Extended Exercise A – Robot Demolition Derby

Occasionally, to help reinforce some of the programming concepts we have learned so far (and to have a little fun!), we will have an extended exercise for students to work on.

In this extended exercise, we will practice the use of arrays and structs by implementing the following game.

GAME DESCRIPTION

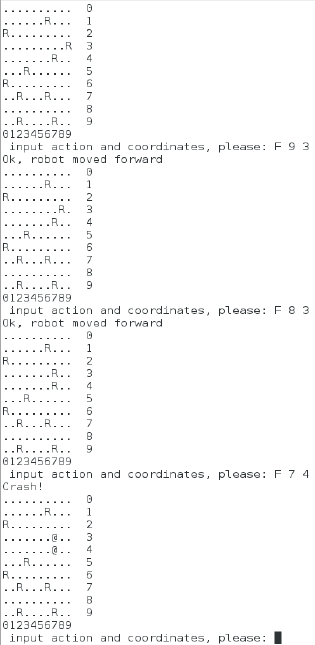
The game is played on a 10 by 10 grid, where each space could be occupied by a robot. The goal of the game is to crash all of the robots into one another, until none of them are still running.  The challenge is that we cannot tell which direction they are initially pointing — we can only see their position on the grid.

The user plays the game by entering a series of actions, each of which is given by a character F, L, or R, and then two integers representing the current coordinates of the robot you want to command.  The three possible actions are:

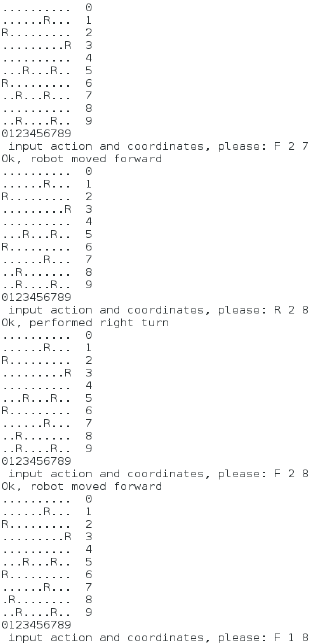
* Move the robot Forward one square.
* Rotate the robot to the Left by 90 degrees.
* Rotate the robot to the Right by 90 degrees.

**Moving Forward.**This is done by entering the letter F followed by the coordinates of where the robot is on the grid.  The robot then moves to the adjacent site in the direction of its current heading (west, north, east, south).  Again, the problem is that the user cannot directly see the direction the robot is pointing, and can only tell this by observing where the robot goes when telling it to move forward.

If the move would cause the robot to leave the grid, the robot stays where it is. However, if the user tries to make the robot move to a site occupied by another robot, there is a crash.  Both robots involved become inactive (dead) and can no longer be moved.



**Rotating the Robot**.  Entering the letter L followed by the robots’ coordinates turns the heading of the robot to the left by 90 degrees (i.e. west turns into south, south into east, etc.)  Likewise, the letter R followed by coordinates turns the heading of the robot to the right.

[](http://sst-csci.com/csci151/wp-content/uploads/illu2_crop.png)

IMPLEMENTING THE GAME

To help get you started, we have provided the following stub file, [robots.c](http://sst-csci.com/csci151/wp-content/uploads/robots-3.c).  At the top of the file, note that we have the following struct definition to represent a single square on the 10 by 10 grid:



|  |  |
| --- | --- |
| 1  2  3  4  5 | typedef struct {     \_Bool hasRobot; // 1 (true) a robot is here; 0 (false) the space is clear     int robHeading; // 0 (west), 1 (north), 2 (east), 3 (south)     \_Bool robAlive; // 1 (true) for running robot; (false) when crashed  } gridSquare; |

At the beginning of the main function, a 10 x 10 array of **gridSquare** items is declared.  After this, you need to initialize each of the **hasRobot** members of the grid to false using a nested loop (marked by the TODO comment).  Don’t worry about setting **robHeading**and **robAlive** right away, since these values don’t mean anything if the square doe not have a robot in it.

Next, we randomly choose 10 places on the grid to place a robot.  Note that we are using the **rand()** function here from math.h to randomly generate int values, and change the resulting range from 0 to 9 by using the modulus (%) operator.  Also note that we are using a do-while loop here to re-generate coordinates in the case where we might place one robot on top of another.  However, you still need to add code to randomly set the heading to 0, 1, 2, or 3, and you also see to make the robot “alive”.

After this, we have the main control loop (a do-while loop) which does the following until there are no more live robots:

1. Outputs the contents of the 10 by 10 grid, which you need to implement.  If the place doesn’t have a robot, then simply print a dot.  If it contains a “live” robot, print the character ‘R’.  If the robot is “crashed”, print the character ‘@’.
2. Gets the command from the user.
3. Executes the command, given that the coordinates indicate a proper robot — you will have to implement the details for each case here.  Note that code is already provided for the “Forward” command to prevent the robot from falling off the edge of the grid.

One thing to remember: if one robot crashes into another, both robots are no longer “alive”.  However, it is possible to crash into a robot that is no longer alive, so you need to be careful to properly update the **liveCount** variable in either case.

