Goal:

The goal of this lab is to complete the movement functionality for the game. When completed your robot should be able to move around the board and interact with all terrain types.

You MUST work in pairs. If you don't have a lab partner, either find one or ask for help finding one. The TA's have been instructed not to help people who are not working with a lab partner.

For each part of the lab you should compile and test your code before continuing.

Part 1: Get the new game template file

1. Download the new Project Template from Blackboard in the Assignment 1 section of the Introduction to C learning module.
2. Unzip the template after moving it to wherever you like in your Linux directories.

my-dir% **unzip robot_template3.zip**

You can probably use the `game.c` included in the template since it contains a solution for most of the objectives of the last lab. If you want to use parts of your code from a previous lab, be aware that some changes were made to game.h and main.h (meaning that your old code might not compile without modification).

Note random levels and energy have been removed from he game. Now levels are loaded from a file (read robot_factory.pdf for all the changes).

Complie the program and run it. You can specify which levels file it uses or it will default to `levels.txt`:

```
% ./r my_levels.txt
```

Part 2: Implement interactions:

Test the game out and note what works and what doesn't. Specifically, ice and explosion hazards do not work. Also, next and previous levels do not correctly stop at the maximum level in the file.

When you run the game the first level looks like this:

```
# # # # # # # # # # # # #
# ~ ~ ~ ~ ~ . . R . X . #
# ~ r r r ~ . . . . X . #
# ~ r R r ~ . . . . X . #
# ~ r r r ~ . . . R X . #
# ~ ~ ~ ~ ~ R # . . . . #
# . R . . . . . . . X . #
# . R ~ . . . . . . . . #
# . R E . . . . R = = * #
# . R = = R . . R = = # #
# . R * . . . . . R = . #
# . R + . . . . . . . . #
# # # # # # # # # # # # #
```

Moving to the right gives this:

```
# # # # # # # # # # # # #
# ~ ~ ~ ~ ~ . . . R X . #
# ~ r r r ~ . . . . X . #
# ~ r R R ~ . . . . X . #
# ~ r r r ~ . . . . 2 . #
# ~ ~ ~ ~ ~ R # . . . . #
# . . R . . . . . . X . #
# . r ~ . . . . . . . . #
# . . E . . . . . 1 = * #
# . . 1 = . R . . 1 = # #
# . . * . . . . . . 1 . #
# . . R . . . . . . . . #
# # # # # # # # # # # # #
```

Level 2 of the game shows what the result should be. The '1' and '2' characters you see are the start of ice-slides and explosions, respectively. Both constants are defined at the top of `game.c`: `kSlideStart` and `kExplosionSite`

Design and implement an algorithm to process complete the ice-slides and explosions. If you'd like to use a completely different algorithm, you are free to change any code in `game.c`. The recommended method is to complete the `DoExplosions` and `DoIceSlides` functions in `game.c` since they are already called for you in the appropriate places. Each algorithm is outlined below in the following sections.

Part 3: Implement Ice-Slides:

The idea of the algorithm is as follows (you need to implement this in C code):

```
DoIceSlides(dx, dy):
   scan board:
      if kSlideStart is found:
         store ice at the site
         find last ice square in the (dx,dy) direction
         put robot in last ice location
         call move_location to move robot
         if robot moved away:
             put ice in previous robot location
```

Finding the last ice in the direction of movement will involve a loop to scan in that direction. Your code will also need to store the last ice location, perhaps in variables called `lastIceX` and `lastIceY`.

Part 3: Implement Explosions:

The idea of the algorithm is as follows (you need to implement this in C code):

```
DoExplosions :
   scan board:
      if kExplosionSite is found:
         make the site empty
         scan surrounding squares:
            if kExplosionHazard:
               set site to kExplosionSite
               DoExplosions
            if  kRobot or kMovedRobot:
               killed_robot(1)
               set site to empty
            if kExit:
               do nothing
            otherwise:
               set site to empty
   restore the outer wall
```

When you remove a robot from the game, you need to call `killed_robots(1)` to inform the code in `main.c` that a robot was killed. This is so we can have level ending conditions which include killing a certain number of robots.

Part 4: Design an interesting level:

Create a file called my_levels.txt and design some levels to test your game and create a challenge for the player.

If you come up with good ones, email me the text or the file, and I'll include them in a puzzles file for the whole class.

Part 5: Final features:

There are a few more features that will need to be completed in the lab next week. Here they are so you can get started on them:

A. Make next and previous level commands ('n' and 'p') work correctly. Currently next level does not stop counting at the last level in your file. You will need to use the Boolean value returned by `load_level` to adjust the level number. The correct behavior should never show a blank level (unless that's what the levels file contains).

B. Make a counter for the number of robots saved and killed and display that counter when you draw the game. This is so the player can track those values (they are not part of the visible game).

C. For some bonus marks, implement Mega Man and program him to do something interesting. (see robot_factory.pdf)