EEN118 LAB ELEVEN

In today's lab you are going to write a program that explores a maze. This provides an example of arrays being used to do interesting things, introduces an interesting area of artificial intelligence (robotic navigation), and may give you some ideas about game programming if you are interested in that kind of thing.

This lab exercise is split into two sessions. On the first day you will be getting the display to look good, and making sure you can move about inside the maze. For the second day, you will work on automation, making the robot/man move around on his own, and making it more of a game.

The first task for your program will be to read a picture of a maze from a file, and store a suitable representation of it in memory. The file containing the picture will be called "`maze.txt`", and it will contain a text representation of a rectangular maze, with the character '`X`' used to represent a wall, and '`.`' will be used to represent open space. This is an example of one possible maze file:

```
XXXXXXXXXXXXXXXXXXXXX
X.......X.....XX.X...X
X.XX.XX.XXX.X..X.X.X.X
X..X.XX.X.X.XX...X...X
XX.X........XXX.XX.XXX
XX.XXXXXXXXXXX....X.X
X......XX....XXXXX...X
XXXXXX..XX.XXX...XXX.X
X.............X..aXbX
XXXXXXXXXXXXXXXXXXXXX
```

It would be a little easier to look at if we used spaces instead of dots, but remember that C++ likes to ignore spaces when reading inputs, and we don't want to introduce unnecessary complications. To save typing, you can download that maze file from:

`http://rabbit.eng.miami.edu/class/een118/maze.txt`

Notice that there is a solid wall of `X`s around the maze. All valid mazes will be surrounded like this; it means you don't have to worry about your robot accidentally wandering out of the maze and falling off the edge of the world. Notice also that there is a single '`a`' and a single '`b`' in the maze, near the bottom right corner. Every valid maze will have one '`a`' and one '`b`' in it. They denote the start and end points; the object is to get your robot from '`a`' to '`b`' without using any squares marked '`X`'. Diagonal moves are not allowed, and no maze will have more than 80 rows or 80 columns.

1. ***(Basic)*** Read the Maze.

Build a program that first opens the file and creates an appropriate two-dimensional array to store the maze, then reads the maze into the array, then closes the file and proves that it read the maze correctly by printing it again in a slightly different format.

It makes perfect sense to make your array be as big as it could ever need to be, instead of trying to work out the correct size for each maze and creating a perfectly-sized array for it.

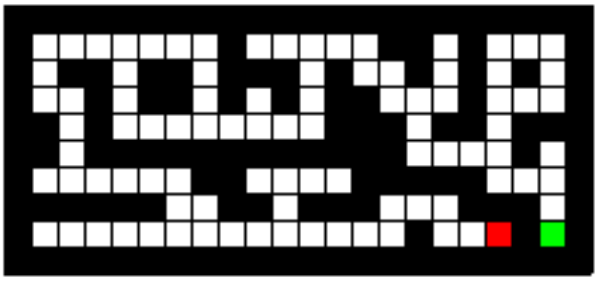
How should you print the maze after reading it? Rememer that the reason for doing this is to be absolutely certain that you really have read the maze correctly, we don't want there to be any risk that you'll accidentally print out the contents of the file and trick yourself. Wait until you have read the entire file and closed it, then reprint the maze from your array. Perhaps use stars to indicate walls and spaces for spaces. It will be easy to see that the maze is the right shape, but it will obviously not be just a copy.

2. ***(Basic)*** Detect A and B.

Modify your program so that it notices where the '`a`' and '`b`' are, and stores the coordinates (row and column) of each point in suitable variables.

3. ***(Basic)*** Draw it Properly.

Instead of drawing the maze with stars and spaces, open a graphics window and draw it properly. I would suggest first drawing a grid of the right size, then filling in the wall parts with a solid colour. Make sure each block occupies enough pixels to be seen clearly. Then mark the positions of the '`A`' and '`B`' in some way that stands out. Perhaps a different coloured blob for each. The '`A`' represents where your robot is now (naturally he hasn't moved yet) so perhaps a very small robot shape could represent it; the '`B`' represents the robot's target, so perhaps something a bit treasure-like would work. Don't waste a lot of time on good graphical representations of robots and treasure until you've got the rest of the lab exercises working. I just went for red and green squares.



4. ***(Basic)*** *Make the Robot Move.*

The library contains a function called `read_char_no_wait();` it takes a single character typed by the user *if one has been typed*. It does not wait for a whole line to be typed. It does not even wait for a single character to be typed. The function returns the ASCII code of the character typed if there was one, or zero if nothing has been typed yet. This is how you should use it to read a single character without requiring the user to press ENTER (which is very annoying for game playing):

```
int c;
```

```
while (true)
{ c=read_char_no_wait();
  if (c!=0) break;
  Delay(0.01); }
```

The `Delay(0.01)` is there so that your program does not use up all the CPU's capacity in a tight little loop just waiting for input. Remember that you are probably the only person using that computer, so your game gains nothing by being greedy.

