CS 31 Programming Assignment 7

Rage Against the Machines

The Mechanoids of Planet Zork have captured you! To satisfy their lust for oilshed and bloodshed, they have armed you with a club and pitted you in combat against killer robots. 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 this Windows program or this Mac program or this Linux program, 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 Windows version or Mac version or Linux 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:

1. Move one step up, down, left, or right to an empty position. 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 (i.e., not moving at all).
2. Attack an adjacent robot. If the player indicates up, down, left, or right, and there is a robot at that position, then this means the player does not move, but instead clubs the robot. This damages the robot; if this is the second time the robot has been damaged, it is destroyed. If this is only the first time the robot has been damaged, the momentum from the club swing knocks the robot back to the position behind it (i.e. adjacent to it on the opposite side from the side the player is on relative to the robot). If the robot is on the edge of the arena, so there is no position behind it, then it is destroyed (presumably because being knocked into the wall of the arena does a second amount of damage, enough to destroy it). A robot knocked back to a position occupied by one or more robots (it is allowable for multiple robots to

occupy the same position) does not suffer additional damage and does not damage those robots or knock them to another position. When the player attacks a position occupied by more than one robot, only one of those robots is damaged and either destroyed or knocked back one position; the others are unaffected.

3. Stand. In this case, the player does not move or attack.

The game allows the user to select the player's action: u/d/l/r for moving or attacking, or just hitting enter for standing. The user may also type q to prematurely quit the game.

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.

The CS 31 assignment was to complete a C++ program skeleton to produce a program that implements the described behavior. For CS 32 Project 1, we will give you a correct solution to the CS 31 assignment. The program 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, 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. (Notice that the arena is displayed after the robots have had their turn to move, but not after the player has moved. Therefore, if a player knocks a robot back one position, the robot will have a chance to move before you see the display, so it might appear to have moved one more

position back, or to the side, or, a glutton for punishment, back next to the player.)

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 7 rows and 8 columns, then the lower-right-most position that could be occupied would be row 7, column 8.)
- 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 tell the Arena object that a robot is being attacked, and find out whether the attack destroyed the robot.
- 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 whether moving from a particular position in a particular direction is possible (i.e., would not go off the edge of the arena), and if so, what the resulting position would be.
- 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 or to move or attack in a direction.
- 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.
- You may tell a Robot object that it has been attacked, and find out whether that attack destroyed the robot.

CS 31 students were told:

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 and will, 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 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.)

Programming Assignment 1

Rage Against the Machines

Time due: 9:00 PM Tuesday, January 12

The appendix to this document is the specification of the last CS 31 project from a previous quarter. We will provide you with a correct[1] solution to that project. Your assignment is to (1) organize the code for the solution in appropriate header and implementation files, and (2) implement a new feature.

You should read the appendix now. It describes a game in which a player has to survive in an arena full of killer robots. You will be working with this code that implements the game. Notice that it is a single file. (Just so you know, the way we tested its correctness is similar to how we'll test the correctness of the programs you write in CS 32.)

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 `#include`s 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`. And what 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`.

The Visual C++ 2015 and the Xcode writeups demonstrate how to create a multi-file project. 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.

Add a feature

If you try running the updated programs (the Windows version, the Mac version, or the Linux version of the full game, and theWindows version, the Mac version, or the Linux version of the smaller version of the game), you'll see they have one new command you can type: `h` for history. This command shows you for each grid point, how many times during the course of the game the player has dealt a fatal blow to a robot that was standing at that point: dot means 0, a letter character A through Y means 1 through 25 (A means 1, B means 2, etc.) and Z means 26 or more.

Your task is to implement this functionality. You will need to do the following:

- Define a class named `History` with the following public interface:

```
class History
{
  public:
    History(int nRows, int nCols);
    bool record(int r, int c);
    void display() const;
};
```

  - The constructor initializes a History object that corresponds to an Arena with `nRows` rows and `nCols`columns. You may assume (i.e., you do not have to check) that it will be called with a first argument that does not exceed `MAXROWS` and a second that does not exceed `MAXCOLS`, and that neither argument will be less than 1.
  - The `record` function is to be called to notify the History object that a robot has died at a grid point in the Arena that the History object corresponds to. The function returns false if row `r`, column `c` is not within bounds; otherwise, it returns true after recording what it needs to. This function expects its parameters to be expressed in the same coordinate system as the Arena (e.g., row 1, column 1 is the upper-left-most position).
  - The `display` function clears the screen and displays the history grid as the posted programs do. This function *does* clear the

screen and write an empty line after the history grid; it does *not* print the `Press enter to continue.` after the display.

