# CS211 – Programming Practicum Fall 2020

**Programming Project 3**

Due: Wednesday, 10/14/2020 at 11:59 pm

## Maze Solving

For this project, write a C program that will find its way through a maze using the depth-first search algorithm. You will traverse the maze and collect coins until you reach the destination or decide there is no path to the destination. This program takes input from a file where the filename is specified in the command line arguments. The input file will always be the same format: The first three lines contain two integer values per line while the rest of the lines contain two integers and a character value per line:

- The first valid line gives the size of the 2-D maze (the number of rows given by the first number, then the number of columns given by the second number), valid values are >= 1
- The second valid line gives the coordinates of the starting position in the maze
- The third valid line gives the coordinates of the ending position in the maze
- The remaining valid lines in the file give the coordinates of positions that are either blocked or contain "coins"
    - The blocked positions are marked by letter 'b' in input file
    - The positions containing coins are marked by letter 'c' in input file

If the command line arguments do not contain a valid filename, you should print an error message and quit the program. This could be because there was not a name given or the name given did not match that of a valid file. If the command line arguments contain multiple file names, give an error message stating that too many names were given and quit.

```
"Too many input files.\n"
```

The following shows an example of an input file. The coordinates are given with the row listed first and the column listed second. A maze of NxM has rows numbered from 1 to N and columns number from 1 to M. The "types" are given as a single character after the coordinates.

```
10 20
1 1
10 20
5 1 b
4 2 b
3 3 c
1 10 b
2 9 b
5 13 c
3 8 b
4 7 b
5 6 b
6 5 c
7 4 b
8 3 c
```

The above input creates the following maze will 8 blocked positions and 4 coin positions:

```
size: 10, 20
start: 1, 1
end: 10, 20

**********************
*s........*..........*
*........*..........*
*..C....*............*
*.*....*............*
**....*......C.......*
*....C...............*
*...*................*
*..C.................*
*....................*
*...................e*
**********************
```

The blocked positions and the edges of the above maze are filled in with *'s. The coin positions are filled with 'C'. The start position is filled in with an 's'. The end position in filled in with an 'e'. The other positions are filled in with periods.

You may assume that the input file will always have the same structure: two integer values on the first three lines and two integers and a character on the rest of the lines. However, the values may be out of range or the given character may not be correct. For character, only valid values are 'c' and 'b'. For integers, any value of zero or less is invalid in this program. For a coordinate value in the maze, valid value ranges from 1 to the maximum row size or column size. Also, the starting and ending positions of the maze must never be blocked.

If any values on the first three lines of input are invalid or the file ends before getting 3 valid lines of input, print out an error message to STANDARD ERROR and quit the program.
If the first three lines are valid, the maze is created and start and end positions are properly set.
From here on, for error handling, if an invalid value is given on an input line, print a descriptive error message to STANDARD ERROR, ignore those input values and continue processing input using the next line of input.
You may only ignore input lines that contain three values. In other words, you must ensure your maze is created first, and then proceed.

Input with invalid values is shown below. The comments on each line are not part of the input, but there to help explain the error.

Example:

```
10 0          => Invalid: Maze sizes must be greater than 0, exit
2 3
5 10
```

Example:

```
15 7          => Maze becomes size 15 x 7
10 20         => Invalid: column 20 is outside range from 1 to 7, exit
5 1
```

Example:

```
15 7          => Maze becomes size 15 x 7
5 1           => Starting position is at position 5, 1
3 3           => Ending position is at position 3, 3
1 10 b        => Invalid: column 10 is outside range from 1 to 7
2 9  c        => Invalid: column 9 is outside range from 1 to 7
3 8  b        => Invalid: column 8 is outside range from 1 to 7
4 7  b
5 6  b
5 1  c        => Invalid: attempting to block starting position
6 5  c
7 4  b
```

Here is a list of printf statements for different errors (for autograder):

```
"Invalid data file.\n" (for files less than 3 lines)
"Maze sizes must be greater than 0.\n"
"Start/End position outside of maze range.\n"
"Invalid coordinates: outside of maze range.\n"
"Invalid coordinates: attempting to block start/end position.\n"
"Invalid type: type is not recognized.\n"
```

The algorithm you are to use to find a path through the maze is a Depth First Search. You MUST use the following form Depth First Search**. You are NOT allowed to use a recursive version of the depth first search** (since you are required to use your own linked list stack to solve this).

