EEN118 LAB TWELVE

In this lab and the next, 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.

Keep in mind that lab thirteen is a continuation of this program, so don't lose it. It would probably be a good idea to read the lab 13 assignment too, so that you know what's coming.

Today you will be getting the display to look good, and making sure you can move a little robot about inside the maze. In the second session, you will work on automation, making the robot/man/woman/whatever move around on his/her own, and making it more of a game. From now on, I'll just refer to the moving thing as "the robot".

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 input, 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. *Read the Maze*

Build a program that creates an appropriate two-dimensional array to store the maze, opens the file, reads the maze into the array, then closes the file and proves to you 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.

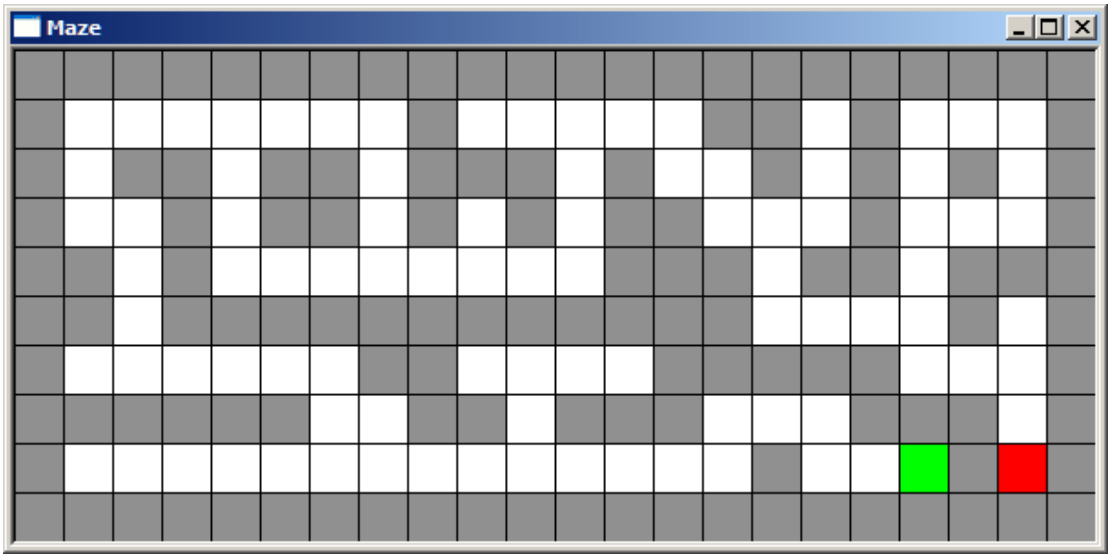
How should you print the maze after reading it? Rememer that the reason for doing this is to be absolutely certain that you really have read the maze correctly, we don't want there to be any risk that you'll accidentally print out the contents of the file and trick yourself. Wait until you have read 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. *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. *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 square 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 here:



4. *Make the Robot Move*

The library contains a function called `wait_for_key_typed();` it waits until the user types a key on the keyboard, then returns as its result the ASCII code for that key. Use it to make the robot move around under your direct control.

Remember that C++ does not expect you to have memorised all the ASCII codes. If you put a single character inside *single* quotes, C++ sees it as the ASCII code for that character. So, if you choose to use the letters `l`, `r`, `u`, and `d` to stand for `left`, `right`, `up`, and `down`, you might have something like this in your program.

```
while (true)
{ char c = wait_for_key_typed();
  if (c=='l')
  { /* make the robot move one square to the left */ }
  else if (c=='r')
  { /* make the robot move one square to the right */ }
  else if (c=='u')
  { /* make the robot move one square up */ }
  else if (c=='d')
  { /* make the robot move one square down */ } }
```

Note that this function does not behave like "`cin <<`". It does not wait until enter has been pressed at the end of the line, and it only takes keystrokes that were aimed at the graphics window. If the black text window is selected, keys typed go to `cin`, and `wait_for_key_typed()` doesn't get to see them. The graphics window must be selected.

You might like to take advantage of the fact that all the non-ASCII keys have also been assigned numeric codes. In particular, the arrow keys have negative numbers assigned to them as follows:

down arrow	`-88`
right arrow	`-89`
up arrow	`-90`
left arrow	`-91`

It might also be a good idea to have an `x` (exit) or `q` (quit) command for when you get fed up with playing.

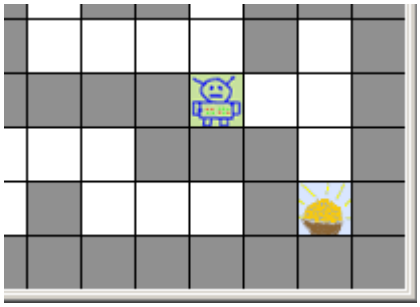
Do not worry about walking through walls or falling off the edge of the world. Just update the robot's position after each move. The player will have to be careful to navigate properly.

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. *Prevent Walking Through Walls*

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


Perhaps you would like to know about two more useful library functions that will display graphics from image files. You could have proper pictures of the robot and the treasure, and even backgrounds for the maze squares:



One is called `image_from_file()`. It takes as its one parameter the name of a file, which may be any `.gif`, `.jpg`, or `.bmp` file, and extracts the image data from it. It returns a strange value whose type in C++ is "`image *`".

The second function is called `draw_image`. It takes three parameters: one of those "`image *`" things I just mentioned, and two `int`s that specify the x and y coordinates of the position in the window where the image should be drawn (top left corner).

It is a very simple process. When the program is first started, it might have a statement or two that look something like this:

```
image * robot_icon = image_from_file("robot.gif");
image * gold_icon = image_from_file("c:/progs/lab12/gold.jpg");
```

and later, when it is time to draw the robot, you would simply write

```
draw_image(robot_icon, x, y);
```

where of course `x` and `y` are replaced by the appropriate position coordinates.