Due: Friday, June 5, 2015 by 11:59pm

CPSC 221, Summer Term 2015
Programming Assignment 1

Date & Time of Latest Revision to this Document:

- posted May 27, 2015

History of Non-Trivial Changes: (For latest clarifications see UBC Connect).

Robbie the robot is on a reconnaissance mission. He must find all possible routes through a maze and report back to his (benevolent-ish) human overlords.

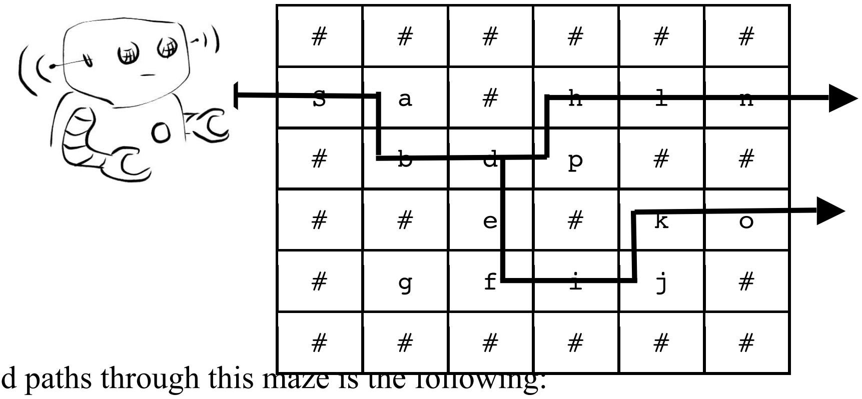
The maze consists of an `n x n` grid, consisting of # symbols, which represent walls, and assorted letters, [`a...z`] representing the stones along the path (i.e. where Robbie can travel). It is possible that these letters might repeat, but there will never be two of the same letters side by side.

The Rules of the Maze:

...are about as simple as could be! There is exactly one entrance to the maze, but there may be multiple exits. To simplify your life, all mazes will start one down from the upper left corner, and will always be labeled with an uppercase S. Exits could be anywhere along the right-most wall. The top wall and bottom walls are always solid rows of # symbols. Robbie's moves are restricted such that he can only move to an adjacent square that contains a letter, and he cannot walk through (or on) walls. Robbie must exit as soon as he can (that is, if there are two side-by-side exits, he will *always* take the first exit he encounters).

You must produce a list of all valid paths through the maze.

The following is an example of a 6 x 6 maze illustrating one possible, valid path.



#	#	#	#	#	#
S	a	#	h	l	n
#	b	d	p	#	#
#	#	e	#	k	o
#	g	f	i	j	#
#	#	#	#	#	#

The list of valid paths through this maze is the following:

```
S,a,b,d,e,f,i,j,k,o
S,a,b,d,p,h,l,n
```

Due: Friday, June 5, 2015 by 11:59pm

Your Task (to be completed in C++ *only*; classes are not necessary for this assignment):

I. Write a function called **`generate_all_paths`** that reads a **`15 x 15`** maze from a file and prints to the screen all possible, valid paths through that given maze. (**Tip**: When coding your solution, start with a very small sample maze, such as **`5 x 5`**, to test your approach, before scaling up.)

You may assume that all mazes have a perimeter wall and that all exits are via the right-most wall. You may assume that there are no loops, but a square may be used more than once in multiple paths.

Your program should produce output in the form of a list of all possible paths, as in the example program run below.

E.g. the small, sample maze on the previous page would appear in a plain text file (called **`maze.txt`**) as the following (and *nothing* else):

```
######
Sa#hln
#bdp##
##e#ko
#gfij#
######
```

A call to **`generate_all_paths`** would generate the following output to the screen:

```
     Path 1: S,a,b,d,e,f,i,j,k,o
     Path 2: S,a,b,d,p,h,l,n
     2 total paths
```

Don't forget to comment your code and include pre- and post-conditions for any functions you may write.

Deliverables:

- ONLY your source code, **`MazeSolver.cpp`**, any header files (**`.h`** files) and a **`README.txt`** file containing *your student information* (name, student number, lab and 4-char ID), a brief description of the approach you took, as well as any comments you may wish to pass on to the marker. If your **`README.txt`** file is missing, you will lose marks!
- You must assume that the markers will compile and run your program using the exactly the following, where **`maze.txt`** is the plain text, 15 x 15 maze file to be read in:
  ```
  >g++ -o MazeSolver MazeSolver.cpp
  >MazeSolver maze.txt
  ```
- Your program should simply output the results of the maze to the screen, and terminate.

Hints

First try to solve this problem without thinking about how you would implement it. Also, think carefully about how you store your maze when you read it in. It will always be square and a maximum size of 15 by 15.

Due Dates and Times
This assignment is due just *before* midnight at 11:59pm on Friday, June 5. Please use the handin assignment code “`pa1`".

Part marks will be awarded, so even if you don’t finish everything, *make sure that your code compiles on our undergrad machines* (this is the default environment) and that your code has at least some of the desired functionality, and then submit what you have completed.

*****WARNING***** Our part-mark policy emphasizes the incremental coding approach. That is, do **not** code large chunks of your program before testing. Code in small increments, ensuring that your code compiles and runs at each stage. This will make debugging **much easier**, and ensure that at almost any stage you have at least something you can turn in for partial marks.

Partners
You may work with a partner (no more than two people working together, *no exceptions*). If you choose to work with a partner, some words of caution: Ensure that *both* partners are equally well versed in the program being written (otherwise the partner not doing his share of the work will be at a disadvantage having had less experience with the course material). The best way to work together is to take turns playing the role of the “driver”, that is, the person on the keyboard. The other person should be discussing the project with the driver, ensuring that both understand what is happening. After a pre-designated amount of time (or after an objective has been met) be sure to switch roles.