Midterm 1 (retake): C Programming

Laurent Michel

October 18, 2021

Grading Structure

The midterm has 4 questions (Q1 to Q4). The grade break down is given as follows:

Questions	Points
Q1	25
Q2	20
Q3.1	25
Q3.2	20
Total	90

Note that *each question* is covered by many grading scripts and that you can earn partial credit. Some are easier to pass then others.

I want to emphasize once again that the *solutions are short.* If you find yourself writing a substantial amount of code, you are most likely making this too hard on yourself.

Q1. Frequency of Leading Digits.

Complete the code in *freq.c* to obtain the frequencies of the first (leading) digit of n positive long integers (when written in decimal representation). For example, if we have the following positive long integers in an array,

123 20 34 455 5 6778 700 8 90 900 999

the frequencies of the leading digits are:

0:0 1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:3

In this example, since three numbers, 90, 900, and 999, start with digit 9, the frequency for digit 9 is 3. The frequency of other non-zero digits is 1.

Specifically, you need to write the following functions in this problem. The first function is

```
int getMSDigit(int x);
```

This function takes x as input and returns the first digit of the number x.

The second function you need to write is the

int main();

The main() function reads integers from the stdin till the user inputs EOF (Ctrl-D). For each value read, it obtains the most significant digit and stores frequencies of digits in array freq[], which has 10 elements, one for each decimal digit. freq[0] stores the frequency of 0, freq[1] stores the frequency of 1, and so on. It then prints these frequencies on the stdout.

Below are some sample sessions running the program.

$./freq
0 1 2 3 4 5 6 7 8 9 Ctrl-D
0:1 1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1

$./freq
100 200 300 400 500 600 800 900 999 1000 2000 3000 Ctrl-D
0:0 1:2 2:2 3:2 4:1 5:1 6:1 7:0 8:1 9:2

Q2. Polynomial hash of a string

Complete the code in *poly.c* to implement string hashing. String hashing is a way to convert a string into an integer known as a hash of that string. An ideal hashing is the one in which there are minimum chances of collision (i.e 2 different strings having the same hash). You need to calculate the hash of a string $s_0s_1...s_{n-1}$ as below

$$\text{hash} = \sum_{i=0}^{n-1} s_i \times 26^{n-1-i}$$

where s_i represents the ASCII value of the character s_i.

The code consists of three functions: readString(), hashString() and main(). We have implemented the readString() function that reads a '\n' terminated string of arbitrary length from the stdin, and the main() function that calls the readString() function, calls the hashString() function and prints the hash value. Specifically, you need to write the following function:

unsigned long hashString(char* s);

This function takes as input a string, computes its hash using the above equation and returns it. Note that you may not use the built-in *pow* function in the math library. (Hint: avoid calculating exponents to speed up the computation!) Below are some sample sessions running the program.

$./poly
Darth
Hash of [Darth] is 32859424

```
$./poly
may the force be with you
Hash of [may the force be with you] is 16943079911126818559
```

Q3. Reverse Polish Notation (RPN) calculator

Complete the code in *token.c* and *rpn.c* to implement an RPN calculator. RPN is a method for representing expressions. In this method, operators are placed after the operands they operate on. One of the advantages of RPN is that there is no need for parentheses to change precedence. For example, we all are familiar with expressions like this:

(1 + 2) * 5

RPN can express the same operations, without parentheses, as follows.

1 2 + 5 *

In this example, +, placed after 1 and 2, operates on 1 and 2. * operates on the result of the addition and 5.

RPN expressions can be easily evaluated with the help of a stack. The expressions are divided into tokens, which can be a number, an operator, or even a command. And the tokens are processed as follows.

- If the token is a number, push it on stack.
- If the token is an operator, pop operands from the stack, perform the operation, and push the result onto the stack. All the operations you need to implement need 2 operands.
- If the token is a command, complete the necessary action (for e.g. ‘p’ should pop the element on the top of the stack and prints it).

You are given the implementation of a stack. The struct IStack is defined in *stack.h* and consists of a pointer to an integer array (t), number of entries in a stack (sz), and maximum number of entries that the stack has space for (mx) as shown below.

```
typedef struct IStack {
  int mx;
  int sz;
  int* t;
} IStack;
```

The functions to (a) make a stack, (b) free a stack, (c) push an entry v in stack, (d) pop an entry from the top of a stack and (e) check if stack is empty are implemented for you in *stack.c*.

You are also given a struct token, which is defined in *token.h* and consists of a type and value of the token as shown below.

```
typedef struct Token {
  char type;  // literal, +, *, -, /, p, q
  int value; // value of the literal
} Token;
```

The token type can be a literal (integer); an operator to add (‘+’), multiply (‘*’), subtract (‘-’) or divide (‘/’); a command to print (‘p’) or quit (‘q’); or EOF. The value holds value of a literal.

You need to write the following functions for this problem.

Q3.1 Read token

The first function to be implemented is in *token.c*.

```
int readToken(Token* t);
```

This function takes as input a token and returns a Boolean (to indicate EOF). If the token is a whitespace, skip to the next. If the token is an operator, command or EOF, assign the type field. If the token is a literal, read all digits (if the value > 10) and assign the type and value fields.

Q3.2 Implement the RPN calculator

The second function you need to write is in *rpn.c*.

```
int main();
```

The main() function implements the RPN calculator. Read tokens from the stdin till the user inputs EOF or the quit command, and process the expressions as described above. Remember your code cannot have memory leaks, otherwise it will not pass the Valgrind test.

Below are some sample sessions running the program.

```
$./rpn
1 2 + 5 * p q
15

$./rpn
15 3 / 4 – p
1
Ctrl-D
```