Programming Assignment 1
Club Zombie

Time due: 11:00 PM Tuesday, April 9

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 zombies. 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 correctnessis 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 `randInt` or `clearScreen` that are used by the implementations of more than one class? Place them in their own implementation file named `utilities.cpp`, and place their prototype declarations in `globals.h`. (Note that for utility functions like `decodeDirection` that are used by only one class implementation, it would be better to declare and implement them in that class's implementation file, not in `globals.h`.)

The Visual C++ 2017 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. Make sure you can also do this with g++ with Linux.

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 Mac version or the Linux version of the

smaller version of the game [sorry, no Windows version]), 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 was at that point when the player dealt a fatal blow to a zombie: 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 the player was at a grid point (in the Arena that the History object corresponds to) when the player dealt a fatal blow to a zombie. 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, display the history grid, and write an empty line after the history grid; it does *not* print the `Press enter to continue.` after the display. (Note to Xcode users: It is acceptable that clearScreen() just writes a newline instead of clearing the screen if you launch your program from within Xcode, since the Xcode output window doesn't have the capability of being cleared.)

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 zombie dies, its arena's history object must be notified about the position of the player when the zombie died.
- Have the Game recognize the new `h` command, tell its arena's history object to display the history grid, and then 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, April 8, 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*:

Zombie.h	Player.h	History.h	Arena.h	Game.h	globals.h	
Zombie.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 g32[2] and one using either Visual C++ or clang++ — 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 macOS, it is acceptable if it contains the additional files that the macOS 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` and the word sequence `pragma once` must not appear in any of the files you submit.
- Your program must not use any global variables whose values may change during execution. Global *constants* (e.g. `MAXROWS`) are all right.
- Except in `Game::play` (or a function it calls) and `History::display`, your program must write no output to `cout` beyond what is already in the program or is required by this spec. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.
- If we replace your `main.cpp` file with the following, the program must build successfully under both g32 and either Visual C++ or clang++:
- ```
  #include "Zombie.h"
  int main()
  {}
  ```

  Furthermore, if `Zombie.h` contains the line `#include "`*`XXXXX`*`.h"`, where *XXXX* represents any of `Game`, `Arena`, `History`, `Player`, `Zombie`, or `globals`, and if removing that `#include "`*`XXXX`*`.h"` line from `Zombie.h` still allows the above program to build successfully, then `Zombie.h` must not contain that `#include "`*`XXXX`*`.h"` line. For example, if `Zombie.h` contained `#include "Game.h"`, but removing `#include "Game.h"` from it still allowed the little three-line program above to build successfully, then`Zombie.h` must not contain `#include "Game.h"`.

- If, in the text and little program of the requirement above, you replace all occurrences of `Zombie.h` with `Player.h`, you must meet the resulting
  ```

requirement. Similarly, if you replace all occurrences of `Zombie.h` with `History.h`, you must meet the resulting requirement. The same also holds if you replace with `Game.h` and with `globals.h`. The requirement must also be met if you replace with `Arena.h` with one exception: `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 `MAXZOMBIES` 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 g32 and either Visual C++ or clang++:

```
#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 "Zombie.h"
#include "Zombie.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 g32 and either Visual C++ or clang++:

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

- If we replace your `main.cpp` file with the following, the program must build successfully under both g32 and either Visual C++ or clang++:

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

- If we replace your `main.cpp` file with the following, the program must build successfully under both g32 and either Visual C++ or clang++:

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

- If we replace your `main.cpp` file with the following, the program must build successfully under both g32 and either Visual C++ or clang++:

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

- If we replace your `main.cpp` file with the following, the program must build successfully under both g32 and either Visual C++ or clang++:

```
#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 g32 and either Visual C++ or clang++:

```
#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 g32 and either Visual C++ or clang++:

```
#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 g32 and either Visual C++ or clang++:

```
#include "Arena.h"
#include "History.h"
#include "Player.h"
#include "globals.h"
#include <iostream>
using namespace std;

int main()
{
    Arena a(3, 4);
    a.addPlayer(2, 3);
    a.addZombie(2, 2);
    a.addZombie(2, 4);
    a.addZombie(1, 3);
    a.player()->moveOrAttack(RIGHT);
    a.player()->moveOrAttack(LEFT);
    a.player()->moveOrAttack(LEFT);
    a.player()->moveOrAttack(LEFT);
    a.player()->moveOrAttack(RIGHT);
    a.player()->moveOrAttack(UP);
    a.history().display();
    cout << "====" << endl;
}
```

  When executed, it must clear the screen (*à la* `Arena::display`), and write the following five lines and no others:

```
....   <== This is the first line that must be written.
.AB.   <== This is the second line that must be written.
....   <== This is the third line that must be written.
       <== This empty line is the fourth line that must be written.
====   <== This 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 `'p' uses undefined class 'Player'` or `variable has incomplete type 'Player'` or `variable 'Player p' has initializer but incomplete type` (and perhaps other error messages):

```
#include "Zombie.h"
#include "Arena.h"
int main()
{
    Arena a(10, 20);
    Player p(&a, 2, 3);
    Zombie z(&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 `'z' uses undefined class 'Zombie'` or `variable has incomplete type 'Zombie'` or `variable 'Zombie z' 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);
    Zombie z(&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 "Zombie.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 macOS, 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 Zombie, say, produce `Zombie.h`, removing the code declaring the Zombie class from `main.cpp`, but leaving in `main.cpp` the implementation of the Zombie member functions. Get that two-file solution to work. Also, make sure you meet those of the requirements above that involve only the `Zombie.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 `Zombie.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, Zombie, putting them in `Zombie.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.