Extended Exercise B – Minesweeper

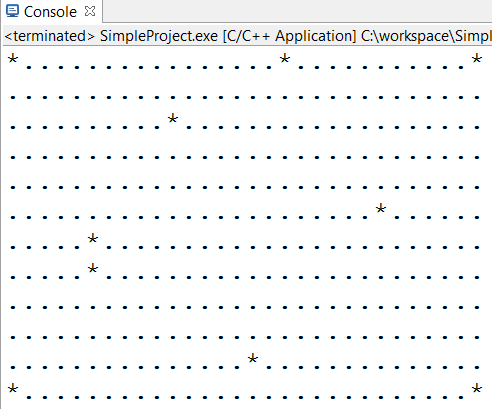
For this extended exercise, we are going to start putting together the basic parts of the well-known game, [minesweeper](http://minesweeperonline.com/).

For the first version of our game, we will randomly place 10 mines on a 12 x 30 row-column grid.

PART 1 – GENERATING THE MINEFIELD

Main Steps:

1. Declare a 12 x 30 row-column matrix of chars – this will be your **map** showing where the mines are.  Using nested for-loops, initialize your **map** so that all spots have a period (i.e., ‘.’), which means that no mine is there.
2. Randomly generate 10 mine coordinate pairs, where each pair is a valid row and column in the map.  Set the corresponding spot in **map** to a star (i.e., ‘\*’), meaning that a mine is there.  However, if a mine already exists in the same spot, you need to choose new coordinates for an empty spot in the **map**.
3. To make sure everything is working so far, output the contents of **map** to the screen; you should create a separate function for this. The output should look something like this:

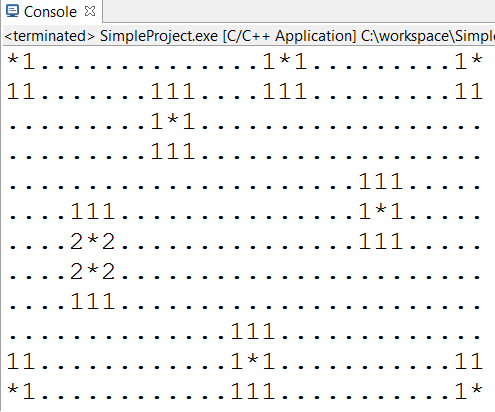
[](http://sst-csci.com/csci151/wp-content/uploads/output1.png)

PART 2 – DISPLAYING PROXIMITY INFORMATION

For this part, modify your **map** so that instead of displaying just where the mines are (and are not), you also provide “proximity” information for each of the cells.  For a cell that does not contain a mine, its “proximity number” is the number of mines that are directly around it in the eight cells that encircle it.  However, don’t modify the cells that:

* Contain a mine, or
* Have proximity number 0 (leave those as dots)

Here is an example of what the resulting **map** might look like:

[](http://sst-csci.com/csci151/wp-content/uploads/output2.png)

To do this, create a new function **addProximities** that is called after your code that adds the \*’s to the **map** in the previous step.  In the body of **addProximities**, use a nested for loop which essentially “counts” the number of \*’s around a given location.  Specifically, inside of this new nested loop, for each cell with coordinate (r, c), do the following:

* Don’t bother with cells that have ‘\*’s in them.
* You may want to create a new counter variable **count** that is incremented for each of the cells neighboring (r, c) that contains a **‘\*’.**
* When checking each of the cells around location (r, c), you need to first make sure that you are not on the “edge” of the map, and thus access something that is “out of bounds”  To do this, you may want to create a separate if statement for each of the eight neighbors: (r-1, c-1); (r-1, c); (r-1, c+1); (r, c-1); (r, c+1); (r+1, c-1); (r+1, c); (r+1, c+1).
* To convert the count digit to a proper character for the map, you can do something like this:  **map[j][k] = count + ‘0’;**

PART 3 – PLAYING THE GAME: USER INTERACTION

Keep in mind that during the game, the player should only see portions of the map, and at the very beginning, it should be completely covered from view.  You can keep track of what is visible by yet another 12 x 30 matrix of booleans called **view**.  Here, true means that the location in the **map** should be visible to the user; otherwise, it should be currently hidden.

So, to actually play the game, you will have another loop that:

1. First gets a coordinate pair (two ints) from the user, and then “uncovers” that cell by setting those coordinates in **view** to true.  You should probably check to make sure that they are legal coordinates first, though.
2. If a mine is uncovered, the game is over — print out “BOOOOM!!!” and the full contents of **map**.
3. If a mine wasn’t uncovered, output the contents of **map** whose corresponding cells in **view** is true, or output a ‘?” if the cell in **view** is false.
4. Win check — if every non-mine cell in **map** has been uncovered, the player wins and the game is over.  An easy way to do this is to count the number of legal guesses the player has made so far, and if it hits 12 x 30 – 10, then the player has won.

Be sure to define and use new functions, where appropriate.

POSSIBLE ENHANCEMENTS

* Modify your program so that it can create a map of variable dimensions R x C, and any number of mines M, where R, C, and M are read in from the user at the beginning.
* While playing the current version of your game, you might get tired of counting off rows and columns.  Add row and column labels to your visible map output in the console — to handle two digit numbers, just use two additional rows at the top to display your column labels, and two additional columns at the left side to display your row labels.
* Minesweeper generally lets you put “flags” on certain spots where you are sure that there is a mine (and prevents you from accidentally selecting one of these spots.)  Implement this feature in your game.
* If you select a cell that has proximity number 0, you know you can safely choose all of the neighbors of this cell.  Minesweeper usually will automatically choose these for you to save time.  Implement this feature in your game.