ESE 224 - Final project

Text-Based Adventure Game

1 Introduction

This is the final project for this semester. This project is a continuation of your midterm project that you completed. This project will include advanced topics that you have learned.

2 Teamwork

You should complete this project with the same group that you started with. You should not have to change your group unless a TA has told you to switch out. If you have changed groups, then please make this clear in your group report.

In the report you will have to detail the work done by each student. Each group should only submit one copy with a list of names of every member in the group report file name.

3 Design Tasks

Your task for this project is to apply the advanced data structure knowledge that you have learned.

Project Objectives

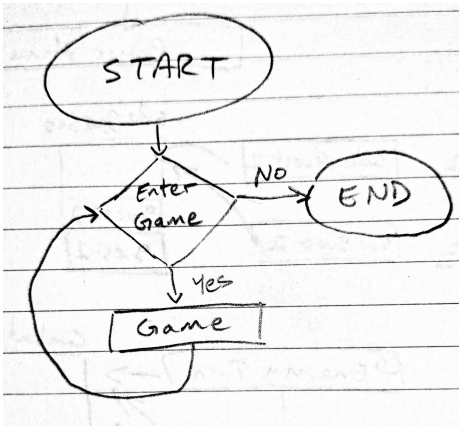
- Practice analyzing and debugging techniques.
- Develop good coding habits:
  - Use separate code files for the declarations and implementations of different classes.
  - Use techniques to make the code concise and modular.
  - Use comments where necessary.
  - Use proper indentation.
- Read/write data from/to files.
- Basic interface design with user selections.
- Implement object-oriented programming concepts such as inheritance, overloading.
- Work with a team

4 Coding / Code Design

General Tasks:

There are a few differences between the final and midterm.

In the final, the game will be executed not only once, unless one wants to stop playing and exit. The game will continue. After completing a game, the user will be given an option to play again.



The game will be similar to the midterm, but there will change a few things to advanced data structures. These changes will be introduced later as tasks.

The program will consist of two main parts: A master program to manage the whole program and a game program to manage each specific game.

Master Program

In this master program you must first get the scoreboard ready, then display the menu, then start the game, repeat the past two steps if necessary, and finally end the program.

At the start of the program:

1. Move the scoreboard information from the scoreboard file to the list.
2. Sort the list
3. Display the menu

Before this master program, display this menu. **This menu is different from the game menu.**

```
Main Menu:
(a) Scoreboard sorted by score
(b) Scoreboard sorted by name
(c) Averaged scoreboard
(d) Search by name
(e) Enter game
(f) Quit
>
```

This menu provides a list of functions to call before playing the game. One of the functions will begin the game. After the game ends, return to this menu so that the player can play again.

Before the game begins, display a menu with the following options:

A. Display scoreboard sorted by score.
Sort the scoreboard list by score and then output the scoreboard list.

B. Display scoreboard sorted by name.
Sort the scoreboard list by name and then output the scoreboard list.

C. Display averaged scoreboard.
Search the scoreboard list for duplicate names and then average the scores. Create a new list and then sort the averaged scores by score. Output the averaged scoreboard list.

D. Search for a name and display scores of that name.
Search the scoreboard list for a user inputted name and then display the scores of that name.

E. Give the option to begin playing the game.
This option should lead to the game created in the midterm with the changes added.

F. Give the option to quit out of the program loop.
Since the program loops now, this option should exit from the loop and close the program.

Begin the game or quit the program.

