DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Homework Assignment No. 06

Basic Math Operations and Numerical Precision

submitted to:

Professor Joseph Picone
ECE 1111: Engineering Computation I
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

January 21, 2017

prepared by:

Damien Ortiz
Email: tut62308@temple.edu

ECE 1111: Engineering Computation I

Homework No. 6: Basic Math Operations and Numerical Precision

Goal: Demonstrate that we must be cognizant of numerical precision when programming. Electrical and computer engineers are expected to be able to write efficient code for embedded systems, even if you are working in a high-level language. We are expected to write code for systems that run perpetually. Roundoff errors can accumulate over time and cause these systems to fail after billions of operations. In this assignment, you will gain some experience with numerical precision issues.

Description: In this homework assignment, we are going to use a loop of the following form:

```
long i_end = 999;
for (long i = 0; i < i_end; i++) {
      fprintf(stdout, "the value of i is: %d\n", i);
}
```

You will also find the following command useful: cat filename.txt | more. Piping to "more" sends the output of your program to a program called "more" that lets you control the output. In addition to "more", there are commands called "less" and "tail". Experiment with these (e.g., "more filename", "less filename", "tail filename").

Place your files in the directory:

```
/data/courses/ece_1111/current/homework/hw_06/lastname_firstname
```

Use subdirectories p01 and p02 for the problems below. To make things easy, use our standard make file template that I demonstrated in class.

The tasks in this homework assignment are:

1. Declare an unsigned character, which is an 8-bit (1-byte) variable. Increment its value using the code below. Explain what happens and why there might be a problem.

   ```
   unsigned char c = 0;
   for (long i = 0; i < 99999; i++) {
     fprintf(stdout, "c = %c (%d)\n", c, (long)c);
     c++;
   }
   ```

   Repeat this for an unsigned short int, an unsigned int and an unsigned long.

   Change this loop to iterate from -99999 to 99999. Repeat the above for signed char, an unsigned short int, a signed short int, an int and a long. Explain what you are observing.

2. Declare a floating-point value for the math constant pi:

   ```
   float my_pi = M_PI;
   ```

   Construct a loop that sums the square of M_PI 99,999 times. Divide the sum by 99,999 and print the difference between value computed and the theoretical value (M_PI * M_PI). Use a format of %15.10. What do you observe? How does the result change if you decrease 99,999 to 999, or increase it to 9 million?

   Repeat this for a double instead of a float. Does the result change? Why? Explain.

In addition to your code, submit the solutions to these tasks as a pdf document using a filename of *lastname_firstname_hw06.pdf* following the homework template provided. Place this in the parent directory. Provide explanations for what you observed when running your programs. Comment on the differences between integer and floating-point arithmetic.

A. BRIEF DESCRIPTION OF YOUR CODE

The goal of this assignment was to demonstrate the limitations of data types in the C programming language. I use multiple for loops printing the character representation and long representation of each variable.

For example, when using an unsigned char, we can observe the limitations of character representation. When the value is between 0 and 255, the program prints the corresponding ASCII character. Once the value exceeds 255, the variable wraps around to 0 due to overflow, demonstrating the fixed size of this data type.

```
unsigned char c = 0;

for (long i = 0; i < 99999; i++) {
  fprintf(stdout, "c = %c (%ld)\n", c, (long)c);
  c++;
}
```

In the second part of the assignment, we were using float values to perform math operations. Through this we can see the errors of computational rounding.

```
float my_pi = M_PI;
float sum = 0;
int loopCount =  99999;

for (int i ; i < loopCount; i++){

  sum += M_PI*M_PI;

}

float ans1 = sum/loopCount;

fprintf(stdout, "%f\n", ans1);
fprintf(stdout, "%f\n", my_pi*my_pi);
```

B. DEMONSTRATION THAT YOUR CODE WORKS

Part 1

Test Case 1. Input: Unsigned char, loop from 0 to 99,999
Output: Prints ASCII characters and integers from 0 to 255. After 255, the values went around to 0, demonstrating 8-bit unsigned overflow.
Test Case 2. Input: Unsigned short int, loop from 0 to 99,999
Output: Prints integers from 0 to 65,535. After 65,535, the value wraps around 0, demonstrating 16-bit unsigned overflow.
Test Case 3. Input: Unsigned int, loop from 0 to 99,999
Output: Prints integers sequentially from 0 to 99,999 without overflow
Test Case 4. Input: Unsigned long, loop from 0 to 99,999
Output: Prints integers from 0 to 99,999 without overflow
Test Case 5. Input: Signed char, loop from 0 to 99,999
Output: Prints integers from 0 to 127, then wraps around to -128, continuing the sequence, showing 8-bit signed overflow.
Test Case 6. Input: Signed char, loop from -99,999 to 99,999
Output: Prints integers from -128 to 127 repeatedly due to overflow, demonstrating behavior of 8-bit signed integers in both negative and positive ranges.
Test Case 7. Input: Signed short int, loop from -99,999 to 99,999
Output: Prints integers from -32,768 to 32,767 repeatedly, demonstrating 16-bit signed overflow.
Test Case 8. Input: Long, loop from -99,999 to 99,999

Output: Prints integers sequentially without overflow (64-bit long can handle this range).

Part 2

Test Case 1. Input: float representation of pi, double representation of pi

```
Output:
The percentage difference is    0.0461656339
The percentage difference is    0.0000000001
```

C. SUMMARY

Through this assignment I learned the limitations of each data type. Integers, chars, longs, and other data types can only represent a set number of bytes. When a value that surpasses the maximum number of bytes a data type can represent is used, the variable goes back to zero as it has been overflown. Understanding this is vital as we want to efficiently use correct data types that will be able to handle the values, we assign them while also making sure we do not allocate more memory than necessary

D. APPENDIX

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>


// main file

//

int main(int argc, char** argv){


  // create unsigned char increasing integer value from 0 to 99999

  // printing the char and long representation

  //

  unsigned char c = 0;


  for (long i = 0; i < 99999; i++) {

    fprintf(stdout, "c = %c (%ld)\n", c, (long)c);

    c++;

  }


  // create unsigned short integer increasing value from 0 to 99999

  // printing the char and long representation

  //

  unsigned short int integer = 0;


  for (long i = 0; i < 99999; i++) {

    fprintf(stdout, "int = %c (%ld)\n", integer, (long)integer);

    integer++;

  }

```

```
  // create unsigned integer increasing value from 0 to 99999
```

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>


int main(int argc, char** argv){


  // create float version of pi

  //

  float my_pi = M_PI;


  // add pi squared a preset amount of times then divide it by the number of times we looped

  //

  float sum = 0;

  int loopCount =  99999;


  for (int i ; i < loopCount; i++){


    sum += M_PI*M_PI;


  }


  float ans1 = sum/loopCount;


  // calculate and prin the percentage difference between the calculated and theoretical values

  //

  float percentageDiff1 = abs((ans1 - M_PI*M_PI)/(M_PI*M_PI)*100);


  fprintf(stdout, "The percentage difference is %15.10f\n", percentageDiff1);
```