Created by: Luiz Navarro, in collaboration with Jacky Chan, Raghavendran Thiruvengadam, and Christopher Lucasius

# ESC190 Lab3

RPN Calculator

**Due:** March 3rd, 2020 at 23:59

## Instructions

1. `git pull` to access the provided starter code. You will find the code under <`utorid`>`_esc190/lab3/`
2. **Only** the `stack.c` and `calc.c` files will be marked. You may modify `main.c` to test your code.
3. Your code must compile and run on the ECF machines with `gcc` version 8.3.1.
4. An additional requirement for this lab is that your program runs clean using `valgrind`. After the executable is generated and you verify that your program functions as expected, you should run

   ```
   valgrind --leak-check=full ./calc sample_in.txt output.txt
   ```

   Ensure that you do not have any memory leaks. A small portion of the grade (20%) is reserved for this.
5. To submit: stage, commit and push your code.

## Motivation

Due to increasing tuition costs, you are resorting to cost-saving measures such as maximizing your calculator battery life (thus spending less money on batteries). With some tinkering you find that your calculator is programmed in C, and you set out to modify the calculator's operation such that it takes the fewest number of button presses to conserve the calculator's battery. After researching online, you discover that inputting operations in Reverse Polish Notation (RPN) will accomplish this. You are excited to learn that you can use a stack to enable your calculator to work with RPN, finally giving you a chance to apply the knowledge you learned from ESC190. With limited memory on the calculator, there is zero tolerance for memory leaks. If, after a computation, you do not free the memory of your calculator, it will crash your program and potentially kill your calculator. You must be careful to avoid any leaks as you are handling your stack.

## Introduction

*Reverse polish notation* (RPN) also known as *postfix* notation is a type of mathematical notation where the operators are placed **after** their operands as opposed to **between** them (*infix*). Instead of writing $1 + 1$ as we are used to, we write $1, 1+$. The main benefit of this notation is that it does not require the use of parentheses to determine the precedence of operations. For example, an expression like

$$(1 + 2) \times 3 + 4$$

Can be written without any ambiguity as

$$1\ 2 + 3 \times 4+$$

As a result, calculators that use *postfix* notation require less keystrokes and less memory (no need to store the entire expression and parenthesis). Because of this, *postfix* notation was commonly used in the early scientific calculators and has persisted to this day in some calculators such as the HP-12C and HP-50g.

## Provided files

- `stack.c` and `calc.c`
  - These are the **only** files that will be marked
- `stack.h` and `calc.h`
  - **Do not modify these files**.
  - These are header files that provide an interface between your c-files and the files used for running and testing your code, they declare the prototypes of all the functions you have to implement.
- `main.c`
  - A sample file that puts all of your code together to test the calculator.
  - You **may** modify this file for testing but it will **not** be graded.
- `Makefile`
  - **Do not modify this file.**
  - A makefile to be used by the GNU make tool to build your code.
- `sample_in.txt` and `sample_out.txt`
  - Example input and output files for testing your code.
  - When the test code is given the `sample_in.txt` file as an input, its output should match what is given in `sample_out.txt`. This is only a sample, you should also create your own input files to test if your program behaves as expected.

## Building and running your code

To build your code, open a terminal, switch to the <utorid>_esc190/lab3/ folder and type

```
(<utorid>_esc190/lab3/)$ make
```

If your code builds with no errors an executable file called `calc` should be created. If you run it with no arguments you should see the following:

```
(<utorid>_esc190/lab3/)$ ./calc
Missing input and output file names. Usage:
./calc input output
```

To use the provided sample input file and save the result to a file called `output.txt` type

```
(<utorid>_esc190/lab3/)$ ./calc sample_in.txt output.txt
```

To compare your obtained output to the given sample output use:

```
(<utorid>_esc190/lab3/)$ diff sample_out.txt output.txt
```

## Tasks

Implement the following functions in the provided `.c` files:

