Assignment 2a: Bits and Bytes

Kudos to Julie Zelenski for this great problem set!

Bits and Bytes

The first four weeks of CS107 is all about solidifying your understanding of C: arrays, pointers, typecasts, parameter-passing, etc. and then supplementing it with a better understanding of compile-time language implementation: the layout and organization of memory, what kind of code is generated from a compiler, which things happen at compile-time and which at runtime, how the runtime memory structures (stack and heap) managed, and so on. The implementation problem set is a break from the hard-core C coding and a chance to try your hand at generating machine code and to do some in-depth experimentation with the compiler, debugger, and Purify in order to dissect your programs and to better understand how these tools work and what exactly is going on.

To Be Completed: October 30th

You aren't required to formally hand in anything for Assignment 2. I will give Assignment 2 out in pieces over a few weeks so that you can use the exercises as a diversion from the C programming. Work through them to verify your understanding of the material from lecture as well to solidify your understanding of complex programming issues, not unlike those you are dealing with in Assignment 1. The "deadline" for completing it will be the midterm, since problems of this type are on the exam and you will want to be adequately prepared to crush all those questions. By removing the requirement that is handed in and graded, we are trying to save you the effort required to formally write up your solutions that often takes more time that just solving the problems in the first place. You are also encouraged to freely collaborate with others on these exercises.

How to compile a program without a makefile

You may want to write little test programs for the lab problems. You're used to relying on our provided makefiles we wrote to build multi-module programs, but it is possibly to just directly invoke the compiler at the shell. Refer to the big UNIX handout for lots of details— here is just a brief summary:

- To compile and link one file into an executable, you use:

  ```
  % gcc binky.c -o binky
  ```

  which says to compile the file `binky.c`, link it, and name the resulting executable `binky`.

- To compile multiple files into one executable, you first compile each file separately:

  ```
  % gcc -c scanner.c -o scanner.o
  % gcc -c analyze.c -o analyze.o
  ```

  This compiles the file `scanner.c` into the output file `scanner.o` and stops without linking it. (same for `analzye.c`). And then you invoke `gcc` again to link like this:

  ```
  % gcc scanner.o analyze.o -o analyze
  ```

  which takes the two compiled object files and links them together into one executable called `analyze`.

- You can also add other flags for compiling after the gcc such as `-Wall` (to generate all warnings), `-g` (to include debugging information, and `-O` (for optimization).
- In order to build a Purify version of an executable, you use the `gcc` link command you would as above, but prefix it with `purify`, like this:

  ```
  % purify gcc binky.c -o binky.purify
  % purify gcc scanner.o analyze.o -o analyze.purify
  ```

Problem 1: Binary numbers and bit operations

Since computers work entirely in binary, learning how to manipulate numbers in the base two system can sometimes come in handy. Here are a few suggestions of some exercises to test out your understanding of binary numbers.

- Try converting a few decimal numbers into binary form, do some binary arithmetic with them (add, subtract, maybe even a multiply), and convert the result back to decimal to verify you have it correct. This lets you know you are on top of the basic workings and know how to do all that tedious carrying.
- Write a test program to find out what happens when you overflow the range of a variable, such as adding two shorts that are both very large. Is any error reported on the overflow? How is the result related to what the desired answer would have been? What happens when you do the same thing with unsigned shorts instead of signed?
- Write a program that assigns values between different-sized integer types. What happens when you go from a smaller-sized type to a larger? What about the other direction? How is the result related to the original number? Does this match your understanding of the binary representation?

In addition to the usual logical AND, OR, and NOT connectives, there are bitwise versions of these operations available in C. The bitwise AND (expressed with single &) compares its two operands bit-by-bit and reports which bits were 1 in both, 0 otherwise. For example:

```
unsigned char a = 12, b = 5, c;
c = a & b;
```

Doing a bitwise AND on 00001100 (12 in binary) and 00000101 (5 in binary) gives the result 00000100, since the two patterns have only one bit in common.

