# Lab Week #2
# CIS 314

October 18, 2017

## Binary Refresher[1]

Apparently, this is the first course where people use binary (which feels insane, but that's beside the point), so it makes sense to do a little refresher. Since it's not easy to think in base 2, it's probably easiest to think back to how we all learned to count in base 10.

### Base 10

Base 10 consists of the digits 0-9, and every digit is a power of ten – the "first" digit has a value that is computed as $a \cdot 10^0$. For example, the value of $4 = 4 \cdot 10^0 = 4 \cdot 1 = 4$... wow! Similarly, bigger numbers can be seen as digits multiplied by 10 to a certain power. We can now see that the value of $385 = 3 \cdot 10^2 + 8 \cdot 10^1 + 5 \cdot 10^0 = 300 + 80 + 5 = 385$ (now is the time to say you're bored).
Interesting things happen, though, when we want to add 1 to 9... we 'overflow' – this results in us flipping the ones place back to 0, and incrementing the tens place from 0 to 1, meaning $9 + 1 = 10$. Okay, maybe it's not *that* interesting...

### Base 2

Let's now consider binary – we only have two digits, 0 and 1, and every digit is now a power of 2, so the number $1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8_{10} + 4_{10} + 0_{10} + 1_{10} = 13_{10}$. Similarly, when we want to add $1_2 + 12$, we need to overflow and jump to the next bit – we end up with $1_2 + 1_2 = 10_2$. It's not so different after all! We can see how this can be generalized to base $n$, we only need $n$ distinct digits, and then each digit becomes $n$ raised to some power.

## Binary Printer

In the first assignment, we saw how to print the bytes associated with particular datatypes. Unfortunately, it printed things in the byte order that our computer stores them, i.e. little endian order. This means that simple numbers like an `int a = 7` would print as `07 00 00 00`... gross! Wouldn't it be great to see the binary representation of datatypes, but in the correct order?
To accomplish this, think about the following example:

---

```
char a = 7; //The associated binary looks like 0000 0111

//If we want to output the binary, it's as easy as asking every bit if it happens to be a '1'

//To do that, we want to print the most significant bit first, so we could write the following code

printf("Is most significant bit zero? %d\n", a>>7 & 1);

//The above line works by shifting 'a' to the right 7 bits, and then 'and'ing the least significant bit by
    1 -- since C doesn't have boolean types, we will output either 0 (false) or 1 (true)

//We could repeat the above 8 times to print all the numbers

int i;
for( i=7; i>=0; i-- ){
  printf("%d", a>>i & 1);
}
```

---

[1] This would ideally appear in lab 1, but I forgot that people didn't know binary...

Wow! We now have a loop that will print the binary associated with a `char` datatype. What happens if we set `a` to `'a', 127, 128`... why?
The above code can be modified slightly (and moved in to a function) to accept a 32 bit datatype:

```
#include <stdio.h>
void _32bitBinaryPrinter(unsigned a){
  int i;
  for( i=31; i>=0; i-- ){
    printf("%d", a>>i & 1);
}
  printf("\n");
}
int main(){
  unsigned a = 8;
  float b = 8.0;
  float c = 8.5;
  _32bitBinaryPrinter(a);
  //_32bitBinaryPrinter(b);
  //_32bitBinaryPrinter(c);
return 0; }
```

What happens if we uncomment the line with arguments `b` or `c`? When we send `b` in as an argument, we could be convinced that it's right – it looks like it's printing 8 after all, but something is definitely wrong when we send in `c`!

## Unions

The problem with the previous code is how C treats casts from floating point datatypes to integer datatypes – in essence, the decimal portion is truncated. This is obviously no good if we want to see how `8.5` is represented in the computer. Assignment 2 has a neat little function that is reproduced below:

```
unsigned f2u(float f) {
  return *((unsigned*)&f);
}
```

Huh?!? If we break it down in to several parts, we can understand how the above works:

```
unsigned f2u(float f) {

  unsigned* a = (unsigned*) &f; //Treat the address of f as pointing to an unsigned integer


  unsigned b = *a; //Dereference the value that a is pointing to


  return b;
}
```

The gist of the function is that we need to convince the compiler that the address of a floating point datatype is actually pointing to an unsigned datatype (wink wink), and then we dereference the value – this returns us an unsigned integer with the same bit pattern as the original floating point. What happens if we now send a float through this wacky funciton, and then through our binary printer:

```
float c = 8.5;

_32bitBinaryPrinter(f2u(8.5));
```

As it turns out, we could use a second method to accomplish the goals of the function `f2u`:

```
unsigned f2u(float f){

  union u{
    float f;
    unsigned u;
  };

  union u u1;
  u1.f = f;

  return u1.u;
}
```

The above takes advantage of a C construct called a *union* which can be thought of as a datatype which allows us to store more than one piece of information in the same memory location – in this case we are telling it to be able to store both an unsigned int and a float int the same four bytes. By populating the union with a float, and then returning an unsigned int, we are accomplishing the same thing as the pointer variant of `f2u()` used in the assignment.
This can be simplified a little bit through the use of `typedef` – recall that `typedef` takes two arguments, the second is some alias, and the first is what we want the alias to refer to. Consider the following:

```
typedef unsigned long long int my_type; //unsigned long long int is the worst to type, I want to type
    my_type
```

We can use the same to simplify the union example from above:

```
unsigned f2u(float f){

  typedef union{
    float f;
    unsigned u;
  }u;

  u u1;
  u1.f = f;

  return u1.u;
}
```