After that piece of program, you can see what character was typed very easily, just remember that single quotes are used to get ASCII codes, not the double quotes that you are familiar with from strings:

```
if (c=='N') ...move the robot North...
else if (c=='S') ...move the robot South...
else .....
```

Use this to let the user (i.e. you) control the robot. If you type 'N' it should move one square to the North; if you type 'S' it should move one square to the South, etc., etc. Remember that C++ is case sensitive: if you test for 'N', but type 'n', there will be no match. Of course you can choose what keys you use for the commands. Perhaps you prefer U=up, D=down, L=left, R=Right, or something else. Include a 'quit' or 'exit' command so that the user can easily stop the program.

Do not worry about walking through walls or falling off the edge of the world. Just update the robots position after each move. The player will just have to be careful to navigate properly.

Try it out, make sure you can navigate the robot to his target. I hope you remembered to update the picture after each movement, so that you see the robot in its new position.

5. ***(Basic)*** Walking Through Walls is Bad.

   Make the exploration more realistic by refusing to obey impossoble commands. If a movement would result in the robot walking through (or into) a wall, then that movement command must not be obeyed.

You should hope to be somewhere around step six (just before or just after) before the beginning of the second session (2nd or 3rd December).

6. ***(Basic)*** A Foolish Robot.

   Your robot does not move except under your direct command. Add another command letter, perhaps 'A', that puts him in Automatic mode. When in automatic mode, the robot should select a random neighbouring empty square, move to it, then wait a second (just so that you can see what is happening), and continue moving in automatic mode until stopped.

   Of course, you need another letter command, perhaps 'M' for Manual, that turns off automatic mode, and puts the robot back in your direct control. This is not difficult, you just have to think about exactly where in your program you need to check for user input.

7. ***(Intermediate)*** Retracing Steps.

In order to avoid getting lost, it is important for explorers to remember the path they followed, so that they can always reliably retrace their steps and get back home. Give your explorer this ability. The easiest way is to create a second array, the same size as the maze-map array. Every time the robot explorer moves into a square, how he got there (which direction he moved in) is stored in the corresponding position in the second array. If ever the robot needs to go back, he just looks in the new array to see what direction he came in, and moves in the opposite direction.

Add a '`B`' command (for "Back") to your program. Each time '`B`' is typed, the explorer should retrace his steps by one square. If you keep typing '`B`', he should eventually get all the way back to his starting position.

Important: the "how I got here" array should only be updated after normal movements, not after '`B`' commands. Otherwise two '`B`'s in a row will have no overall effect.

8. ***(Intermediate)*** Semi-Intelligent Behaviour.

It is fairly easy to make your robot explore the maze totally unaided, and find a path to the treasure (if a path exists). This is the strategy:

- Whenever the robot finds himself anywhere in the maze, he should examine each of the four neighbouring squares. If there is any neighbouring square that he has never been in before, he should move into it.
- If there is more than one unexplored neighbouring square, just pick any one.
- If ever the robot finds there are no neighbouring squares that he has never visited before, then he should retrace his steps by one square, and continue the procedure from there.
- If ever the robot finds himself at the destination square ('`B`') he has solved the maze, and should stop.
- If ever the robot finds himself trying to retrace his steps back from the starting square ('A'), that means that the maze has no solution, and he should give up.

Automate your explorer, so that he follows this strategy when you put him in Automatic mode (you can make up your own if you like, but it won't be easy to find another one that always works) and finds his way from '`A`' to '`B`' without any help. You will almost certainly want to put a `Delay(`*xxx*`)` inside the loop so that you get a chance to see what is happening.

This strategy is what makes this a valid robotics experiment. Even though the program has posession of a complete map of the maze, it doesn't "cheat" by using it too much. At every step it only considers the immediately neighbouring squares when deciding what to do next, working under the same restrictions as a real robot lost in a maze, only having the information that its own sensors can gather.

9. ***(Intermediate)*** The yellow brick road.

Once the robot successfully reaches the target 'B', it would be nice to make the whole path he took clearly visible, perhaps be colouring those squares in a way that stands out. Of course, we would only want to see the squares he visited that were on the successful path, not ones that he later backed out of by retracing his steps.

The information required to do this very easily is already recorded in your program. Try to work out what to do, and then do it.

10. ***(Advanced)*** Monsters.

Put a monster at a random position in the maze. Every time the robot makes a move, the monster also makes a move, heading towards the robot's position one square at a time, not being allowed to walk through walls of course. If the monster reaches the robot, you can decide what happens.

To make it more of an interesting game, you could make the monster move at its own pace. Instead of just taking turns, with the monster only making a move after your robot has made its own move, you could perhaps have the monster move one square every second, regardless of what your robot has done.

You could have different monsters with diffrent degrees of intelligence: the dimmest ones could just move at random, cleverer ones could rush for the treasure and try to ambush your robot on the way.

You probablt have a better idea of what makes a good video game than I have, so just get on with it. Make it good.

11. ***(Intermediate)*** Appearance.

How about making it look better? Your robot (and the monsters) could be represented by little pictures. Perhaps, in order to have enough space for a good monster picture, you could reduce the number of maze squares shown. Even though the maze may be 40x40 in size, you might only show the squares within 4 or 5 units of distance from your robot's current position. That would add an element of surprise: you would never know where the moster or the treasure is until it is close, and possibly you would never even know how many monsters there are.

Use your imagination, and maybe you'll have a program that you can actually be pleased with.