- `struct stack* create_stack(void);`
  - This function creates and initializes an empty stack. An empty stack must have size 0 and have its top set to `NULL`.
  - **Return value**: The function should return a pointer to the created stack. If it was not possible to allocate memory for the struct, this function should return `NULL`.
  - **Hint**:
    Try running the following code:

```c
#include <stdio.h>
#include <stdlib.h>

int* create_a(void){
    int i = 10;
    return &i;
}
int* create_b(void){
    int *i = malloc(sizeof(int));
    *i=10;
    return i;
}
int main(void){
    int *a = create_a();
    int *b = create_b();

    printf("a=%d\tb=%d\n", *a, *b);
}
```

    These 2 functions try to do the same thing but behave differently. Can you explain why? Think of the difference between automatic and dynamic memory.

- `void delete_stack(struct stack* s);`
  - **Input**: this function takes as an argument a pointer to a `struct stack` and deletes the stack, freeing **all** memory used by it.

  - If the **struct** `stack` contains any pointers to memory this function must call `free` on **all** of these pointers. After that, this function must also call `free` on the pointer `s`.

- **double** `pop`(**struct** `stack*` `s`);
  - **Input**:this function takes as an argument a pointer to a **struct** `stack` and pops the last item added to the stack, i.e., the top item of the stack is removed from the stack and returned by the function. This function should update the size of the stack appropriately.
  - **Return value**: this function returns the value popped from the top of the stack. If the stack is empty or `s` is `NULL` this function should return 0.

- **int** `push`(**struct** `stack*` `s`, **double** `e`);
  - **Input**: this function takes as an argument a pointer `s` to a **struct** `stack` and a **double** `e` and pushes `e` into the stack represented by `s`.
  - **Return value**: If the operation is successful the function should return 0. If it was not possible to add the element to the stack this function should return $-1$.

- **double** `compute_rpn`(**char*** `rpn`)
  - **Input**: a single line string containing an RPN expression that evaluates to a single final number
  - **Return value**: the number the RPN expression evaluates to.
  - **Description**: this is your core calculator function. Given a RPN expression this function must execute the operations specified by it and return the final result. There are 2 types of valid operators: binary operators, which require 2 operands, and unary operators, which require only 1 operand. The valid operators are:
    - Binary operators:
      - '+' : `op1` + `op2`
      - '−' : `op1` − `op2`
      - '*' : `op1` × `op2`
      - '/' : `op1`/`op2`
      - '^' : `op1` ∧ `op2`
      - 'f' : flip the order of the operands
    - Unary operators:
      - 's' : sin(`op1`)[1]
      - 'c' : cos(`op1`)
      - 't' : tan(`op1`)
      - 'e' : exp(`op1`)
      - 'i' : 1/`op1`
      - 'm' : −`op1`
      - 'r' : $\sqrt{\texttt{op1}}$

    The `calc.h` file defines the macros `unary_op(c)` and `binary_op(c)` to check if a given operator is a valid unary or binary operator respectively.
  - **Assumptions**:
    - Every RPN expression is given in a single line and contains only numbers and valid operators.

[1]The trigonometric operators assume an angle in radians

* Every number and operator is separated by a single whitespace (feed forward).
* The number zero is always given as "0", never ".0", "−0" etc.
* Every RPN expression will be less than 256 characters long.
* Every RPN expression will evaluate to a single number in the end (the stack won't have any leftover numbers).

– **Hints**:

* You can use the `atof()` function to convert a string into a double. This function takes a pointer to a null terminated string and converts it to a float, returning 0 if the string does not correspond to a float.
* You can use the `strtok()` function to extract tokens from strings. This function takes pointers to 2 strings, the string to be parsed and a delimiter. At each occurrence of the delimiter, this function breaks the string into separate tokens and for each successive call returns the next token.
* To learn more about these functions, check the example below. You can also check the manual by opening the terminal and typing

```
$ man atof
$ man strtok
```

