Lab Week #1
CIS 314

September 29, 2017

Hello World!

Our first program, *Hello World*, is simple enough:

```
#include <stdio.h> //need this library to do input and output

int main(int argc, char** argv){ //the signature of main always looks like this

  printf("Hello World!\n");

  return 0;
}
```

Once we've saved the above program somewhere convenient (I've named my file `lab1.c`), we can compile and run it by opening the terminal and typing `gcc lab1.c`, then we can run the produced executable with the command `./a.out` (that would be `./a.exe` for Windows users). Easy!

Everything is Bits

C is likely the first language you will encounter that peels back the layers of abstraction – we'll be dealing with bits directly! To motivate this, imagine we have the following line of code:

```
//Declare a variable named 'a' that occupies one byte of memory and holds the decimal value 50
char a = 50;
```

We can now use C to show us a couple interesting things:

```
char a = 50;


printf("a = %d\n", a); //print 'a' as an integer

printf("a = %c\n", a); //print 'a' as a character
```

The second print statement ends up printing '2'! If that seems wrong, pull up an ASCII table[1] and stare at it for a *bit*. The main takeaway from this section is to note that we are storing bits and then telling C how to interpret those bits – nothing changed in the hardware for us to print both 50 and '2', only how we told the computer to interpret those bits changed.

Datatypes

Datatypes in C can (essentially...) be thought of as containers of varying size – the smallest being `char` which occupies one byte, up to `long long` which holds 8 bytes[2]. Conspicuously missing from the list of built in datatypes is bool. In C, integers are used for boolean values, 0 being false and nonzero values being true.

Signed vs. Unsigned

Additionally, C has a concept of *signed*-ness, signed datatypes being capable of holding negative numbers (the exception being with the floating point datatypes which can hold positive and negative numbers without any special identifier).

[1] I've been using `http://www.asciitable.com/`

[2] Yes, we can hold much more in a dynamically allocated array, but we're not there yet!

In general, specifying just a datatype (i.e. `char`) means that you will have a signed datatype, and specifying *just* `unsigned` means you have an `int`.

Bitwise Operators

The first interesting bitwise operator is left shift (denoted `<<` in code), which has the effect of multiplying the shifted number by 2 for every bit shifted (i.e. a left shift of n results in a multiplication of 2^n).

More interesting than that is the case of right shift. Ideally, we would want right shift to be the "opposite" of left shift – we want something to divide by 2. This works just fine with positive numbers, but negative numbers prove more of a challenge. As it turns out, there are two types of right shift to deal with these two cases, called *arithmetic* for signed datatypes and *logical* for unsigned datatypes. In C, we don't have to worry about two different syntaxes, but we absolutely need to know what's going on[3]!

The next bitwise operators all *seem* familiar, but are subtly different than the logical operators you've previously encountered. Consider the following example:

```
char a = 0;
char b = 2;

printf("a||b = %d\n", a||b);
printf("a|b = %d\n", a|b);
```

The first print statement should have been expected, as we know that 0 is false in C, and 2 is true, so 0||2 produces 1 (true). The second is perhaps more interesting – the computer is performing a logical 'or' on each pair of bits directly, like so:

```
a   =   0000 0000
b   =   0000 0010
----------------
output 0000 0010
```

Now we can clearly see how it's ending up with 2! The same sort of idea is how bitwise 'and' (`&`), 'xor' (`^`), and 'not' (`~`) work. Be careful not to use the logical not instead of bitwise not!

Typedef

If you're following along in the book, you may come across the keyword `typedef`, which allows the programmer to make aliases of existing types. For example, if my program made judicious use of `unsigned long long int`'s, I may want to alias that type to something shorter to write such as `myInt`. I could accomplish this with a `typedef` as follows:

```
typedef unsigned long long int myInt;
```

Endianness

Looking at page 42 in the book, if we run the code as written and test it on something (for example running `show_int(7);`), we will see that the output is inexplicably reversed (my computer shows `07 00 00 00`). This is due to the concept of *endianness*, where my computer stores the least significant byte in the lowest memory location. If you have a big endian computer, I want to see it!

[3] As always, Wikipedia is helpful `https://en.wikipedia.org/wiki/Arithmetic_shift`