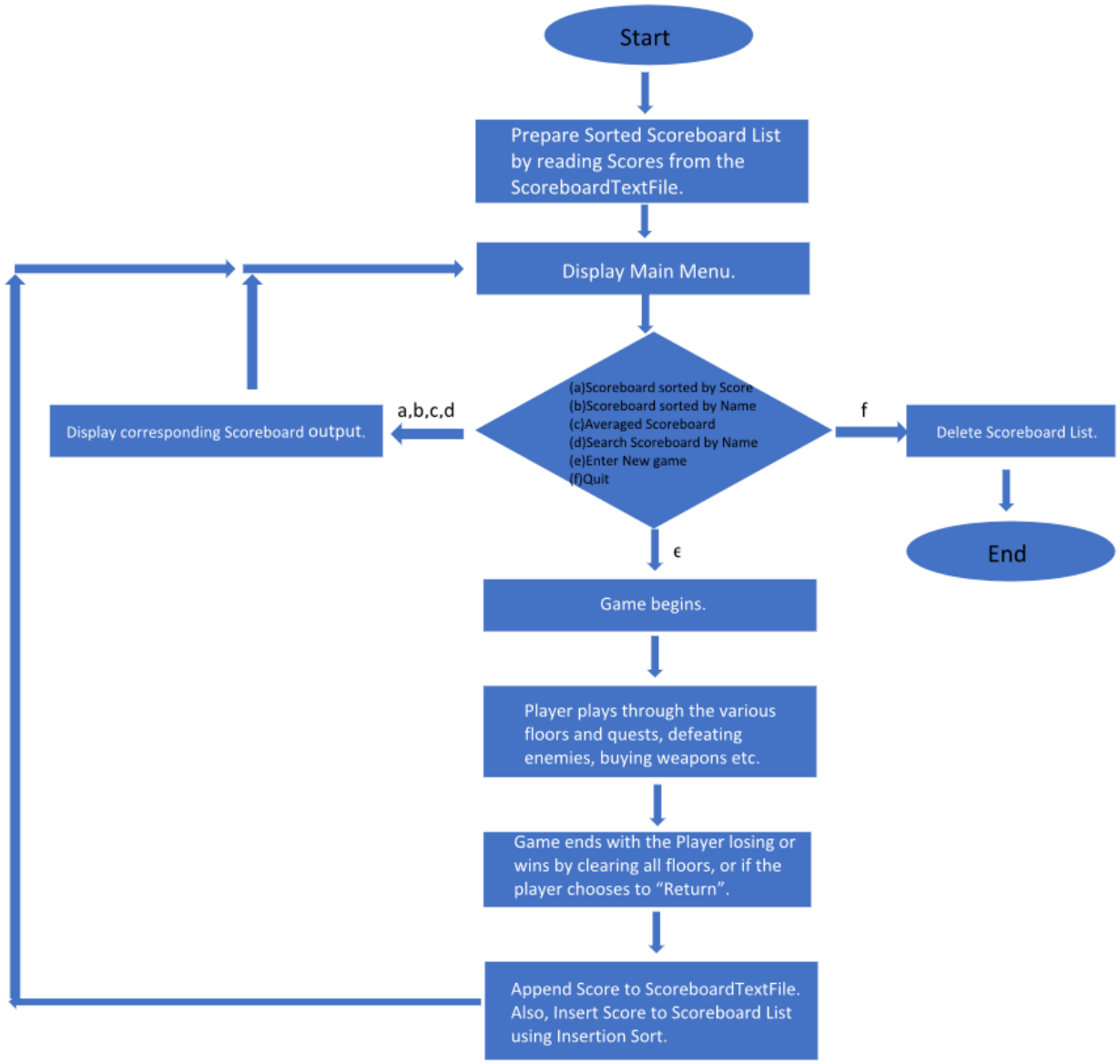
At the end of every game:

4. Append the scoreboard file, add the score from the Game to the file.
5. Start a new game or end the program.

At the end of the program:

6. Quit out of the Game loop. Delete the scoreboard list.

At the end of the Game the scoreboard should **NOT** be deleted. The scoreboard list is only deleted if the user decides to exit from the entire program.
Scores added to the scoreboard must be added through insertion sort. This will be easier to handle scores in the scoreboard list, rather than resorting the entire list.



Game Program

The game should work exactly like it did the midterm project, aside from the advanced data structure topics that were added in this final project. In the midterm, you created a player options menu to interact with the player information and scores. Fundamentally, this will not change. You also designed how the game will be played. The main design has also not changed too much either, as the game still revolves around travelling through floors and fighting enemies.