The class declaration (with any private members you choose to add to support your implementation) must be in a file named `History.h`, and the implementation of the History class's member functions must be in `History.cpp`. If you wish, you may add a public destructor to the History class. You must *not* add any other *public* members to the class. (This implies, for example, that you must *not* add a public default constructor.) The only member function of the History class that may write to `cout` is `History::display`.

- Add a data member of type History (*not* of type pointer-to-History) to the Arena class, and provide this public function to access it; notice that it returns a *reference* to a History object.

```
class Arena
{
  …
  History& history();
  …
};
```

- When a robot dies, its arena's history object must be notified about the position where it died.
- Have the Game recognize the new `h` command and tell the arena's history object to display the history grid, and then have the Game print the `Press enter to continue.` prompt and wait for the user to respond. (`cin.ignore(10000,'\n');` does that nicely.) Typing the `h` command does not count as the player's turn.

Turn it in

By Monday, January 11, there will be a link on the class webpage that will enable you to turn in your source files. You do not have to turn in a report or other documentation for this project. What you will turn in for this project will be one zip file containing *only* the thirteen files you produced, no more and no less. The files must have these names *exactly*:

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

The zip file itself may be named whatever you like.

If we take these thirteen 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, your score on this project will be zero. "Do you mean that if I do everything right except misspell a file name or include an extra file or leave off one semicolon, I'll get no points whatsoever?" Yes. That seems harsh, but attention to detail is an important skill in this field. A draconian grading policy certainly encourages you to develop this skill.

The only exception to the requirement that the zip file contain exactly thirteen 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.

To not get a zero on this project, your program must meet these requirements as well:

- Except to add the member function `Arena::history`, you must not make any additions or changes to the public interface of any of the classes. (You are free to make changes to the private members and to the implementations of the member functions.) The word `friend` must not appear in any of the files you submit.
- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Game.h"
#include "Game.h"
#include "Arena.h"
```

```
#include "Arena.h"
#include "History.h"
#include "History.h"
#include "Player.h"
#include "Player.h"
#include "Robot.h"
#include "Robot.h"
#include "globals.h"
#include "globals.h"
int main()
{}
```

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "History.h"
int main()
{
    History h(2, 2);
    h.record(1, 1);
    h.display();
}
```

`History.h` must not contain any `#include` line that, if removed, still allows the above replacement `main.cpp` to compile successfully under both Visual C++ and either clang++ or g++.

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Robot.h"
int main()
{
    Robot r(nullptr, 1, 1);
}
```

`Robot.h` must not contain any `#include` line that, if removed, still allows the above replacement `main.cpp` to compile successfully under both Visual C++ and either clang++ or g++.

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Player.h"
int main()
{
    Player p(nullptr, 1, 1);
}
```

`Player.h` must not contain any `#include` line that, if removed, still allows the above replacement `main.cpp` to compile successfully under both Visual C++ and either clang++ or g++.

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Arena.h"
int main()
{
    Arena a(10, 18);
    a.addPlayer(2, 2);
}
```

`Arena.h` must not contain any `#include` line that, if removed, still allows the above replacement `main.cpp` to compile successfully under both Visual C++ and g++, except that Arena.h should include `globals.h`. (Even if`History.h` includes `globals.h` and `Arena.h` includes`History.h`, good practice says that the author of `Arena.h`who wants to use `MAXROBOTS` and knows that it's declared in `globals.h` shouldn't have to wonder whether some other header already includes `globals.h`.)

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "globals.h"
#include "Player.h"
#include "Arena.h"
int main()
```

```
{
    Arena a(10, 20);
    Player p(&a, 2, 3);
}
```

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Arena.h"
#include "Player.h"
int main()
{
    Arena a(10, 20);
    Player p(&a, 2, 3);
}
```

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Player.h"
#include "Arena.h"
int main()
{
    Arena a(10, 20);
    Player p(&a, 2, 3);
}
```

- If we replace your `main.cpp` file with the following, the program must build successfully under both Visual C++ and either clang++ or g++:

```
#include "Arena.h"
#include "History.h"
#include "Player.h"
#include "globals.h"
int main()
{
    Arena a(4, 4);
    a.addPlayer(2, 4);
    a.addRobot(3, 2);
    a.addRobot(2, 3);
```

```
    a.addRobot(1, 4);
  a.player()->moveOrAttack(LEFT);
  a.player()->moveOrAttack(UP);
  a.player()->moveOrAttack(LEFT);
  a.player()->moveOrAttack(LEFT);
  a.player()->moveOrAttack(DOWN);
  a.player()->moveOrAttack(DOWN);
  a.player()->moveOrAttack(LEFT);
  a.player()->moveOrAttack(UP);
  a.player()->moveOrAttack(UP);
  a.player()->moveOrAttack(UP);
    a.history().display();
 }
```

