CMPE-380: Applied Programming

Laboratory Exercise 09

Floating-point Numbers and Function design

Table of Contents

Floating Point Numbers

Pre-Lab – 10 pts

In this lab session, we will review Floating Point, and Complex Numbers. Review the class notes on IEEE floating point, complex numbers and numerical computing.

Calculate the following hex "&" masks based on your understand of IEEE single precision format (32 bit) (e.g. A "&" mask of 0x00000003 "anded" with a 32 bit integer number "X" will extract the bottom 2 bits of X):

```
#define SIGN_MASK              // extract the sign bit
#define SIGN_SHIFT             // The number of bits to shift the sign mask
#define EXPONENT_MASK
#define EXPONENT_SHIFT
#define EXPONENT_BIAS
#define SIGNIFICAND_MASK
```

Put these macros in your **prelab.txt** file

Your TA will verify your **prelab.txt** file and conduct a brief oral quiz on the basic IEEE floating point fields. The TA may ask questions about the: sign bit, (mantissa) significand, bias and implied 1 features. Your TA will also conduct a brief oral quiz on scientific function design. The TA may ask questions about the relative value of design for chaining versus performance design using pass by reference or pass by value. The TA will as why returning pointers to data on the local stack is dangerous.

Interactive Examples – 30 pts

IEEE format

In this exercise will examine the IEEE format and extract the key parts. We will be using the "and" masks you created in the prelab to extract the key IEEE format fields. We require a union because C won't let us mask floating point values. You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of the exercise before redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

This subroutine will require "bit and" and "bit shifting operations. The bit and symbol is "&" and the bit shift symbol is ">>" (right) or "<<" (left).

e.g   var2 = var1 & 0x00000001 – will extract the last bit
  var 2 = var 1 >>1   - will shift var1 by 1 bit to the right which is like a divide by 2.

1) Modify the provide lab_a.c code to print out the IEEE fields. Your code will REQUIRE a union containing a float and an integer. Create the union in the ieeePrint subroutine.
2) Use the provided variables: sign, exponent and significand in your calculations. Use the masking and shifting values you identified in the prelab. A print format string is provided. Note: The significand must be left shifted by 1 because it is only 23 bits long and is left justified.

3) Compile and test your IEEE print routine with the following examples saving the results in **exercise.txt**:

```
./lab_a 0
        Number 0.000000 => Bin sign:0 Dec exponent:-127  Hex significand: .0
./lab_a 1

        Number 1.000000 => Bin sign:0 Dec exponent:0  Hex significand: .0
./lab_a -1

        Number -1.000000 => Bin sign:1 Dec exponent:0  Hex significand: .0

./lab_a 1.5

        Number 1.500000 => Bin sign:0 Dec exponent:0  Hex significand: .800000
./lab_a 1.25

        Number 1.250000 => Bin sign:0 Dec exponent:0  Hex significand: .400000
./lab_a 1.125

        Number 1.125000 => Bin sign:0 Dec exponent:0  Hex significand: .200000
```

IEEE limits

In this exercise will examine the IEEE format and the limits of floating-point calculations. We are going to write code to determine the smallest single precision value IEEE can support. Our technique is to start with a floating-point increment value and keep cutting it by half until cutting it no longer changes the answer. The smallest possible increment value is called Epsilon. You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of the exercise before redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

1) Modify the provided code lab_b.c to cut the size of epsilon by two in each iteration. Save the value of the previous epsilon before cutting the size. The last "previous epsilon" is the smallest possible single floating-point number. Use %2.6g format to clearly print your value of Epsilon.
2) Each floating-point format has an Epsilon which is documented in math.h. Add a print statement to the output to clearly print "FLT_EPSILON". Your epsilon should match the system epsilon.

3) Expand your code to calculate the double precision format epsilon as indicate in the code. Copy your while loop code and eliminate the "float" casts in the code, print your Epsilon and the system "DBL_EPSILON". ". Your epsilon should match the system epsilon.

