EEN118 LAB TWELVE

A few weeks ago, in lab ten, you wrote a program that lets a human user control the computer's exploration of a maze. Today you will be experimenting with auomatic exploration, the beginning of artificial intelligence. It is perfectly acceptable to modify and re-use any of the programs you may already have written. This lab also gives you a little experience of programming with objects.

1. Make sure it Still Works.

You should already have a program that reads in a maze as illustrated below (height, width, starting row, starting column, then the maze picture), and draws it using the `graphics.h` library functions. Make sure that you remember how it works.

Example maze in the style of lab ten:

```
10
20
1
1
XXXXXXXXXXXXXXXXXXXX
X        X     XX X
X XX XX XXX X  X   X
X  X XX X X XX   X X
XX X         XXX XX X
XX XXXXXX XXXXX    X
X       XX    XXXX XX
XXXXXX  XX XXX    XX
X               XX  X
XXXXXXXXXXXXXXXXXXXX
```

However, this week, the maze descriptions will be slightly different. The input will consist of the maze-picture alone, no numbers. The program will have to work out how big the maze is for itself. That is actually quite easy, and because you have already programmed something very much like it before, I'll give you most of the program here:

```
#include <vital.h>

struct mazedata
{ int map[50][50];
  int rows, cols; };
```

That definition creates a new kind of object, called a "`mazedata`". Any `mazedata` variable will have three components called "`map`", "`rows`", and "`cols`". Grouping them into a single object makes it convenient and efficient to pass all the information about a maze into a function with a single parameter. I hope you remember all that from class. Note how the function `read-maze` (below) is defined. It only has one parameter, but that parameter is a whole `mazedata` object, so all three components are received at once. Remember that the `&` in front of the parameter's name) which makes it into a *Reference Parameter*) is needed if the function is allowed to modify the parameter, but it also improves efficiency.

```
void read_maze(mazedata &m)
{ m.rows=0; m.cols=0;
  int r, c;
  for (r=0; r<50; r+=1)
  { for (c=0; c<50; c+=1)
    { m.map[r][c]=-1; } }
  for (r=0; r<50; r+=1)
  { string s=read_line();
    int linelength=s.length();
    if (linelength==0) break;
    m.rows+=1;
    if (linelength>m.cols) m.cols=linelength;
    for (c=0; c<linelength; c+=1)
    { if (s[c]==' ')
      { m.map[r][c]=0; } } } }
```

The `int`s stored in a `mazedata`'s `map` component are `0` for an open space, and `-1` for a solid wall. The function first fills the map with solid wall so that there can be no accidental gaps, then it reads the picture line by line. If you have read a string into a variable called `s`, then `s.length()` tells you how many characters are in that string (i.e. its length). This function uses a blank line to signify the end of the maze picture; blank lines have a length of zero.

The `read_maze` function works out the size (`rows`, `cols`) of the maze as it reads it. After each line is successfully read, `rows` is incremented, and `cols` is set to the length of the longest line seen so far. The function scans through each line, storing a zero in the maze's `map` every time it sees a blank space.

The `print_maze` function below could be used to test the program, but is not really good enough. We really want to be using proper graphics:

```
void print_maze(const mazedata &m)
{ int r, c;
  for (r=0; r<m.rows; r+=1)
  { for (c=0; c<m.cols; c+=1)
    { if (m.map[r][c]==0)
        print(".");
      else
        print("*"); }
    newline(); } }

void main(void)
{ mazedata X;
  read_maze(X);
  print("This is the maze that I read:"); newline();
  print_maze(X); }
```

Remember that an ampersand "&" in front of a parameter's name makes a *reference* parameter. References make it efficient to pass large objects (such as whole mazes) into a function, and also make it possible to modify the parameter. If we want the extra efficiency, but don't want the modifiablilty (to ensure that the parameter can not be accidentally modified), the special word `const` is used.

You must now modify this function (or write a whole new one of your own if you prefer). Your function should be able to read a maze as shown in the example below. The maze picture contains four different characters: '`X`' represents solid wall, space represents open space, '`A`' rep-

resents your starting position in the maze, and '`B`' represents the position you are trying to get to. As you read the maze, you should note the positions of the '`A`' and the '`B`', but do not record them in the array; they should be treated as open spaces. It would be a good idea to add four new integer variables to the `mazedata` object, `Arow`, `Acol`, `Brow`, `Bcol`.

Test your program by making it print the maze immediately after reading it (but using different characters to represent walls and spaces, so you can be sure the program is really doing something and not just echoing what it reads), together with a report on the positions of the A and B.