There is a bitwise OR operator (single |) that works similarly to logical OR, but operates at the bit level. There is also a bitwise exclusive OR (^) that reports which bits are on in one operand, but not both. The bitwise NOT (~) is a unary operator that just inverts all of the bits in its operand. Bit manipulations are used in a variety of situations (graphics, robotics, cryptography, etc.) especially when you need to work with a form of packed data.

- How could you use bit operations to determine the remainder of a number when divided by 4 (or any power of two for that matter?) How could use a bit approach to determine if an integer would lose data when assigned to a `short` or a `char`?
- How can you use bit operations to convert a number from negative to positive or vice versa? (Refer back to the “two’s complement” representation for negative numbers we showed in class and try to work through what the patterns are in terms of bits.)
- One neat feature of the bitwise XOR operation is that it is completely invertible. If you XOR `a` with `b` and then XOR the result with `b` again, you get back `a` (trace this

out for yourself). This makes this operation useful for encryption / decryption. How could you construct a simple program that encrypts a file using XOR and a specified "key"? What would happen if you run the program twice in a row using the same key?

Problem 2: Trix are for kids

This puzzle is reproduced from a cereal box "magic trick". Get our your scissors and prepare to win yourself a few bets. Cut out the cards below. Have a volunteer pick a number between 1 and 63, hand these cards to them, and ask them to give you back only those cards on which their chosen number appears. In a feat of amazing wizardry, you then immediately tell them what their number is. How do the cards work?

1	3	5	7	9	11
13	15	17	19	21	23
25	27	29	31	33	35
37	39	41	43	45	47
49	51	53	55	57	59
61	63				

8	9	10	11	12	13
14	15	24	25	26	27
28	29	30	31	40	41
42	43	44	45	46	47
56	57	58	59	60	61
62	63				

2	3	6	7	10	11
14	15	18	19	22	23
26	27	30	31	34	35
38	39	42	43	46	47
50	51	54	55	58	59
62	63				

16	17	18	19	20	21
22	23	24	25	26	27
28	29	30	31	48	49
50	51	52	53	54	55
56	57	58	59	60	61
62	63				

4	5	6	7	12	13
14	15	20	21	22	23
28	29	30	31	36	37
38	39	44	45	46	47
52	53	54	55	60	61
62	63				

32	33	34	35	36	37
38	39	40	41	42	43
44	45	46	47	48	49
50	51	52	53	54	55
56	57	58	59	60	61
62	63				

Probelm 3: ASCII and extended ASCII

In lecture, we showed the binary polynomial representation used for the integer-derived types. Characters belong to the integer family and although computers agree on the bit pattern used to represent the number 10 or 250, the mapping from number to character is where things break down. The ASCII character set establishes mappings for 0 to 127 that are used by all computers, but the extended ASCII characters, from 128 to 255, varies from system to system.

In `/usr/class/cs107/assignments/hw2`, there is a file called `ascii.txt` that contains a table of all possible characters. Try viewing this file on UNIX and then again on Macintosh or a PC (either by using ftp to transfer the file or viewing the file with a Web browser) and observe how the exact same bit pattern can be interpreted as different characters on different systems. Which number-to-character mappings seem to be reliable? Which are not?

What implications does this have for using extended ASCII characters in a Web page intended to be viewed by all? What about sending those characters in e-mail exchanged between different computers? Does this exercise help explain any weird character translation problems you may have run into in the past?

Problem 4: Floating point numbers

Floating point numbers use an entirely different representation than the binary polynomial used by integers. As we talked about in class, the IEEE floating point standard is a form of scientific notation where some bits are used for the mantissa and others for the exponent. In addition to allowing fractional values to be stored, the `float` can represent a much greater magnitude than the int (typically up to 10^{38}). The IEEE standard also prescribes how exceptional conditions such as overflow and underflow must be handled.

Start by opening the `<float.h>` header file (available in `/usr/include/float.h` on the UNIX machines) to get familiarized with the specifications for floating point on this machine. These constants give the representable range of numbers, minimum and maximum exponents, and decimal digit of precision for the types.

- What happens when you overflow the floating point range? Try multiplying FLT_MAX by 2 and trying to store the result in a float. How does this differ from how integer overflow is handled?
- What happens when you try to divide a float by zero? What if you try to divide zero by zero? How does this differ from the way integer division handles these cases?