4) Redirect the output of your program to **exercise.txt**

Function design

In this example we will examin different function call methods and hidden problems. The sample code provided has warnings, do not fix them. We will be compiling our code with different optmization levels (-O0 and -O1). You will be documenting your comments in exercise.txt, be sure to clearly indicate what step you are commenting on. You will want to test each stage of the exercise before redirecting the output into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

1) Implement the 4 “complex add” functions in the provided code **lab_c.c**, and compile them with **-O0**, ignore any “function returns address of local variable” warnings.
2) Describe in your **exercise.txt** file the advantages or disadvantages of each addition method.
3) Run lab_c and redirect the output to exercise.txt, are all the results correct?
4) Recompile the code with **-O1** and run again redirecting the output to exercise.txt, are the results correct? Are they identical to the **-O0** run? Which function is not reliable? Why? Document your answers in **exercise.txt**.

Show your **exercise.txt** file to your TA for grading.

Note: Running lab_c on non-RIT Linux clusters MAY result in crash. Newer compilers will NOT ALLOW returning pointers to local variables and will replace them with NULL.

Assignment – 60 pts

Objective

Implement a C module to calculate non-real roots. Please use header files file rootfinding.h, Timers.h and ClassErrors.h from your previous assignment.

Debugging and Timing the code

Debug your module functions and verify that there are no memory leaks. Try different optimization levels, (**gcc** option **-O0 .. -O3**) and select the one that gives the "best" performance result. Record timing of your algorithms and save in your results file.

Complex Numbers

All of the previous root finding methods only find "real" roots but polynomials may contain imaginary roots. Implement a quadratic solver called **quadTest.c** which can find real and/or imaginary roots for any real coefficient 2nd order polynomial.

Use the quadratic equation to find the roots:

```
        = -b  +-  sqrt(b**2-4ac)
                  ----------------------
                          2a
```

The quadTest program must use **getopt_long_only()** to parse command line parameters and should be designed such that it takes the following required inputs:

```
  quadTest   -a Anum   -b Bnum   -c Cnum          or
quadTest   -numa Anum   -numb Bnum   -numc Cnum      or any combination of the two
```

which represent to real coefficients of the 2nd order polynomial: f(x) = ax**2 + bx + c

Your command line parser must flag an error if "Anum" is zero or if any of the 3 required parameters are missing. Your program must print friendly error messages and a clear success message of the form:

```
/quadTest -a 1 -b 0 -c 1
        The roots of: 1x**2 +0x +1
          Root1: 0 + -1i
          Root2: 0 + 1i
```

Hints: You will need to use a square root function, the standard function sqrt() is for doubles, not complex values, you must use csqrt(). While the compiler supports native complex operations, printf() does not. You will need to use creal() and cimag() to print the roots.

Makefile

Write a make file with the following targets:

"all"	-should make **quadTest** with **timing enabled** and **quadTest_mem** with **timing disabled**
"tests"	- should run the quadradic solver test with all the following parameters quadTest -a 0 -b 0 -c 1 quadTest -a 1 -b 0 quadTest -a 1 -b 0 -c 1 and redirect stdout and stderr to the output file **out.txt**. Note: **BOTH** successful and error cases should be directed to out.txt. The make file should continue running.
"mem"	- Run valgrind for quadTest with good data, redirecting output to mem.txt.
help, clean	- should do the normal things

Results and Analysis

Write in the file **analysis.txt** an explanation of the results obtained. Did you learn about any short comings in the C99 implementation of complex numbers? Summarize and document the performance observed for each of the optimization levels. Did the optimization work as you would expect? As always, provide all your raw data and provide data to justify your answer.

Submit a tarball with all C program(s) together with your analysis and Makefiles.

Grading Criteria

1. (40 points) Correct program(s)/algorithms.
2. (2 points) Memory leaks
3. (10 points) Makefile
4. (8 points) Analysis.

Student Name: ____

Laboratory Grading Sheet

Lab 09 - Complex Numbers

Component	**Point Value**	**Points Earned**	**Comments and Signatures**
Pre-Lab: define variables	**5**		
Pre-Lab: oral quiz	**5**		
Interactive Exercises: IEEE format	**10**		
Interactive Exercises: Epsilon	**10**		
Interactive Exercises: Function design	**10**		
Total	**40**		

You must turn this signed sheet in at the end of lab to receive credit!