* The following example converts the string "1;2;3;4" into the array [1,2,3,4] :

```c
#include <stdio.h>
#include <string.h>  //necessary for strtok
#include <stdlib.h> //necessary for atof

int main(void){
    // the string to be parsed
    char str[10]="1;2;3;4";

    // The first call to strtok should include str (the
    // string to be parsed). strtok returns a pointer to
    // a string containing the first token
    char *tok=strtok(str,";");

    // The array to store the parsed numbers and the index
    // to track how many items have been added
    int i=0, array[4];

    // if the pointer returned by strtok is NULL then there
    // are no more tokens to be parsed in the string
    while(tok){ //same as tok==NULL
        // convert token to int and print it
        array[i]=atoi(tok);
        printf("%d ", array[i]);

        // subsequent calls to strtok on the same string
        // should pass NULL as the first argument.
        // strtok maintains an internal copy to the
        // string  passed on the first call and will reset
        // it if str is given  again
```

```
            tok=strtok(NULL, ";");
            i++;
        }
    }
```

- **Examples**:
  - `compute_rpn("1 1 +")`=2
  - `compute_rpn("1 1 + i")`=0.5
  - `compute_rpn("1 1 + 2 ^")`=4
  - `compute_rpn("1 1 2 + + r")`=2
  - `compute_rpn("2 3 ^")`=8
  - `compute_rpn("2 m")`=−2
  - `compute_rpn("1 2 m +")`=−1
  - `compute_rpn("1 2 f m +")`=1
  - `compute_rpn("1 2 /")`=0.5
  - `compute_rpn("1 2 f /")`=2
  - `compute_rpn("0 e")`=1
  - `compute_rpn("1 e")`=2.7182
  - `compute_rpn("3.1415 2 / s")`=1
  - `compute_rpn("3 3.1415 2 / * c")`=−0.00013
  - `compute_rpn("7.2")`=7.2

- `char* get_expressions(char* filename)`
  - **Input**: name of the input file
  - **Return value**: contents of the input file in a character array
  - **Description**: this function should open the file given by `filename` and use functions such as `fscanf()` or `fgets()` to read the contents of the file and store them in a character array.
  - **Hints**:
    - Depending on which function you use to read the file and how you use them it may be necessary to manually add the newline characters (`'\n'`) back into the character array to separate the RPN expressions.
    - Similar to when you were creating the stack, you should be careful with how your allocate you character array. The following example illustrates a case when 2 functions try to do the same thing but achieve very different results:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* hello_a(void){
    char *hello = malloc(20*sizeof(char));
    strcpy(hello, "Hello World!");
    return hello;
}

char* hello_b(void){
```

```
    char hello[20] = "Hello World!";
    return hello;
}


int main(void){
    char *a = hello_a();
    char *b = hello_b();

    printf("a=%s\nb=%s\n", a, b);
}
```

- **void** `process_expressions`(**char*** `expressions`, **char*** `filename`)
  - **Input**: a character array containing multiple RPN expressions separated by a newline character and the name of the output file.
  - **Description**: this function takes the output of the `get_expressions`() function, runs each line through the `compute_rpn`() function and saves the result to the output file with name given by the second argument `filename`. Each result is printed in a separate line. See the sample output file for an example of the output.
  - **Hint**: you can use the `strtok`() function to separate the rpn expressions, but beware that you may need to use `strtok_r`() instead. Remember that `strtok`() keeps an internal copy of the string it receives when the first argument is not NULL. If you call a function that also uses `strtok`() in between your calls the copy will be overwritten. `strtok_r`() allows you to provide a buffer to hold this copy between calls, bypassing this problem. To learn more about the difference you can go into a terminal and type

    ```
    $ man strtok
    ```

## Submission Instructions

Commit your code to your local repository and when you are ready, push to the remote.

Use `git add`, `git commit`, and `git push` for this. Don't be afraid to push incomplete versions of your code, we will only grade your latest version (from before the deadline). Make sure all your work is in the `stack.c` and `calc.c` files, as we will not grade or use any other files you submit.

You do not need need to submit object and executable files, we will not use them. Do not worry about deleting them when committing and pushing your code, the provided `.gitignore` makes sure these files are not tracked (uploaded). If you do want to delete the object files and executables, just run `make clean` in the appropriate directories.