- Mark all unblocked positions in the maze as "UNVISITED"
- push the start position's coordinates on the stack
- mark the start position as "VISITED"
- While (stack is not empty and end has not been found)
    - if the coordinate at the Top of the Stack is the end position
        - then end has been found (break out of loop)
    - if the coordinate at the Top of the Stack has an unvisited (and unblocked) neighbor
        - push the coordinates of one unvisited neighbor on the stack
        - if the position contains "coins"
            - increase the number of collected coins
        - mark that one unvisited neighbor as visited
    - else
        - pop the coordinate at the Top of the Stack
- If the stack is empty
    - The maze has no solution
- else

- The items on the stack contain the coordinates of the solution from the end of the maze to the start of the maze.
- Give the number of collected coins

When referring to neighbors, those positions will be the ones above, below, left or right of the current position (not diagonal). So for position x,y its neighbors are at:

- x+1, y => right
- x-1, y => left
- x, y+1 => down
- x, y-1 => up

The order in which you are going to evaluate your neighbors is { “right”, “down”, “left”, “up”}. You must use the following order to make sure your output is matching what autograder is expecting.

Your program is to first output the size of the maze, the start and ending coordinates and an ASCII drawing of the maze. The code **maze.c** creates an ASCII drawing of an empty maze of size 30X30 or less. The maze.c program uses a static sized 2-D array; **however, your program MUST use a dynamic 2-D array sized to reflect the maze size given in the input file.** You must also dynamically deallocate this array at the end of your program. The maze.c program also does not do any error checking for invalid input, your program MUST check for invalid input.

Once the maze solving algorithm is run, you must print out a message stating either:

- the maze has no solution

or

- listing the coordinates of the locations of the path in the maze **from the start of the maze to the end of the maze** that was found by the algorithm. Note this means printing out the contents of the stack in reverse order.

Example with solution:

```
size: 10, 20
start: 1, 1
end: 10, 20

**********************
*s........*..........*
*........*..........*
*..C....*...........*
*.*....*...........*
**....*......C.....*
*....C.............*
*...*..............*
*..C...............*
*..................*
*..................e*
**********************

This maze has a solution.
The number of coins collected: 2
```

```
The path from start to end:
(1,1) (2,1) (3,1) (3,2) (3,3) (4,3) (5,3) (6,3) (7,3) (8,3) (9,3) (10,3)
(10,4) (10,5) (10,6) (10,7) (10,8) (10,9) (10,10) (10,11) (10,12) (10,13)
(10,14) (10,15) (10,16) (10,17) (10,18) (10,19) (10,20)
```

Example with no solution:

```
size: 10, 15
start: 1, 1
end: 9, 14

*****************
*s........*.....*
*........*......*
*..C....*.......*
*.C....*........*
*C....*.........*
*....*.....C....*
*...*...........*
*..*............*
*.*......C....e.*
**..............*
*****************

This maze has no solution.
```

The stack MUST use a linked list of coordinate values. **The head of the linked list MUST NOT be global. It must be declared as a local variable in main( ) or some other function.** You may create a structure to contain the head of the stack if you desire but again the initial instance of the structure must be a local variable. The code for each stack operation MUST be done in its own function where the head is passed in as a parameter. The only exception to this is that the initializing function may return a newly created instance.

You MUST write functions for the following operations (these are the "same" functions as Project 2):

- initializing the stack,
- checking if the stack is empty,
- pushing an element onto the stack,
- popping an element off of the stack,
- accessing the top element on the stack, and
- resetting the stack so that it is empty and ready to be used again.

These functions must NOT contain memory leaks!

You get an **extra 10 points** if your solution passes the Valgrind test on Gradescope. The total points you can get from this project is 110/100.

## Command Line Argument: Debug Mode

Your program must be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program is to display the coordinates of the maze positions as they are pushed onto the stack and popped off the stack if the Top of Stack coordinate does not have an unvisited (and unblocked) neighbor. When the flag is not given, this debugging information should not be displayed.

```
printf ( "(%d,%d) pushed into the stack.\n", variable1 ,variable2);
printf ( "(%d,%d) popped off the stack.\n", variable1 ,variable2);
```

Since the input file for the maze also comes from the command line arguments, you may not assume which order in which the command line arguments are given. Thus the command line arguments may be given as:

- `./a.out mazeInput.txt`
- `./a.out mazeInput.txt -d`
- `./a.out -d mazeInput.txt`

One simple way to set up a "debugging" mode is to use a boolean variable which is set to true when debugging mode is turned on but false otherwise. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
        printf (" Debugging Information \n");
```

## *Coding Style*

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Function prototypes
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

## *Program Submission*

You are to submit the main.c for this project on **Gradescope**.