DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Homework Assignment No. 03:

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

September 12, 2022

prepared by:

Leo Berman
Email: leo.berman@temple.edu

A. ASSIGNMENT

ECE 1111: Engineering Computation I

Homework No. 3: 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_03/lastname_firstname

Use subdirectories p01 and p02 for the problems below. To make things easy, use our standard Makefile 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_hw03.pdf following the homework template provided. Place this in the parent directory.

B. PURPOSE/BRIEF DESCRIPTION OF YOUR CODE

Problem 1 : The code for figures 1 – 10 assign a variable type to the variable c. What this does is set the amount of information that the variable c can communicate to the fprintf command. For example, in figure 1a it shows that the number resets at 255 because an unsigned char can only carry 1 byte which is 8 bits of information which means that it can carry 256 combinations or 255 numbers because the first combination stands for 0. This is a problem because there are very clearly 99999 different combinations that the variable has to hold so when it runs out of combinations it just resets. This can be seen in figure 1 where I show the reset point which happens at 255. However when you change the variable to something that can carry more information, for example in fig 2a an unsigned short int can carry 65535 different combinations then it can show 65535 pieces of information. However, since we were still trying to print more combinations than shown it will eventually reset which is shown in fig 2a. For fig 10a for a long integer, it can carry more than the 199998 combinations so it never has to reset so you can see that it makes this in Fig 10a because the last number on the right for unique character is 199997 (accounting for that zero at the beginning).

Problem 2 : Seen in Figure 11, my code showcases the rounding errors that accumulate due to different information types being store. Since PI isn't a real number, technically the computer can't possibly register it to 100% accuracy so it juts sets it to a certain amount of decimals for what it thinks is enough accuracy. What happens is that when the computer uses this value of pi so many times it strays farther and farther from the original value because some of the precision is loss every time it processes a piece of information outside of the scope of the variable. As can be seen under my Proof That My Code Works section, the output of my code is some extremely small decimal even though if we manually did the calculations, it would be zero. This happens because of those rounding errors that happen when the code is performing calculations outside the variables scope of information.

C. PROOF THAT MY CODE WORKS

Input : ece-000_[1]: p02.exe

Output : 0.000000663302728

D. SUMMARY

The purpose of these problems was to demonstrate the significant of the amount of information a single type can hold. For problem 1 the amounts of information held within the types determined how many unique characters could be generated for the lists. Essentially, this means that the types that can hold enough information in regard to unique combinations (~200,000 combinations) would be able to generate the entire list of unique characters for the loop. If they were a type that couldn't hold enough unique combinations the list would reset. This is important because when we are trying to code many different situations we need to make sure we use data types that will suit our needs for unique combinations. It functions similarly to our language if we need a million words, we couldn't assemble them with only 3 letters.

For the second problem, we were meant to develop an understanding of how rounding errors can be so small but affect the outcome. By using an irrational number such as pi we can observe how the lack of infinite accuracy for a variable can cause a rounding error because use something that goes past the decimal points and as we use it more there is even more significant information lost. The more we reuse the variable the farther it strays from the original value to the point where you can start having significant errors. The example given in class was with banking and how those fractions of a penny used to be adjusted for in less sophisticated way to the point where banks were changing accounts by a penny each month and then explaining the errors to their patrons.

E. APPENDIX

Fig 1 – unsigned char 99999

```
#include "p01a.h"
int main(){
unsigned char c = 0;
 for (long i = 0; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 2 - unsigned short int 99999

```
#include "p01a.h"
int main(){
unsigned short int c = 0;
 for (long i = 0; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 3 – int 0 > 99999

```
#include "p01a.h"
int main(){
int c = 0;
 for (long i = 0; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 4 – unsigned long 0 > 99999

```
#include "p01a.h"
int main(){
unsigned long c = 0;
 for (long i = 0; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 5 – unsigned char -99999 > 99999

```
#include "p01a.h"
int main(){
unsigned char c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 6 - signed char -99999 > 99999

```
#include "p01a.h"
int main(){
signed char c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 7 – unsigned short int -99999 > 99999

```
#include "p01a.h"
int main(){
unsigned short int c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 8 – signed short int -99999 > 99999

```
#include "p01a.h"
int main(){
signed short int c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 9 – int -99999 > 99999

```
#include "p01a.h"
int main(){
int c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 10 – long int -99999 > 99999

```
#include "p01a.h"
int main(){
long int c = 0;
 for (long i = -99999; i < 99999; i++) {
   fprintf(stdout, "c = %c (%d)\n", c, (long)c);
   c++;
 }
}
```

Fig 1a – unsigned char 99999

c = ü (252)

c = ý (253)

c = þ (254)

c = ÿ (255)

c = (0)

c = (1)

c = (2)

c = ——————————— (3)

Fig 2a – unsigned short int 99999

c = ü (65532)

c = ý (65533)

c = þ (65534)

c = ÿ (65535)

c = (0)

c = (1)

c = (2)

c = ——————————— (3)

Fig 3a – int 0 > 99999

c = — (99991)

c = ˜ (99992)

c = ™ (99993)

c = š (99994)

c = › (99995)

c = œ (99996)

c = (99997)

c = ž (99998)

Fig 4a – unsigned long 0 > 99999

c = — (99991)

c = ˜ (99992)

c = ™ (99993)

c = š (99994)

c = › (99995)

c = œ (99996)

c = (99997)

c = ž (99998)

Fig 5a – unsigned char -99999 > 99999 *c = ü (252)* *c = ý (253)* *c = þ (254)* *c = ÿ (255)* *c = (0)* *c = (1)* *c = (2)* *c = ——————————— (3)*	*Fig 6a - signed char -99999 > 99999* *c = ü (-4)* *c = ý (-3)* *c = þ (-2)* *c = ÿ (-1)* *c = (0)* *c = (1)* *c = (2)* *c = ——————————— (3)*
Fig 7a – unsigned short int -99999 > 99999 *c = ü (65532)* *c = ý (65533)* *c = þ (65534)* *c = ÿ (65535)* *c = (0)* *c = (1)* *c = (2)* *c = ——————————— (3)*	*Fig 8a – signed short int -99999 > 99999* *c = ü (-4)* *c = ý (-3)* *c = þ (-2)* *c = ÿ (-1)* *c = (0)* *c = (1)* *c = (2)* *c = ——————————— (3)*

Fig 9a – int -99999 > 99999	*Fig 10a – long int -99999 > 99999*
c = 6 (199990)	*c = 6 (199990)*
c = 7 (199991)	*c = 7 (199991)*
c = 8 (199992)	*c = 8 (199992)*
c = 9 (199993)	*c = 9 (199993)*
c = : (199994)	*c = : (199994)*
c = ; (199995)	*c = ; (199995)*
c = < (199996)	*c = < (199996)*
c = = (199997)	*c = = (199997)*

Fig 11 –

```
// includes libraries in header file
//
#include"p02.h"

// main entry point of program
//
int main (){

  // sets variable my_pi to float variable M_PI which is the preloaded vale of pi
  //
  float my_pi=M_PI;

  // sets the value of my_pi squared * 99999
  //
  my_pi=pow(my_pi,2)*99999;

  // set the value of my_pi and divides by 99999
  //
  my_pi=my_pi/99999;

  //
  //
  float diff= my_pi-pow(M_PI,2);

  //prints the value difference
  //
fprintf(stdout, "%10.15f\n",diff);

//exits gracefully
//
 return 0;
  }
```