Assignment 2
Motion Planning
Due: Monday, March 17, 2014, 11:59pm

You might not know this but the first version of Wall-E, the famous robot from the Pixar movie, was actually a cooking robot. Named **DAFFY (Drink And Food FactorY)**, the robot was designed to wake up every morning, make its way to the fridge, and cook three delicious meals for its owner. Unfortunately, the developers of DAFFY could not implement a motion planning algorithm for Wall-A, so it could never find its way to a fridge. Disappointed in the piece of junk they had created, they decided to use it for the only purpose it was good for: compacting junk.

In a parallel universe, you have been tasked with the job of implementing a workable motion planning algorithm for DAFFY. You can't work on an actual robot of course, but you can code the algorithm for the robot. As part of this assignment, you will be given a grid map of a one-storey home which contains the initial location of the robot and the fridge. However, when the robot is first powered up, it does not know the location of the fridge. Furthermore, it cannot sense anything more than one step away. So it tries to find the fridge using a *recursive backtracking* algorithm, moving one step at a time throughout the house. Once it has found the fridge, it can then use the path it discovered to navigate to the bridge every day.

Input Format:

The input file (entered as a command-line argument) will contain a map of the house in the form of a 2D grid. The map includes the starting location of the robot, indicated by R, and the location of the fridge, indicated by F. The walls of the house are represented by the hash symbol (#) and empty spaces represent parts of the map where the robot can go freely. You can assume that there will be always be a valid path from the starting point of the robot to the fridge. Here is an example:

```
################################
#             #                #
#            F#        #       #    #
#   ########################    #
#                               R#
################################
```

Output Format:

Your program should output another version of the map that contains a collision-free path from the start point of the robot to the fridge. The path would be indicated by the letter R placed along the path on the map. The output may show the robot wandering around a room to get to the fridge, but it should never retrace its steps or backtrack, i.e. it should never go over a spot it has already gone over before. Obviously, there can be several such paths to the fridge and your program should find and display only one such path. Here are two examples of valid outputs for the map shown above:

Example 1:

```
################################
#            #                 #
# RRRRRRRRRF#       #       #  #
# R#########################   #
# RRRRRRRRRRRRRRRRRRRRRRRRRRRR#
################################
```

Example 2:

```
################################
# RRRRRRRRRR#                  #
# RRRRRRRRRF#       #       #  #
# R#########################   #
# RRRRRRRRRRRRRRRRRRRRRRRRRRRR#
################################
```

Your main.cpp:

Most of your code will exist in one or more classes. Your main() function is required to be very simple. The following main() function should suffice:

```
int main(int argc, char** argv) {
  Robot robot(argv[1]);  //read in the input map file.
  robot.findFridge();    //find the path to the fridge
  robot.displayMap();   //display the map with the path that was found
}
```

Rules:

Your program should obey the following rules:

i) The robot can only take one step at a time and it cannot move diagonally. So it can only move to a maximum of four possible steps at a time.

ii) The robot can only see one step away and it cannot see diagonally. So it can only see a maximum of four cells at a time.

iii) The robot can replace the character stored in any point of the grid.

Example Interaction:

```
student@student-Virtualbox:~/cs212/asgn2$ ./fad-m input1.txt
################################
#            #                 #
#          F#       #      #   #
#  ########################    #
#                            R#
################################

Path To Fridge:

################################
# RRRRRRRRR#                   #
# RRRRRRRRRF#       #      #   #
# R########################    #
# RRRRRRRRRRRRRRRRRRRRRRRRRRRR#
################################
```

Example Input Files:

Three sample input files are provided at: http://tahirazim.com/cs212/assignments/assignment2/

Grading Criteria:

80% of the points in this assignment are for functionality. We are providing you three input maps on which your program should work correctly to get the 80% functionality points. Note that your code must follow the rules above to get these 80% points. Otherwise, you will get 0.

20% of the points are style points. This includes decomposing your code into functions, good naming of variables, good indentation, comments, and so on. For style, you will receive either 0, 10 or 20 points:

0 points means your code is in horrible condition.
10 points means your code could use some significant improvements.
20 points means your code is in reasonably good shape.

Bonus Extensions:

There are a number of interesting extensions that can be made in this assignment for extra credit. If you have implemented an extension, you still need to submit by the assignment deadline. However, to get credit for an extension, you need to demo your submitted code to me in the lab on Friday, March 21. Here are a couple of examples:

i) 10% extra credit: As you can see in one of the example outputs above, the path to the fridge may not be the shortest possible path. You can get 10% extra credit by adding

an extra function called `findShortestPathToFridge()` to your `Robot` class, which should find the optimal shortest path to the fridge. Obviously, this function should work correctly when called from `main()` in conjunction with `displayMap()`. This is only required to work correctly for the two smaller input maps. On the larger map, you may require more advanced algorithms which you will likely study in future courses.

ii) 5% extra credit: Implement additional algorithms that yield optimal or near-optimal paths from the starting point to the fridge. You can add additional functions of the form `findFridgeUsingNewAlgorithm()` but you need to make sure these functions work correctly when called from `main()` in conjunction with `displayMap()`.