Carolyn Cotterman
Stat 243
October 4, 2010

Assignment 1

1 Introduction

In completing this assignment, I have learned basic Unix commands and have also learned how to write, compile, run, and collect the output of C programs. This assignment has furthermore taught me about the different types of numeric variables – their representation in binary form, their limitations in terms of the range of values that they can take, and the output formats (using printf) that are appropriate to use with each type. I have also learned about function prototypes and variable scope. Finally, I learned how to use functions located outside of the standard .h directory.

2 Function Prototypes

In answering this question, I discovered that it is important to indicate what type of parameters a function is going to receive through a proper function prototype. I created a program that was designed to add 3 plus 4. When I declared sum as

```
double sum();
```

then my output was

```
the sum is: 0.000000
```

C didn't know what I was passing to the function sum and so converted 3 and 4 to integers before passing them to sum since the values 3 and 4 appear to be integers if no other information is provided. The function sum was expecting to be passed doubles, however, which take up more memory in the stack. So, when it got integers rather than doubles, it looked at the wrong area of memory in the stack, which corresponded to 0.000000.

When I provided the correct function prototype for sum, then I was returned the proper ouput:

```
the sum is: 7.000000
```

Please see the following files containing the code that corresponds to this question:

- HW1_Q1_main.c
- HW1_Q1_function.c

3 Determining Machine Constants

My general strategy for answering this second question was to write a program that first assigned the value of 1 to a variable. I wrote a loop that would then double (or halve) the variable's value each time the loop was executed until doubling the variable's value caused the variable to return an incorrect answer (i.e., not actually double the previous value.) This incorrect answer was an indication that I had surpassed the largest value of the variable that the computer could handle. Depending on what type of variable I was dealing with, the incorrect value took on different forms, as described below. Please find the following attachments for the code and output corresponding to this question:

- H1_Q2.c (source code)
- H1_Q2_out.txt (output)

- **Largest Integer:** We learned in class that a [long] integer consumes 8 bytes of space on a 64-bit machine. In other words, an integer is represented by a string of 32 zero/one bits, corresponding to 2^{32} possible values. We also learned that the left-most digit of an integer's binary representation is interpreted as a negative sign when it has a value of 1 (if this left-most digit is 0, then the integer is considered positive.) So, since the left-most digit is reserved for designating the sign of the integer, then an integer can take 2^{31} different non-negative values. If one of these non-negative values corresponds to zero, then the largest integer that can be handled is $2^{31} - 1$. I verified that this is the case by doubling my variable's value (starting at the value of 1), and stopping when a negative number was output (indicating that the left-most binary digit had been reached). The largest value was then calculated as $(i_{prior} - 1) * 2 + 1$ where i_{prior} is the value of my variable immediately before doubling it returned a negative number. This value corresponds to **2147483647**. A short [integer] is represented similarly, though with fewer bytes. I therefore employed a similar strategy for finding the largest value that a short can take, and discovered that it is **32767**.

- **Largest Unsigned Integer:** An unsigned integer is represented in binary code similar to how a signed integer is represented, except that none of the bits are interpreted as a sign (an unsigned integer is never interpreted as negative.) So, to find the largest unsigned integer value that can be handled by my computer, I could employ a similar strategy to that used to find the largest signed integer value that can be handled. The only difference is that rather than expecting a negative number when my computation pushes the value of an integer past its limit, I expect the value of zero to result. In employing this "stopping rule" for my while-loop that doubles the value of my created integer, I find that the max value that can be properly handled by my computer for a long unsigned integer is **4294967295**, and for a short unsigned integer it is **65535**.

- **Largest Floating Point Number:** Finding the largest floating point numbers (of both single and double precision) that can be handled by my computer was somewhat

trickier than finding the largest integers; doing so mandated that I understand how floating point numbers are stored by my machine (in binary representation.) I then understood that I needed to (a) find the maximum value that the exponent component of the float could take, (b) find the maximum value that the mantissa component could take, and (c) combine these values through a simple multiplication. When I exceeded the maximum for the exponent component, I was returned 'inf'; when I exceeded the overall maximum for a float (exponent component multiplied by the mantissa), I was returned an unchanged value (despite having added a factor of 2 to the mantissa). My results indicate that the maxium single precision float that can be handled by my computer is $3.402823466385288598117041834845169254 40 * 10^{38}$, and the maximum double precision float that can be handled by my computer is $1.797693134862315708145274237317043567980706 * 10^{308}$.

- **Machine Epsilon:** To find the machine epsilon, I started by defining a double with the value of one. I created a loop that with each iteration halved the value of my double, and added the result to the number 1. Following the definition of machine epsilon, provided by this assignment, I knew that I had gone beyond the machine epsilon when I was returned a value of 1 when I added a candidate machine epsilon value to a double of value 1. I conclude that my machine epsilon is $1.1102230246251565 * 10^{-16}$.

4 Probability Tables for the χ^2 Distribution

For this question, I wrote a program that prompts the user for degrees of freedom and probability level, and which returns the critical χ^2 value. This program calls a FORTRAN subroutine called dcdflib. Please see the following attachments for my code and output examples:

- HW1_Q3.c (source code)
- HW1_Q3_out.txt (output)

5 Conclusion

This assignment demonstrates that C can be tricky for an unexperienced programmer, but there are good reasons for even the features that cause some amount of pain the beginning: efficiency and flexibility are two attributes of the C language that I will likely appreciate with time.