Example Input:

```
XXXXXXXXXXXXXXXXXXXX
X       X     XXBX X
X XX XX XXX X  X   X
X  X XX X X XX   X X
XX X        XXX XX X
XX XXXXXX XXXXX    X
X      XX    XXXX XX
XXXXXX  XX XXX    XX
X   A          XX  X
XXXXXXXXXXXXXXXXXXXX
```

Corresponding Output:

```
********************
*.......*.....**.*.*
*.**.**.***.*..*...*
*..*.**.*.*.**...*.*
**.*........***.**.*
**.******.*****....*
*......**....****.**
******..**.***....**
*.............**..*
********************
start(A) row 8 col 4
end(B) row 1 col 16
```

Remember that it is traditional in C++ to start counting at 0, so row 1 is really the second row from the top, and column 16 is really the 17th column.

2. Rendering the Maze.

Here is the definition of `draw_block()`, from lab ten.

```
void draw_block(int col, int row, int numcols, int numrows)
{ window.fillarea(col*width/numcols, row*height/numrows,
                  width/numcols-1, height/numrows-1); }
```

This function assumes that the `width` and `height` of the window in which we are drawing the maze are defined as global constants. Remember that the coordinates of the maze start at (0,0) (so that (1,5) refers to what you would normally think of as the second column of the sixth row).

Adapt your `draw_maze()` function from lab ten so that it can be used with the new maze structure, and correctly draw the maze it represents, with a special shape or symbol indicating the position of the explorer (initially the same as the position of the A) *and* the explorer's target (the position of the B). Using different colours is probably easier than drawing different shapes.

3. An Unintelligent Robot.

Make your explorer automatically explore the maze. Don't bother with any hint of intelligence yet, just get him moving around. Recall from class that the simplest way to explore any kind of map is with a little recursive function. In order to explore from a particular location, you find all the other locations that can be reached directly from it (in this case, that is any of the four immediate neighbours that are not solid wall), and (recursively) explore from each of them.

Every time your program starts to explore from a new location, it should update the graphics display so that you can actually see the explorer moving around. **Important**: remember that you must use `window.update()` and the `delay(1.0)` function to make the program pause for a second or so after each change, or you'll never see anything. Don't worry that your program never terminates, just use control-C to stop it after a while.

4. An More Intelligent Robot.

You almost certainly noticed that after a very short while your explorer gets stuck in a rut, just going backwards and forwards between a few (probably only two) locations. This is because it has no memory of where it has been before, so can't possibly avoid following the same path every time.

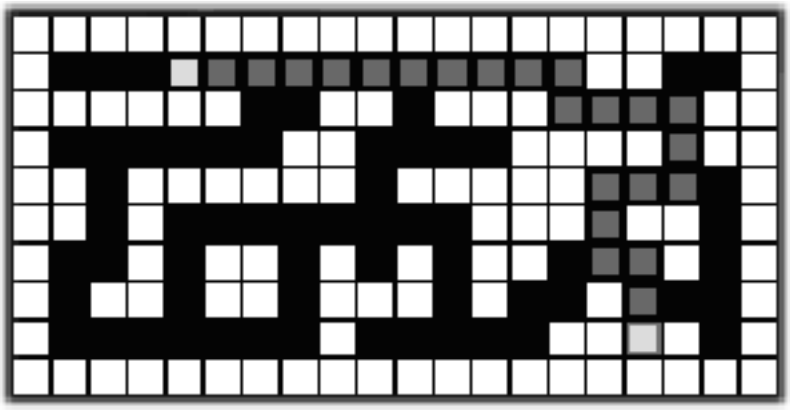
Give your robot a memory. It should always know how many steps away from its starting point it is (if you are using a recursive exploring function this is very easy. Distance from home should be a parameter to the exploring function, and with every recursive call it should be incremented. Returning from the recursions will naturally decrement the parameter at the right time). Every time it goes to a new space, it should record how far from home it is, and it should never revisit a space that it has already found a shorter path to. This is exactly the same as the "shortest route between two cities" algorithm we investigated in class.

This is a bit tricky. Expect to spend some time making this part work properly.

If you get it right, you will find that your explorer re-explores parts of the maze quite a large number of times (each time it finds a slightly shorter route to a place it has already visited, it will have to revisit that place and everywhere else reached through it). While you are experimenting you will definitely want to be working with a much smaller maze than the example provided earlier.

5. Verification.

Make the program tell you the length of the shortest route between the start and end positions (A and B) once it has completed its explorations. That way you can check that it is correct. Try to find a way to draw the exact route (think of how we did it for the road map in class) in a human-friendly manner.



6. Anything Else.

Once you have got the robot automatically exploring the whole maze, you can easily make all sorts of other things happen. Try putting some other robots in, perhaps searching for the same target. Perhaps the robots are not friendly with each other. Use your imagination and try to make an interesting program.