement): -62 -19 64 0
It could be 1 float: -118.625
It could be a lot of other things as well, depending on how many bytes we group (half of a double? two short ints?) and what bit model is applied (two's complement, ASCII?).