In summary, the in-game menu will stay the same. **Only now the quit option should return the user to the main menu.**

```
What would you like to do?
[E]xamine Self, [C]ontinue, [B]uy from Shop, [S]coreboard, [Q]uit
>
```

```
What would you like to do?                                    [R] Return
[E]xamine Self, [C]ontinue, [B]uy from Shop, [S]coreboard, [Q]uit
>
```

After the game ends, everything should be cleared and the menu should appear asking if the player wants to play again. The game should continue like in the midterm, but with the changes made in the Final.

Data Structure Tasks:

Task 0

In the player class, operator overload the “operator<<” to print everything. Use this to output to the file.

For example:

```
friend ostream& operator <<(ostream& os, const player& p);
```

Turn the enemy class into a derived class of Player Class. This means that Player Class is a base class. This uses the idea of inheritance.

The enemy class will gain all of the member variables and member functions from the player class. You will need to add the member variable level and the member functions to get and set the level variable.

For example:

```
class enemy: public player
```

Operator overloading for the “operator<<” should also be added to the enemy class.

For example:

```
friend ostream& operator <<(ostream& os, const enemy& e);
```

Task 1

Scoreboard list will be split into two programs. Use in the master program. This should be added in the beginning of the main program, before the game.

Create a list data structure to store the scoreboard data (name and score) from the scoreboard file. Use the list class template. (i.e. Use the list header file from the C++ library) This will replace the vector sorting from the midterm.

You will need to create some additional functions to use the list class template.

1. Create a function to read the scoreboard file and to then create scoreboard objects for each player score in the scoreboard file. This function should load all the objects into a list.
2. Create a function to ask the user for a name and to then search for that name in the list. All players with that name should be outputted with their scores to the console screen. Use any search method that you prefer.

3. Create an operator overload of the operator<< to print the names and score of all the objects in the entire list.
4. Use the clear member function from the list class template to delete everything from the list. Call this function at the end of your program.

Task 2
Scoreboard sorting and search will be split into two programs. The following sort functions will be used in the master program and will be added into the beginning of the main program.

Using the list class template, create two sort functions.

1. Create a sort function to sort the list by score.

To sort the list with the default sort() function, which accepts no arguments, you will need to use operator overload for the " operator< " inside the scoreboard class.

Once the operator overload has been set to compare the member variable scores between each other, you can now use the default sort() function from the list class template to sort the scoreboard.

2. Create a sort function to sort the list by name.

Add another overload to the sort() function that accepts a function pointer as an argument and use that for comparison while sorting. This will allow you to use the sort() function for two different cases.

Define a comparator or function object that will compare two objects based on their name. You can create a struct that has an operator overload that compares object names. Then when you call the sort() function with the function object or comparator as a parameter, you will be able to use the sort () function for a different member variable.

Hint:

```
struct ScoreboardComparator
bool operator ()(const scoreboard & player1, const scoreboard & player2)

sort(ScorebordComparator)
```

NOTE: If you want to use a lambda function as a parameter, then that will be accepted as well. But you **must** overload the sort() function in the list class template.

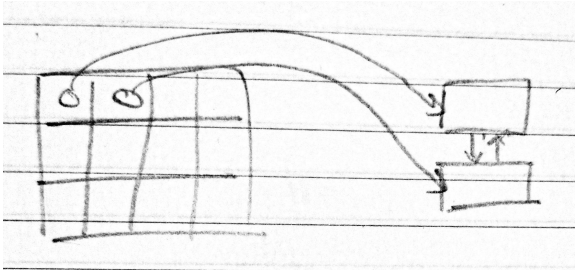
The following function will be used at the end of the game program:
Create an insertion sort function to insert the player score into the right spot in the scoreboard file.

Task 3
Use this in the master program.
Create a new list for averaged scores from the scoreboard.

Create a function to search the whole scoreboard list created previously for any names that repeat. If any names repeat, then take the average of all the scores. Then add this averaged score to the new scoreboard.
Insert the averaged score into the proper place, so that the list is sorted already from highest to lowest. If a name doesn't repeat, then also add that score to the list as well.

