Integer Encodings

C supports both signed and unsigned integer types. Depending on your computer system and the C implementation, different ranges may be supported. The following compares the typical ranges of some integer data types provided by C on 32-bit and 64-bit systems with what C guarantees.

Data Type	Min. 32-bit	Max. 32-bit	Min. 64-bit	Max. 64-bit	C Guaranteed Min.	C Guaranteed Max.
int	-2,147,483,648	2,147,483,647	-2,147,483,648	2,147,483,647	-32,767	32,767
unsigned int	0	4,294,967,295	0	4,294,967,295	0	65,535
long	-2,147,483,648	2,147,483,647	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	-2,147,483,647	2,147,483,647
unsigned long	0	4,294,967,295	0	18,446,744,073,709,551,615	0	4,294,967,295

A few things we notice is that the actual ranges provided by an implementation may be larger than what C guarantees. We also notice that the ranges provided may not actually be symmetric. This has to do with how we represent the sign of a number in our binary representation. The C guarantees define a minimum range but depending on the implementation of how numbers are signed, the actual range may be different.

bit vectors

It is helpful to describe our binary numbers as a bit-vector:
v = [v_w-1, v_w-2, ..., v_0] where *w* is the word size.

unsigned encodings

Under an unsigned encoding, every number between 0 and (2^w) - 1 has a unique encoding as a *w*-bit value.

We can express our interpretation of an unsigned integer value through the function B2U_w ("binary to unsigned, length w):

B2U_w(**v**) = SUM{i=0,w-1} v_i * 2^i

Example: Interpret the following bit-vectors as an unsigned integer:

```
[0001]
        = B2U_4([0001])
        =(0 * 2^3) + (0* 2^2) + (0 * 2^1) + (1 * 2^0 )
        =  (0) + (0) + (0) + (1)
        = 1

[1111]
        = B2U_4([1111])
        =(1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)
        =(8) + (4) + (2) + (1)
        = 15
```

Twos-Complement Signed Encoding

under twos-complement encoding, every integer between -2^(w-1) and s^(w-1) - 1 has a unique binary representation.

We express our interpretation of twos-complement encoding through the function B2T_w ("binary to twos-complement, length w):

B2T_w(**v**) = -v_(w-1)*2^(w-1) + SUM{i=0,w-2} v_i * 2^i

We call the most significant bit the **sign bit** and it has a value of 1 when the represented number is negative. it has a value of zero when the value is nonnegative.

Example: Interpret the following bit-vectors as a two-complement integer:

```
[0101]
        = B2T_4([0001])
        = -(0*2^3) + (1* 2^2) + (0 * 2^1) + (1 * 2^0)
                =  -(0) + (4) + (0) + (1)
                = 5

[1111]
        = B2T_4([1111])
        = - (1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)
        =  -(8) + (4) + (2) + (1)
        = -1
```

Conversions

C does allow us to convert between signed and unsigned... but BUYER BEWARE.

what's wrong with this code?

```
float sum_it(float *vals, unsigned int length) {

  int i;
  float result=0;

  for(i=0;i<length-1;i++) {
    result += vals[i];
  }

  return result;
}
```

ANSWER: the arithmetic length - 1 is performed as an unsigned int. 0 - 1 for an unsigned integer will actually produce UMAX (the maximum value for an unsigned int). the for loop will very quickly read past the array boundaries and (likely) generate a segmentation fault.

As a rule, C leaves the underlying bits unchanged when converting between data types of the same word length... however that does not mean that how the values are interpreted will not change!

Endianess

There are two conventions for ordering the bytes representing an object. It is system dependent which means we generally only care when we are transmitting data between systems (such as networking operations). In a **little endian** system, bytes of an object are logically ordered in memory from the least-significant byte to the most significant byte. In a **big endian** system, the bytes are ordered logically from most signficant to least significant.

```
Consider the object represented by 0x01234567

Big-endian:
        Memory Location:        |  0x100        |  0x101        |  0x102        |  0x103        |
        Bytes:                  |    0x01       |    0x23       |    0x45       |    0x67       |

Little-endian:
        Memory Location:        |  0x100        |  0x101        |  0x102        |  0x103        |
        Bytes:                  |    0x67       |    0x45       |    0x23       |    0x01       |
```

Under most circumstances, this will not cause you any headache as a developer. The three main conditions when it appears are:

1. **Looking at Machine Code**: You will need to know the endian-ness of your system or you will be reading the HEX values of the machine code in the wrong byte ordering.
2. **Transmitting across a network**. For network programming, there are defined protocols for handling endianess in order to avoid big-endian and little-endian systems from having communication trouble.
3. **When doing evil things in C**: Well, not really evil, but rather clever but potentially dangerous things such as circumventing strict data typing via casting, etc.

Integer Arithmetic

We must take care when performing arithmetic in a computer system because unlike, arithmetic on paper where our answer and storage space is infinite, on a computer system it is finite. **Overflow** is when an arithmetic operation results in a value that is outside of the range defined for the data type. In C, airthmetic that results in overflow is not treated as an error condition. Behavior is undefined for signed values and results in a modulo 2^w being applied to the result.

for *unsigned integers*, addition can be defined by:

```
x + y = {  x + y                 |        x+y < 2^w
           x + y - 2^w           |      2^w <= x + y < 2^(w+1)   }
```

We deal with signed numbers represented with two's-complement in a similar way, but we have to deal with two cases: when the result is a positive overflow and when the result is a negative overflow

```
x + y = {  x + y                 |        -2^(w-1)<= x + y < 2^(w-1)     Normal
           x + y - 2^w           |         2^(w-1) <= x + y                    Positive Overflow
           x + y + 2^w |       x + y < -2^(w-1)                          Negative Overflow  }
```

Examples:

x	*y*	*x* + *y*	*x* + *y* (2C)	Description
-8 1000	-5 1011	-13 1 0011	3 0011	Negative Overflow
-8 1000	-8 1000	-16 1 0000	0 0000	Negative Overflow
-8 1000	5 0101	-3 1101	-3 1101	No Overflow
2 0010	5 0101	7 0111	7 0111	No Overflow
5 0101	5 0101	10 1010	-6 1010	Positive Overflow

Multiplication

For multiplication, we take the product of x*y and modulo 2^w:
(x*y) mod 2^w