Data Representation in C

How do we interpret the number 742? The number is:

- 7 hundreds
- 4 tens
- 2 ones

Different bases are more natural in certain situations: base 10 is convenient for humans, but binary is more convenient in digital systems (*on* and *off*).

Base 2 (binary) has 2 symbols) (0, 1). Decimal has 10 symbols (0-9) and hexadecimal has 16 symbols (0-9 + A-F). Note that for any base n, the symbols represent a range $(0, n-1)$.

As a generic representation, multiply each digit by its weight, then sum the results. Mathematically if $d_0 d_1 d_3 d_4$ is in base n:
$d_0 d_1 d_3 d_4 = d_3 \times n^3 + d_2 \times n^2 + d_1 \times n^1 + d_0 \times n^0$

In a computer, data are stored as binary digits in fixed-size cells, called *words*. An 8-bit word is usually called a *byte*, and is an extremely common size for a memory cell. 4 bits, in a burst of cuteness, is called a *nybble* and represents a single hexadecimal digit.

Decimal to binary conversion

In [1]:

```c
#include <stdio.h>
#include <stdlib.h>

// convert decimal numbers to binary
unsigned long long decimalToBinary(int decimalnum){
    long remainder = 0;
    long temp = 1;
    unsigned long long binarynum = 0;

    while (decimalnum > 0){
        remainder  = decimalnum % 2;
        decimalnum = decimalnum / 2;
        binarynum = binarynum + remainder * temp;
        temp = temp * 10; // shift left one column
    }
    return binarynum;
}

// print the first 15 binary numbers
int main(void){
    for (long i = 1; i <= 10; i++){
        printf("%6ld = %20llu\n", i, decimalToBinary(i) );
    }
    return EXIT_SUCCESS;
}
```

```
     1 =                    1
     2 =                   10
     3 =                   11
     4 =                  100
     5 =                  101
     6 =                  110
     7 =                  111
     8 =                 1000
     9 =                 1001
    10 =                 1010
```

In [1]:

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
    printf("int type has %lu bytes\n", sizeof(int)); // use %lu to print long unsigned integer
    printf("long type has %lu bytes.\n", sizeof(long));
    printf("long long type has %lu bytes.\n", sizeof(long long));
    printf("float type has %lu bytes.\n", sizeof(float));
    printf("double type has %lu bytes.\n", sizeof(double));
    printf("unsigned long long type has %lu bytes.\n", sizeof(unsigned long long));
}
```

```
int type has 4 bytes
long type has 8 bytes.
long long type has 8 bytes.
float type has 4 bytes.
double type has 8 bytes.
unsigned long long type has 8 bytes.
```

Signed Magnitude

Note that in a signed integer type, we *lose* one bit to represent the sign. In a naive implementation, we would just use the first bit in the word to represent a negative sign:

Cell contents	Value in Base 10
011111111	127
...	...
000000000	+0
100000000	-0
...	...
111111111	-127

It is bothersome that this implementation gives us two different representations of zero. Maybe we can do better.

A better implementation, used in basically all digital logic, is called *two's complements* format. We negate a binary number by:

- flip all the bits
- add one

Cell contents	Value in Base 10
00000000	0
00000001	1
...	...
01111111	+127
10000000	-128
...	...
11111110	-2
11111111	-1

Characters

Characters are also represented by binary values, or *character codes*. C by default uses ASCII, the *American Standard Code for Information Interchange*, which includes 95 letter characters and 30 control codes. It has 128 values and fits in 7 bits.

Floating Point

Binary numbers can have a *binary point*, where digits to the right are fractional and digits to the left are whole, analagously to the decimal point.

Some decimal fractions produce a repeating fraction when converted to binary. To store these values in a fixed-size cell, it must be truncated, and will produce a small error.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
    int s = 0;
}
```

Normalized Scientific Notation

To store a normalized binary number in a 24-bit word we use 16 bits for the base (*mantissa*), with an exponent:

- 8 bit exponent
- 16 bit mantissa
  - 1 sign bit (0 is positive, 1 is negative)
  - no leading one/binary point
    - this is probably equivalent to incrementing the exponent
  - 15 bits store the fractional point of the mantissa

In [ ]:

```
```