Task 4
Use this in the game program.

At the beginning of a game, create the shop list for weapons available to purchase. In the midterm project you made a 2D matrix for all your weapons as well as a shop to purchase these weapons. Modify the shop so that it is a double-linked list. You will need to make nodes and iterators. Your linked list will keep growing bigger after each floor.



At the start of each floor the shop linked list will need to be updated with new weapons. Create a function to add weapons onto the shop list depending on the floor.

Players will need to be able to buy weapons from the list. Create a function to remove weapons from the list. The weapon should then be given to the player.

When looking at the shop, players should see all the available weapons. Create a print function to display all the weapons in the shop list.

When the player leaves the game you will need to clear the entire shop list. Create a clear function, so all the nodes are deleted. (call this at the end of the program to prevent dangling pointers)

```
Shop:
    Weapon Name:     Weapon Damage:     Weapon Cost:
[1] Weapon1          10                 5
[2] Weapon2          15                 10
[3] Weapon3          20                 15
>
```

Task 5
Use this in the game program.

Add a shop tracker. After changing the shop, create a stack to keep track of all the weapons bought from the shop. Once a player leaves the shop, the stack should be outputted. This reminds the player about their most recent purchases. This stack should use the class template.

- Create an insert function to add to the stack
- Create a print function to output the entire stack. Output from the top to bottom.
- Create a clear function to delete the entire stack (call this at the end of the program to prevent dangling pointers)

```
Shop Tracker:
Recently bought weapons:
Weapon2
Weapon1
```

Task 6

Use this in the game program.

Create a stack that holds the enemies defeated. Once an enemy is defeated, add that enemy object onto a stack. You should use the stack class template. (i.e. Use the list header file from the C++ library).

Before the player leaves, output all the enemy names from the stack to show what the player the enemies they defeated while playing.

You will need to create some additional functions to use the stack class template.

1. Create an insert function to add enemies to the top of the stack.
2. Create an operator overload of the operator<< to print the names and level of all the enemy objects in the entire stack.
3. Create a function to clear the whole stack. This function should be called at the end of the program.

```
Enemies defeated this game:
Goblin
Troll
Orc
```

Task 7

Use this in the game program.

Create side-quests for the player to complete. Once the player completes this side-quest, give the player bonus points to be added onto the player's score.

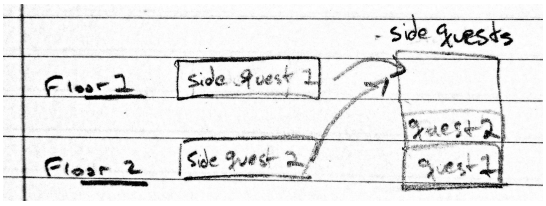
This side-quest system should be implemented like so:

1. Ask the player to do a simple task at the start of a floor.
    a. Save up a certain amount of currency. (e.g. Save 100 coins)
    b. Don't buy a new weapon. Defeat the enemy only using your free weapon.
    c. End a battle with more than 50% health.

d. Have attack damage higher than your current health.

The following are examples, but you are allowed to have side-quests different from the one suggested here.

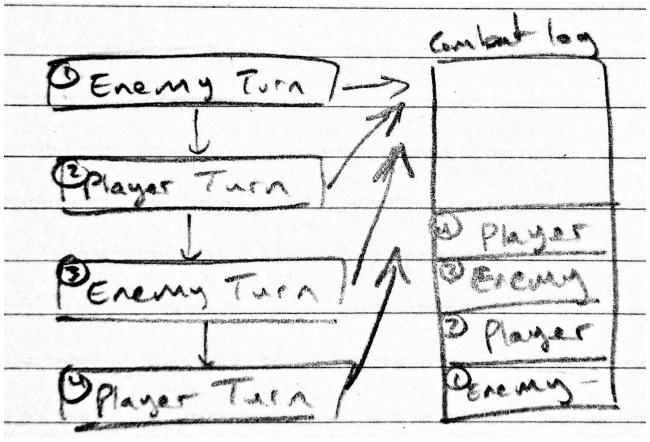
2. Check at the end of the floor if the side-quest was completed.
3. if the side-quest was completed, then add the bonus points to the score.
4. Create a queue. For every side-quest completed add a string phrase that notes what the side-quests that were completed.
5. Before the player exits the game, print out the phrases in the queue to show what side-quests the player completed.

