

# CSSE 152 Fundamentals of Computer Science II – Spring 2009

## Programming Assignment 2: Maze Puzzle Game

Preliminary design due: Tuesday April 28, 2009, 11:00am

Assignment and reflection due: Thursday, April 30, 2009 by midnight

### Summary

In this assignment, you will be creating a program that allows the user to solve a (not very common) maze puzzle. The user must traverse a maze (read in from a file) such that they visit every empty square exactly once ending with the end point. Since it is possible for the user to get stuck, your program should allow for a quick save and restore feature.

### Directions

1. At the start of the program, prompt the user for a file name. This file contains information related to a specific maze with the number of rows, columns, and indications of paths and walls.
2. The size of the maze is not fixed. The first row of the maze file will contain two numbers:  $m$   $n$ . This means the maze will consist of  $m$  rows and  $n$  columns. Then the file contains  $m$  lines each corresponding to a row of the maze. Each line will contain  $n$  characters. Each character can be one of the following:
  - 'B' Beginning point of the maze
  - 'E' Ending point of the maze
  - 'X' Barrier (user cannot go into a square with a barrier)
  - '.' Empty space
3. The program must then check to see if the maze is valid. A maze is valid if and only if the following criteria are met:
  - There is exactly one beginning point in the entire maze.
  - There is exactly one ending point in the entire maze.If the maze is invalid, exit the program with an error message.
4. Display the maze to the screen, using a '@' to represent the current position of the player (see example) and ask the user for a move.
5. A user may make one of the following moves by entering one of these uppercase letters:
  - 'U' Move up
  - 'D' Move down

- 'L' Move left
- 'R' Move right
- 'S' Save current arrangement of the board
- 'B' Revert back to previously saved arrangement of the maze
- 'Q' Quit

6. If the move is valid, redisplay the maze. Any squares previously visited by the player should be marked with a '\*'. A player can only visit a square exactly once, and so you must use a '\*' to mark visited squares. A '\*' should be treated like a barrier in future moves.
7. If the move is invalid (outside the boundary, hit a barrier, invalid input), display an error message and allow the user to try again. Keep in mind the exit is also a barrier if the user has not visited all of the empty squares.
8. Continue to process moves until the user either reaches the exit or quits the program.

## Saving and Restoring

Anytime the user presses 'S', the current state of the maze is saved by the program, including the current position of the player. Only one maze is saved by the program, pressing 'S' will overwrite any previously saved maze.

When they press 'B', the maze reverts back to the saved state. If the user enters 'B' before a maze is saved, it reverts back to the starting position. In other words, the program should save a copy of the original maze at the beginning.

See the section on **Program Design** for tips and requirements for implementing this feature.

## Error Checking and Assumptions

- Your program must handle the case where the input file does not exist. If it does not exist, exit the program with an appropriate message.
- If the file does exist, you can assume the input file meets the criteria listed in step 2 above.
- You cannot assume the maze is valid with respect to the criteria in step 4 above. You do not have to check other criteria such as determining if the maze has a solution or not.
- Assume that the user enters a single character at a time. Any character that is not one of the seven legal moves should be considered invalid. An error message should be printed and allow the user to try again.

## Example

On the following page, there is a portion of play in the middle of the program. Start on the upper left and proceed down the left column. When you reach the bottom, go to the right column. Note the save and restore functions in action.

```

*.....
*.X..
*.....
***@.
..X..
.....
....E
Enter move (UDLRBQ): U
*.....
*.X..
*..@.
****.
..X..
.....
....E
Enter move (UDLRBQ): L
*.....
*.X..
*.@*.
****.
..X..
.....
....E
Enter move (UDLRBQ): L
*.....
*.X..
*@**.
****.
..X..
.....
....E
Enter move (UDLRBQ): D
Cannot move - wall in way.
*.....
*.X..
*@**.
****.
..X..
.....
....E
Enter move (UDLRBQ): S
*.....
*.X..
*@**.
****.
..X..
.....
....E
Enter move (UDLRBQ): U

```

```

*.....
*@X..
****.
****.
..X..
.....
....E
Enter move (UDLRBQ): U
*@...
**X..
****.
****.
..X..
.....
....E
Enter move (UDLRBQ): R
**@..
**X..
****.
****.
..X..
.....
....E
Enter move (UDLRBQ): U
Cannot go outside edge of maze.
**@..
**X..
****.
****.
..X..
.....
....E
Enter move (UDLRBQ): B
*.....
*.X..
*@**.
****.
..X..
.....
....E
Enter move (UDLRBQ): U
*.....
*@X..
****.
****.
..X..
.....
....E
Enter move (UDLRBQ):

```

*continued on next column*

## Program Design

In this program, you must use a class to represent the maze grid. It must have the following member functions:

- a **constructor** that reads in the input file
- a **copy constructor**
- **overloaded assignment operator**
- **destructor**
- a function to determine if the maze **is valid**
- a function to **display** the board to the screen
- a function to process a **move**

The underlying implementation of the maze must consist of a dynamically allocated two-dimensional array. You may have additional data members.

To implement the save and restore feature, you will use the copy constructor and overloaded assignment operator. In main, you will have two maze variables. One that represents the current state of the maze and another that represents the saved state. Pressing 'S' will copy the current state into the saved state. Pressing 'B' will copy the saved state into the current state.

You are not allowed to do any of the following for the reasons described below:

- Use an array that is not dynamically allocated. You do not know the size of the maze ahead of time.
- Read the input file more than once. This is bad design – the input file could change when you read it the second time.
- Implement a default constructor. This (with the previous rule) forces you to use the copy constructor to initialize the saved maze.
- Store the saved maze with a duplicate set of data members thus eliminating the need for two maze variables. This is inflexible – a future iteration of the program may want to implement the ability to save more than one maze at a time.
- Create a member function that returns the array. There is no need to do this and violates the principle of information hiding.

You will use three files:

- `maze.cpp`: Implementation of the maze class
- `maze.h`: Interface of the maze class
- `main.cpp`: Main program, includes main and other functions as needed.

Three input files (called `maze1.txt`, `maze2.txt`, and `maze3.txt`) can be copied using the command:

```
cp /home/fac/roshanak/152files/PAs/PA2/maze*.txt .
```

Note: All three of the provided maze puzzles are solvable.

## Submitting your Programs

In addition to the electronic submission of your program, you also need to submit a brief design description for your program (in hard copy). Refer to Programming Style Sheet for CSSE 152 for information. **The preliminary design is due 2 days before the actual program.**

Run the following script in the directory with your program:

```
/home/fac/roshanak/submit/152/scripts/p2_runme
```

This will copy the file associated with this assignment to a directory that can be accessed by the instructor. If you get an error, please double check to make sure that you named the files correctly. You can view your submission at `/home/fac/roshanak/submit/152/*`

The submission program will try to compile your program using the following command line:

```
g++ -ansi -pedantic -Wall -Werror maze.cpp main.cpp
```

Your program must not produce any error messages of any kind (including warnings) or the submission program will reject your program. Programs that fail to compile will not be submitted and will not be graded.

Remember that an assignment will only be graded once.