Project 1 Appendix: Battle for Zion (the Original)

The Machines that created The Matrix are attacking Zion and Neo is nowhere to be found! Before they reach Zion you've been tasked with holding them off. Well, that's the scenario for a new video game under development. Your assignment is to complete the prototype that uses character-based graphics.

If you execute the sample Windows, Mac, or Linux program provided on the website, you will see the player (indicated by @) in a rectangular arena filled with killer robots (usually indicated by R). At each turn, the user will select an action for the player to take. The player will take the action, and then each robot will move one step in a random direction. If a robot lands on the position occupied by the player, the player dies.

This smaller version of the game may help you see the operation of the game more clearly.

At each turn the player may take one of these actions:

- Stand. In this case, the player does not move or shoot.
- Move one step up, down, left, or right. If the player attempts to move out of the arena (e.g., down, when on the bottom row), the result is the same as standing. It is allowable for the player to move to a position currently occupied by a robot. If no robot occupies that position after the robots have moved, the player survived the turn.
- Shoot in one of the four directions up, down, left, or right. If there is at least one robot within 5 steps in that direction (this only matters with anything larger than a 6x6 arena), the nearest one in that direction is the candidate victim. If more than one robot is nearest (i.e., they occupy the same position), only one robot at that position is the candidate victim. With 2/3 probability, the candidate victim is damaged; with 1/3 probability, the shot is ineffective and no robot is damaged. A robot that has been damaged for the second time is destroyed (i.e., to destroy a robot takes two shots that hit).

The game allows the user to select the player's action: u/d/l/r for movement, su/sd/sl/sr for shooting, and just hitting enter for standing. The user may also type q to prematurely quit the game, or c to have the computer select the player's move.

When it's the robots' turn, each robot picks a random direction (up, down, left, or right) with equal probability. The robot moves one step in that direction if it can; if the robot attempts to move out of the arena, however, (e.g., down, when on the bottom row), it does not move. More than one robot may occupy the same position; in that case, instead of R, the display will show a digit character indicating the number of robots at that position (where 9 indicates 9 or more). If after the robots move, a robot occupies the same position as the player, the player dies.

Your assignment is to (1) organize the C++ program skeleton main.cpp provided on the website in appropriate header and implementation files, and (2) complete the code that implements the described behavior in the paragraphs above. (We've indicated where you have work to do by comments containing the text TODO; remove those comments as you finish each thing you have to do.) The program skeleton you are to flesh out defines four classes that represent the four kinds of objects this program works with: Game, Arena, Robot, and Player. Details of the interface to these classes are in the program skeleton, but here are the essential responsibilities of each class:

Game

- To create a Game, you specify a number of rows and columns and the number of robots to start with. The Game object creates an appropriately sized Arena and populates it with the Player and the Robots.
- A game may be played.

Arena

- When an Arena object of a particular size is created, it has no robots or player. In the Arena coordinate system, row 1, column 1 is the upper-left-most position that can be occupied by a Robot or Player. (If an Arena were created with 10 rows and 20 columns, then the lower-right-most position that could be occupied would be row 10, column 20.)
- You may tell the Arena object to create or destroy a Robot at a particular position.
- You may tell the Arena object to create a Player at a particular position.
- You may tell the Arena object to have all the robots in it make their move.
- You may ask the Arena object its size, how many robots are at a particular position, and how many robots altogether are in the Arena.
- You may ask the Arena object for access to its player.
- An Arena object may be displayed on the screen, showing the locations of the robots and player, along with other status information.

Player

- A Player is created at some position (using the Arena coordinate system, where row 1, column 1 is the upper-left-most position, etc.).
- You may tell a Player to stand, move in a direction, or shoot in a direction.
- You may tell a Player to take an action of its choice.
- You may tell a Player it has died.
- You may ask a Player for its position, alive/dead status, and age. (The age is the count of how many turns the player has survived.)

Robot

- A robot is created at some position (using the Arena coordinate system, where row 1, column 1 is the upper-left-most position, etc.).
- You may tell a Robot to move.

- You may ask a Robot object for its position.

The skeleton program has all of the class definitions and implementations in one source file, which is awkward.

Organize the code

Take the single source file, and divide it into appropriate header files and implementation files, one pair of files for each class. Place the main routine in its own file named main.cpp.
Make sure each file #includes the headers it needs. Each header file must have include guards.

Now what about the global constants? Place them in their own header file named globals.h. How about utility functions like decodeDirection or clearScreen? Place them in their own implementation file named utilities.cpp and place their prototype declarations in globals.h.
From the collection of the eleven files produced as a result of this part of the project, make sure you can build an executable file that behaves exactly the same way as the original single-file program.

To make sure you organized your code correctly your solution should pass these requirements.

Complete the code

Complete the implementation in accordance with the description of the game. You are allowed to make whatever changes you want to the private parts of the classes: You may add or remove private data members or private member functions or change their types. You must not make any deletions, additions, or changes to the public interface of any of these classes — we're depending on them staying the same so that we can test your programs. You can, of course, make changes to the implementations of public member functions, since the callers of the function wouldn't have to change any of the code they write to call the function. You must not declare any public data members, nor use any global variables other than the global constants already in the skeleton code, except that you may add additional global constants if you wish. You may add additional functions that are not members of any class. The word friend must not appear in your program.

Any member functions you implement must never put an object into an invalid state, one that will cause a problem later on. (For example, bad things could come from placing a robot outside the arena.) Any function that has a reasonable way of indicating failure through its return value should do so. Constructors pose a special difficulty because they can't return a value. If a constructor can't do its job, we have it write an error message and exit the program with failure by calling exit(1);. (We haven't learned about throwing an exception to signal constructor failure.)

Turn it in

You will turn in for this project a zip file containing only the eleven files you produced, no more and no less. The files must have these names exactly:

Robot.h	Player.h	Arena.h	Game.h	globals.h	
Robot.cpp	Player.cpp	Arena.cpp	Game.cpp	utilities.cpp	main.cpp

To create a zip file on a lab machine, you can select the eleven files you want to turn in, right click, and select "Send To / Compressed (zipped) Folder". Under Mac OS X, copy the files into a new folder, select the folder in Finder, and select File / Compress "folderName"; make sure you copied the files into the folder instead of creating aliases to the files.

If we take these eleven source files, we must be able to successfully build an executable using Visual C++ and one using clang++ or g++ — you must not introduce compilation or link errors.

If you do not follow the requirements in the above paragraphs, you will suffer a deduction in points.

The only exception to the requirement that the zip file contain exactly eleven files of the indicated names is that if you create the zip file under Mac OS X, it is acceptable if it contains the additional files that the Mac OS X zip utility sometimes introduces: __MACOSX, .DS_Store, and names starting with ._ that contain your file names.

In summary, what you will turn in via BruinLearn/Canvas for this assignment is:

- A zip file containing the eleven files listed above. This program must build successfully, and its correctness must not depend on undefined program behavior. Your program must not leak memory: Any object dynamically allocated during the execution of your program must be deleted (once only, of course) by the time your main routine returns normally.
- A report typed into a Word document or text file that will contain:
    - A brief description of notable obstacles you overcame
    - A description of what works in your program and what does not work.