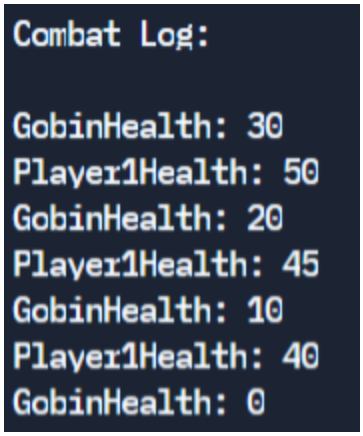


Task 8

Use this in the game program.

Create a combat log. At the start of a combat phase create a stack that will keep track of the health of both the enemy and the player. After the enemy phase this stack should be updated. After the player phase this stack should be updated again. Once the combat phase is over, the stack should be outputted and then deleted





Task 9
Use this in the game program.
Change how players fight your enemies.

1. Create a queue using the class template with the enemies objects
2. Before combat, call an enemy from the queue.
3. After combat, remove the enemy from the queue.

BONUS:
You can earn a maximum of 10 points bonus. Choose the bonus task.

Bonus (10 points)
Keep the mid-term structure and sort the scoreboard vector using heap sort. This function will sort the scores from the scoreboard.

Bonus (10 points)
Use a binary search tree to track the score board. In the binary search tree, you have three optional orders of operation. (In-order, Pre-order, and Post-order Implementation). Have the option to print the whole tree and to search the tree based on scores. Also, add in a search function to find the name of a user.

5 Code
TAs will run your code and then test for any errors. Your project code should have the following qualities:

• All codes should be completed by yourselves. TAs will read your codes, plagiarizing work from other groups or other resources is forbidden.

• Make sure your codes can be compiled successfully before you submit them.

• Your codes should be concise and modular. All files, including .cpp, .h and all data files, should be submitted in a project folder, and all codes in one main file is not recommended.

• Provide comments in your codes where necessary. This will make understanding your code easier for TAs when grading.

• Include a readme file to explain how everything works.

You may send a zip file of your entire project, but that is not necessary. What is necessary is that the project must have all the .cpp files and .h files, and any other additional data files needed to compile your code.

Sending the entire project may be the easiest way to check your code as well as the easiest way to avoid any missing files.

Take screenshots of your console demonstrating your code works.

6 Report

TAs will read your report and then check your codes accordingly. Your project report should describe clearly:

• Architecture of your project and the explanation of each section it has. Please explain how some of the functions/classes work. Explain the designing logic behind the codes.

• Specification of classes, including the attributes and functions of each class.

• System functions you implement, including how parameters are passed when each function is called.

• All the highlights of your projects. Some of the students may have some features that are not exactly the same as the requirements from the project and they think they are good to be included. That is fine, but you need to explain well.

• Important: always explain why you code like that.

• Detail of the work each student has done. You should explain which section of code you were responsible for.

• Notice the system you are using (Windows, Mac, MS, or xcode).

7 Grading

The grading scale of the first submission is:

• Code (70 points)
• Report (30 points)
• Bonus (10 points)

	Excellent	Average	Poor	Points

Code	All of the requested functions are implemented and work as they should. No elements are missing. The code is compiled with no errors.	Most of the requested functions are implemented and work as they should. Occasionally there are errors that were not fixed in debugging.	There are missing elements. Only a few functions are implemented. There are errors throughout the code.	70
Report	All work is documented and explained.	Some of the work is documented. There are missing explanations of the work.	Very little work is documented. Explanations of what was done are missing.	30
				Total: 100