Major Practical: Fight Arena Game

1. Originality and complexity of selected problem

Project Idea: This will be a two-player based game where the player gets to choose between four types of characters and fight to win by getting opponent's health to zero. Players will have the following attributes: Health, Armour, Mana, and some hidden magical powers (depends on the selected player type). Using different types of attacks, the players will either defend or attack the opponent stats.

Design Requirement:

- ⇨ **Memory allocation:** The game requires string datatype (stack memory) to store the names of the player and arrays of size 3 (dynamic memory) to store the stats of both players. A range of objects are required to make the game as efficient as possible.
- ⇨ **User Input and Output:** The game will mostly revolve around I/O of different data types, and will check with a while loop if incorrect input is entered.
- ⇨ **Object-oriented programming and design:** Player can select from 4 children classes that are inherited from main player class. Player class is an abstract class and character type are allocated using polymorphism.

2. Specification

- ⇨ Each player has three stats: health, armour, and mana; and has a name. Also, after each turn it checks if the stats are in the range of 0 to 100
- ⇨ The game calculates the final overall result of which player won after playing the number of rounds entered
- ⇨ The game updates the leader board after each round is plays, that is, updates the points of the player won in that round
- ⇨ The game displays the correct colour codes wherever required
- ⇨ Each of the 5 shots are fired correctly and the stats that get updated are enhanced before it is displayed
- ⇨ Anytime if an incorrect input is entered, the game prompts an error message with asking to enter again, and works until a correct input is entered
- ⇨ The game waits for 2 seconds after each round is played
- ⇨ If a player selected to be an archer, then after a random number of turns, a random number is generated between 1 and 4. According to which, a pre-defined magic power appears using a switch statement. You get unlucky by rolling 3 or extremely lucky by rolling 4.

3. Readability

- ⇨ **Structure:** a class diagram is used to structure the code of the game and is used to demonstrate the flow of the game. Also, it can be used to present virtually how the program works.
- ⇨ **Commenting:** the program is well commented on every stage where it needs slight explanation
- ⇨ The files are named correctly and operated using make files. Also, every for loops, if statements, switch statements, and while statements are correctly idented.
- ⇨ The variable names are used in a way to make the code more interpretable

4. Efficiency

- ⇨ To make the program more efficient, required arguments are initialised before using the behaviours of the classes to prevent passing these arguments each time to use those behaviours
- ⇨ Leader board statistics, and names are stored in stack memory, as they are never passed as an argument. The player objects are stored in dynamic memory to use polymorphism and also because it must be passed many times as argument
- ⇨ At any point of time, if user inputs other than the requirement, then an error message is displayed. This is done by making a custom class input which checks every time if the input is entered with the specifications. Enhanced input cases are rigorously checked, including inputs with spaces and/or special characters

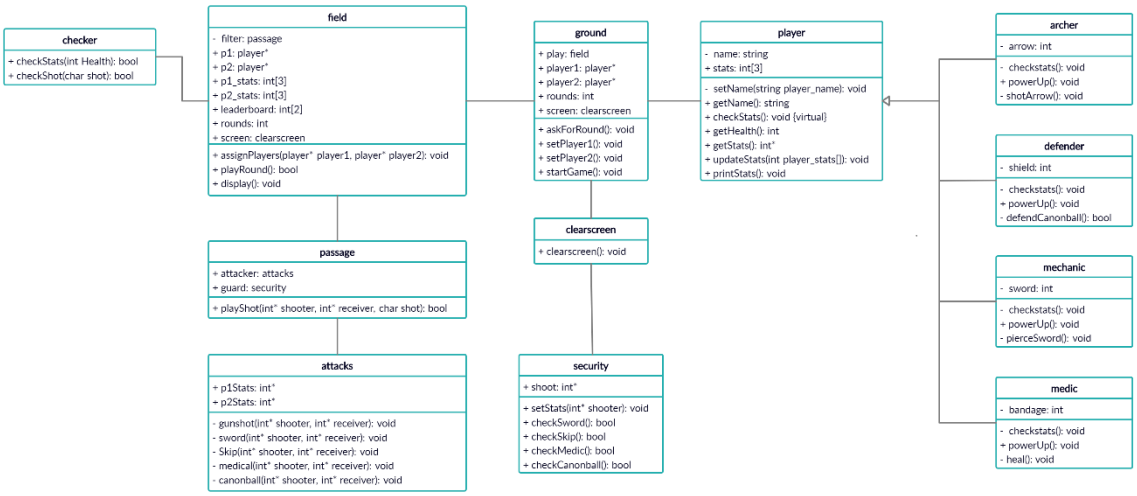
5. Use of the Concept

- ⇨ **Memory allocation from the stack and the heap**
  - The two integer stats arrays of size 3 (of the two players) are stored as dynamic memory. It makes the work a whole lot easier, as I have to just allocate the memory once, and then pass along the pointers of those arrays.

- ⇨ **User Input and Output**
  - At the start of the game, user is prompted to enter number of rounds, is then allowed to select between four-character types, and then enter the name for the player
  - In each round, each player can enter one of the five keywords allowed to fire (‘g’, ‘s’, ‘S’, ‘m’ & ‘c’). If any incorrect output is displayed, then the program asks the user to enter it again.
  - After every attack being fired, the stats are updated real-time according to the fire used. Also, in each round the keyword for the attacks, and the damages they do are always displayed
  - The output terminal window is entirely colour coded in different colours to make it more impressive. Also, the leader board is updated and displayed after each round has been played.

⇨ **Object-oriented programming and design**

- Class-diagram explains a bit much about the design and how the program is structured



- **Abstract Class:** the player class has been made purely virtual, making it impossible to declare it as an object. checkStats() behaviour has been purely virtual, which is then used by their children classes. Children classes inherit from the parent class, while having individual behaviours that come in action when specifically called or allocated by the user.

⇨ **Testing**

⇨ **Test inputs and expected outputs:** every possible boundary case have been tested using input from a .txt file and checking the output with .txt file that has already been written as expected output. For an example, my program stats cannot be negative or more than 100, therefore, any where if an argument consisting of these stats is passed, then it has been already tested to have it in the way it is required.

➔ **Unit Testing:** Each class file has been tested individually by creating their object files, and checking every states and behaviours.

- **attacks, security, checker, passage:** Testing was similar for the three classes. For an example, in attacks class five behaviours are tested with various input arguments to check if the behaviour as it was required to. Two arrays of size 3 were passed as arguments. Input test ./txt file contained 20 arrays to take in as input to check

every behaviour twice. Also, boundary cases were checked to make sure the efficiency stays.

- **player, archer, defender, medic, mechanic:** As player is the parent class, and the other four class inherit from player class, unit testing was applied on player class and children classes were checked by using polymorphism in the player test file.
- **field, ground:** these classes were tested to simulate the playing of the entire game, as it consisted of objects of every other classed. Also, incorrect inputs were entered to check if the program can display error messages.

➔ **Automated Testing:** Every unit test file was automated. Each input .txt was randomly generated by a different .cpp file according to the requirements of the test file. An output file was manually created to test the output with this expected output to test if it passed. If it passes, then it means that the class file passed every test case including error messages and boundary cases. (| diff – was used to check with expected output)

➔ **Integrated Testing:** Integrated testing has been implemented in the unit testing of the field class and the ground class. As these classes, require inputs that already simulates the real user entering the desired input, therefore, checking objects of every other classes included in the field and ground object files.

➔ **Regression Testing:** During the entire process, regression testing was implemented. Every class files were checked before connecting to other files, and the main program was run each time an update is made or a new class file is implemented.