Problem Set 2: Maze and Command Line Calculator

Please send back to me via NYU Brightspace

- A zip archive named as
  `PS02_Last_First.zip`
  Where `Last` is your last name, `First` is your first name, and the archive contains **`maze_recursion.c`** and **`cmd_line_calc.c`** files.

Look up how to use any of the functions you might want to use at https://www.cplusplus.com/. Look at example code to determine what `#include <>'s` are needed in addition to `#include <stdio.h>`.

Total points: 100

Download the file PS02.zip from NYU Classes.

Problem 1
Solve Maze using Recursion
50 Points

For You are given the files **maze.c, maze.h** and **maze.txt**. These files are complete and do not require any additional code. It is your task to create

- **maze_recursion.c**

which contains function **solveMaze()** that recursively solves the maze.

You can use the bash script **`./build_maze.sh`** as an easy way to compile your program.

The structure of **maze_recursion.c** should be

```
#include <stdbool.h>
#include "maze.h"

bool solveMaze(int i, int j) {

     (your code here)

}
```

You must write the function **solveMaze(i, j),** is such a way that it can call itself recursively to solve the maze.

The maze is file **maze.txt,** which is read into the global character array grid[i][j]. This is done in **maze.c**. You position is (i, j) where i (row) and j (column) are declared in **main()**

and are the array indices in grid[i][j]. Variable i is the row index, or North/South, and variable j is the column index or East/West

To start (that is, on the first call to your function):

- The position (i, j) is at 'S' in the maze.

In the following description of the tests in function solveMaze(), you should test against the *symbolic values* #define'd in maze.h, e.g. END_MARKER and not 'G'.

On each call to **solveMaze():**

- If the character at the current maze position grid[i][j] is the END_MARKER ('G') then you found the goal and you are done, so return **true**;
- If the character at the current maze position grid[i][j] is the VISITED_MARKER ('.') then you have already been at this position and do not want to back track, so return **false**.
- If the character at or position of grid[i][j] is "illegal" then you cannot go to this position, so return **false**.
  Illegal characters are:
    - A Wall, or WALL_MARKER ('|')

  Illegal positions are:
    - Anywhere outside of the maze. This would be an invalid grid[i][j] array index (i.e. less than 0 or greater than DIM_I-1 or DIM_J-1).
- The three tests above should be the three initial test statements in your solveMaze() function.

If you haven't encountered any of the above conditions, then

- Drop a breadcrumb, i.e. set the current position maze grid[i][j] character to VISITED_MARKER ('.') to indicate that you have visited this position.
- Display maze grid by calling display()

Next

- Move one grid step in the N direction and call **solveMaze()** -- this is the recursion. If this returns **true**, then you found the goal, and so then:
    - Set current position maze grid[i][j] character to the SOLUTION_MARKER ('*'). This is the "backtrace" path that is the implicit solution provided by the recursion process.
    - Display maze grid by calling display()
    - Return **true**

Otherwise

- Repeat the previous step (i.e. "Move one grid step …") for the other points of the compass (S, E, W).
- If none of the calls to solveMaze() (N , S, E, W) returned true, then return **false**.

Note that the directions are:

North i-1, j unchanged

South	i+1, j unchanged
East	j+1, i unchanged
West	j-1, i unchanged

The description above results in a series of `if() else if() else` statements:

```
if ( solveMaze(i-1, j) ) { //North
     (statements if true)
}

…    //code for South, East

else if ( solveMaze(I, j+1) ) { //West
     (statements if true)
}
else {
     return false;
}
```

Note that it the "statements if true" shown above are largely the same, they could be realized as a call to a function.

An alternate way to code the algorithm is to note that the "early exit" properties of an `if()` statement having a compound relational expression permits the sequence of four tests discussed above. The expressions in the `if()` statements would be:

```
if (solveMaze(i-1, j) || solveMaze(i+1, j) ||
        solveMaze(i, j-1) || solveMaze(i, j+1)) {

     (statements if true)

    }
    else {
        return false;
    }
```

Any of these ways is correct.

Problem 2
Command Line Calculator
50 Points

Create a program **`cmd_line_calc.c`** that implements a simple 4-function calculator. Your program must have command-line arguments as follows:

```
./clc x1 operator x2
```

Where x1 and x2 are float values and operator is one of +, -, x, or /.

Your program executable must be named **`clc`**, and you can use the bash script **`./build_clc.sh`** as an easy way to make this happen.

Your program should

- Check that there are exactly 4 command line arguments.
- If not, it should print these usage lines:

```
Usage: ./clc x1 operator x2
where x1, x2 are floats and operators are + - x /
```

Otherwise, parse the command line:

- Convert the second and forth command line arguments to float variables. HINT: use atof() to convert from the command line argument (which is a string) to a fload. Look it up at https://www.cplusplus.com/ to check if it needs any #include <>'s.
- Convert the third command line argument to a single character. HINT: the command line argument is a NULL-terminated string of length 1 character. You only want the first character, and not the NULL.

Use the Command Line Arguments to apply the desired calculation

- Use a switch statement or a series of if/else if/ else statements to apply the operator to the two floats and save the result in a float y.
- Print an error if the operator is not one of `+ - x /`
- Print an error is the command line would result in a divide by zero
- If no error, print the result using something like this:

```
printf("%f %c %f is %f\n", x1, op, x2, y);
```

Note: we are using 'x' as the multiply operator instead of '*' so we can all use the test script below.

Test your program by executing the bash script **`./test_clc.sh`**. If you are using a Windows platform, use **`./test_clc_win.sh`**.
It should produce this output:

```
Usage: ./clc x1 x2 operator
where x1, x2 are floats and operators are + - x /
2.500000 + 3.500000 is 6.000000
2.500000 - 3.500000 is -1.000000
2.500000 x 3.500000 is 8.750000
2.500000 / 3.500000 is 0.714286
Error: Divide by zero
Error: Unknown operator %
```