Although the range for a `float` is much larger than an `int`, it is necessarily imprecise. There are an infinite number of values within the range of representation and only a finite number of distinct bit patterns to represent them in. Therefore calculations with floating point numbers are limited in accuracy, with only about 6 decimal digits of true precision (15 for doubles), despite how many digits are printed out by `printf`. Here are some experiments that will help demonstrate how that inaccuracy might show up:

- Sometimes numbers that shouldn't be equal will appear to be so. Take a very large `float`, let's say 90% of `FLT_MAX`, and add to it a pretty small number, like `100`. Compare the result and you'll find that the addition appeared to have no effect. To add, the two numbers were normalized to the same exponent, and converting 100 to match the 38th power of the large number ends up being approximated by zero because the mantissa is just too small.

- Sometimes numbers that should be equal won't be. Comparing two floats with == will only tell if you if they have exactly the same bit pattern, but depending on what calculations created the number it might have a slightly different value. The following simple program is up in `/usr/class/cs107/assignments/hw2/floats.c`. Compile and run it to get the results.

```
int main(void)
{
   double d1, d2, d3, d4;

   d1 = 0.2;      // start with constant value of .2
   d2 = 1.0/5.0;     // three different ways of computing .2
   d3 = .02/0.1;
   d4 = 0.3 – 0.1;
   printf("Does %g == %g? %d\n", d1, d2, d1 == d2);
   printf("Does %g == %g? %d\n", d1, d3, d1 == d3);
   printf("Does %g == %g? %d\n", d1, d4, d1 == d4);
}
```

  To properly compare floating point numbers, you compare the absolute value of the difference between the two and if it is sufficiently small, you consider the two equal. How small should the epsilon be? See `float.h` for the constants `FLT_EPSILON` and `DBL_EPSILON` that give the appropriate values. Try changing the above program to use a new function of your own creation, `DoubleEqual`, that does just that instead of ==. Do all the comparisons pass the new test?

Problem 5: It's just bits and bytes

One key idea we've are stressing is that memory is just one big glob of bytes. You can't tell whether the byte at address `0x1024` is a character or the first or third byte in an integer, whether it's initialized, whether it's in use, or anything meaningful at all from the bits stored there. Many bit patterns will have reasonable values in several interpretations. The `x` command in `gdb` will allow you to examine memory in all variety of interpretations (try "`help x`" in `gdb` for details on how to use it), so you can try "What if the contents of `0x1024` were a `float`, what would its value be? What if it were a machine instruction?" and so on. Try these few experiments and report your findings:

- Put the string `"hi!"` somewhere in memory and use gdb's `x` command to see how those 4 characters would be interpreted if treated as an integer. What about as a machine instruction?
- Use `x` to learn what the integer bit pattern `3` would be re-interpreted if treated like a `float`. Does the result turn out to be `3.0`?

This type of `X` operation is also available at runtime using a typecast. For safety reasons, most languages don't expose the typecast mechanism to the programmer and therefore constrain type conversion to a limited set of operations that actually make sense. C, on the other hand, allows the typecast to be freely used in (almost) all situations, allowing the programmer immense control over interpretation of data, but also providing opportunity for lots of errors. You can do a lot of very twisted and weird things with typecasts, most of which you probably shouldn't even want to do. To show off your new prowess as a master of data manipulation, here are a few pressing needs you can solve. Try to construct a code snippet using a typecast that will:

- Print out the contents of a 4-byte `struct` as though it were a 4-byte `float`.
- Directly copy the first four characters of a string and assign them into an integer variable.
- Report whether the architecture is big or little endian. Recall big-endian means the most significant byte of a multi-byte value is at the lowest address, vice versa for little-endian.
- Lastly, construct a code snippet that will take the value of a 4-byte `float` and show what the same bits would read if treated as an integer, but without using a typecast. (Hint: exploit the untyped nature of the `printf` variable-argument function. Does this help you to see why variable-argument functions are inherently unsafe? Huh?)