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 the Windows 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.)