When executed, it must clear the screen (*à la* `Arena::display`), and write the following five lines (the fifth line is an empty line):

```
...A   <== This is the first line that must be written.
.B..   <== This is the second line that must be written.
....   <== This is the third line that must be written.
....   <== This is the fourth line that must be written.
       <== This empty line is the fifth line that must be
written.
```

- If we replace your `main.cpp` file with the following, the program must *not* build successfully; attempting to compile it should produce an error message like `'r' uses undefined class 'Robot'` or `variable has incomplete type 'Robot'` or `variable 'Robot r' has initializer but incomplete type` (and perhaps other error messages):

```
#include "Player.h"
#include "Arena.h"
int main()
{
    Arena a(10, 20);
    Player p(&a, 2, 3);
    Robot r(&a, 1, 1);
}
```

- If we replace your `main.cpp` file with the following, the program must *not* build successfully; attempting to compile it should produce an error message like `'a' uses undefined class 'Arena'` or `variable has incomplete type 'Arena'` or `variable 'Arena a' has initializer but incomplete type` (and perhaps other error messages):

```
#include "globals.h"
#include "Robot.h"
#include "Player.h"
int main()
{
    Arena a(10, 10);
}
```

- If we replace your `main.cpp` file with the following, the program must *not* build successfully; attempting to compile it should produce an error message like `'History' : no appropriate default constructor available` or `no matching constructor for initialization of 'History'` or `no matching function for call to 'History::History()'` (and perhaps other error messages):

```
#include "History.h"
int main()
{
    History h;
}
```

- If a .cpp file uses a class or function declared in a particular header file, then it should #include that header. The idea is that someone writing a .cpp file should not worry about which header files include other header files. For example, a .cpp file using an A object and a B object should include both A.h (where presumably the class A is declared) and B.h (where B is declared), without considering whether or not A.h includes B.h or vice versa.

To create a zip file on a SEASnet machine, you can select the thirteen 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.

Advice

Developing your solution incrementally will make your work easier. Start by making sure you can build and run the original program successfully with the one source file having the name `main.cpp`. Then, starting with Robot, say, produce `Robot.h`, removing the code declaring the Robot class from `main.cpp`, but leaving in `main.cpp` the implementation of the Robot member functions. Get that two-file solution to work. Also, make sure you meet those of the requirements above that involve only the `Robot.h` header.

Next, split off `Player.h`, testing the now three-file solution and also making sure you meet those of the requirements above that involve only the `Robot.h` and `Player.h` headers. Continue in this manner until you've produced all the required headers (except `History.h`, since you're not yet adding the history feature), the whole program still works, and you meet all the applicable requirements.

Now split off the member function implementations of, say, Robot, putting them in `Robot.cpp`. Test everything again. You see where this is going. The basic principle is to *not* try to produce all the files at once, because many misconceptions you have will affect many files. This will make it difficult to fix all those problems, since many of them will interfere with each other. By tackling one file at a time, and importantly, not proceeding to another until you've got everything so far working, you'll keep the job manageable, increasing the likelihood of completing the project successfully and, as a nice bonus, reducing the amount of time you spend on it.

Help

While we will provide you assistance in clarifying what this assignment is asking for and in using Visual C++ and either clang++ or g++, we will otherwise offer minimal help with this assignment. This is to give you a chance to honestly evaluate your own current programming ability. If you find that you're having trouble with the C++ program itself (not simply the Visual C++, Xcode, or g++ environment, which may be new to you), then you may want to reconsider your decision to take this class this quarter. Perhaps you've let your C++ programming skills get rusty, or maybe you

didn't learn the material in CS 31 or its equivalent very well. If you decide to take the course later, what you should do between now and then is program, program, program! Solve some old or current CS 31 or PIC 10A or early PIC 10B projects, and read and do the exercises in a good introductory programming text using C++. You'll have to be self-motivated to make time for that, but the payoff will be a greater likelihood for success in CS 32.

Endnote

[1] "Correct" in terms of what a CS 31 student would know. For example, a CS 31 student wouldn't know that sometimes you need to write a copy constructor, so the posted solution ignores that issue. (You don't have to worry about that issue for this project, either.)