Lab 12 Report

Sage Salmon and Jordan Reese

April 15 2018

Introduction

The main task in this lab was running a simulation of two characters who needed to pass different challenges in order to move around a room. When run, the code chose several of the challenges when run and if passed, the characters would be allowed to move in the specified direction. If the challenges did not pass the characters would not move.

Challenges

There was a list of challenges that we were required to write code so each challenge would pass. These challenges are listed below:

- **Calculation**: Given a string of a mathematical expression, return the integer value
- **Sub-String**: Given two strings, return a pointer to the beginning of the first occurrence of the specified string
- **Range**: Given two integers, return the range between them as an integer
- **Mean**: Given an array of floats and the number of floats in the array, return the arithmetic mean as a float
- **Minimum**: Given an array of floats and the number of floats in the array, return the minimum value as a float
- **Maximum**: Given an array of floats and the number of floats in the array, return the maximum value as a float
- **Reverse**: Given a string, return the string in reverse
- **Find**: Given a string and a char, return the integer index of the position of the first occurrence of the char
- **Tokenize**: Given a string, tokenize it by whitespace and return a pointer to a malloced 1-D array of the token strings.

Whether or not these challenges passed when called was the determining factor in how the characters moved. Many of these challenges were not too complicated to understand or to code. Several of them were a means of doing simple mathematical operations and just needed the syntax done properly. On the other hand, a few of the challenges, such as **Calculation** and **Tokenize**, were more challenging and required more work to figure out.

The next hardest part of the challenges was getting all of the answers passed to the proper location so the simulator could verify our answer. The simulator gave us a void pointer that we needed to cast our answer to. There were a couple ways you could do this. You could directly cast the answer using some pointer notation, or you can do what we did and make a union. Our union contained all of the data types needed for the challenge answers. We pointed the void pointer at the union and then never had to worry about the void pointer again. We only had to make sure the data type our function returned matched the one needed in the union.

Handling the challenges

We created a function that passes all of the returns values from the challenges function to the void pointer and makes sure the simulator gets the correct answers. Below is the handle function, function. It is just a series of else if statement that check to see with function was called. This function makes it easy to determine where an error would be if the simulator doesn't receive our answer.

```
  printf("\n\n");
  union Challenge_Answer *ans = round.ans_ptr;
  if (round.chal_typ == calc){
    // printf("Did I get here?\n");
    ans->i = Calc(round.chal_dat.calc);
    // *(int *)round.ans_ptr = Calc(round.chal_dat.calc);
    //ans->i = 5;
  }
  else if (round.chal_typ == substr){
    ans->str = Sub_Str(round.chal_dat.substr.s1,
round.chal_dat.substr.s2);
  }
  else if (round.chal_typ == range){
    ans->i = Range(round.chal_dat.range.a, round.chal_dat.range.b);
  }
  else if (round.chal_typ == mean){
    ans->f = Mean(round.chal_dat.minmaxmean.nums,
round.chal_dat.minmaxmean.datalen);
  }
  else if (round.chal_typ == min){
    ans->f = Min(round.chal_dat.minmaxmean.nums,
round.chal_dat.minmaxmean.datalen);
```

```
  }
  else if (round.chal_typ == max){
    ans->f = Max(round.chal_dat.minmaxmean.nums,
round.chal_dat.minmaxmean.datalen);
  }
  else if (round.chal_typ == rev){
    Reverse(round.chal_dat.rev, ans->arr);
  }
  else if (round.chal_typ == find){
    ans->i = Find(round.chal_dat.find.s1, round.chal_dat.find.key);
  }
  else if (round.chal_typ == token){
   ans->str_array = Token(round.chal_dat.token);
}
  return 0;
}
```

Movement

The movement was fairly simple when it came to what to code. There was a vision matrix with ones and zeros which distinguished yes from no and there were cardinal directions given with the rows. The code we wrote was four if statements that checked to see which movement direction passed and then moved the character in that direction. The movement code is below.

```
Round round){
  int i = 0;
  int j = 0;
  for(i = 0; i < 3; i++){
    for(j = 0; j < 3; j++){
      printf("%d ", round.vision[i][j]);

    }
    printf("\n");
  }
  if(round.vision[0][1] != true){
    return 1;
  }
  else if(round.vision[1][2] != true){
    return 2;
  }
  else if(round.vision[2][1] != true){
    return 3;
  }
  else if(round.vision[1][0] != true){
    return 4;
  }
```

```
  return 0;
}
```

This is not even close to the most efficient way to move around the map, but it does guarantee that the character always moves unless it is completely trapped. This algorithm does have issues when the character starts in a corner. We didn't spend too much time on the movement because in our minds it wasn't the main focus of the lab.

Issues

There were a lot of problems that we ran across when working through the lab requirements. The biggest issue we kept running into was a problem with the lab given code. We were working through the challenges and writing additional code to check to make sure the code returned the correct values, however, the challenges still failed in the simulation for most of the challenges. This issue was not resolved for several days, which gave us grief when we were wanting to move forward with the lab instructions. We also had issues interpreting some of the lab manual instructions. Some of the requirements were very unclear and worded poorly. There needs to be more elaborated on some of the challenges and how exactly they are supposed to be done or what they are looking for.

The gnuplot was probably the worst part of this lab. Using the program had a really high learning curve, and we still couldn't figure out how to properly print out a heat map. The map as it sits will roughly show where the ones are located. The gradient prevents any super solid colors from popping out. But we did figure out how to uses the gnuplot inside the c code. Our code pulls a script from the directory to make life a little easier.

Another problem we had was writing the tokenize function. The built in C function strtok that can tokenize has a problem. It changes the string passed into it. We had to make several copies of the original string so we didn't alter the pointer.

Complexity

This code is not very complex. The code varies in complexity based on the challenges passed to it. If we are given some of the easier functions our Big O is O(n) and our complexity goes up to $O(n)^2$. We have our loops buggin out if there are error, and if any of the files don't get opened properly. Based on what the simulator sends us then we should be able to compensate for an errors.

Conclusion

This lab seemed to be a fair amount of work but within the ability of students in this course. There were a lot of functions, but most of them were very simple

and did not take much time or effort when creating the code. After getting clarification and understanding what the lab manual wanted, the coding was not overly complicated. The hardest part of this lab was trying to work through it while the simulation was not working. We had to delay some of our work because we couldn't debug without knowing if our challenges passed or not. All in all, the exercises in this lab exercised the skills we have been learning throughout the semester and showed a physical impact that our code had on a simulation. We were able to see changes in something that didn't just involve lines of code or numbers on a screen, which made it easier to realize the impacts knowing good coding practice can make.