University of Nottingham
UK | CHINA | MALAYSIA

EEEE2065 - **Lab Assessment 3**

Programming an adventure game

Total marks: 100

In this lab assessment you are to create `.cpp` programs that accomplish the tasks described in the problem statement below. This lab assessment is made available to the students from Moodle on Wednesday Dec 7th and is due back uploaded on Moodle by 5pm, Monday Dec 19th. You must use the OOP programming paradigm and classes in this assessment. This assessment is worth 10% of your final grade.

Marking Rubric

- 0-25%: Codes developed but did not compile properly.
- 26-50%: Codes developed and compiled, but not achieving expected result, programs are buggy or inefficient.
- 51-75%: Codes developed and compiled with correct bug-free and efficient results but modularity/clarity/commenting practices are not well followed.
- 76-100%: Codes developed and compiled using taught best practice standards, with clear, well-commented code and modularity rules being demonstrated/followed.

Plagiarism statement: The codes you submit are to be developed by the student alone. Any codes which are **significantly** similar to each other, or to code found on the internet will be penalized as plagiarism and points will be deducted accordingly.

1 Problem statement: programming an adventure game

So far in our practical sessions, we have been developing the basics of an adventure game for a 4-room world. In this assessment, you are to use the skills and knowledge you have acquired in the practicals to build a larger adventure game world with some additional features that you will need to add yourself. The map for the adventure game to be created is as shown in Figure 1.

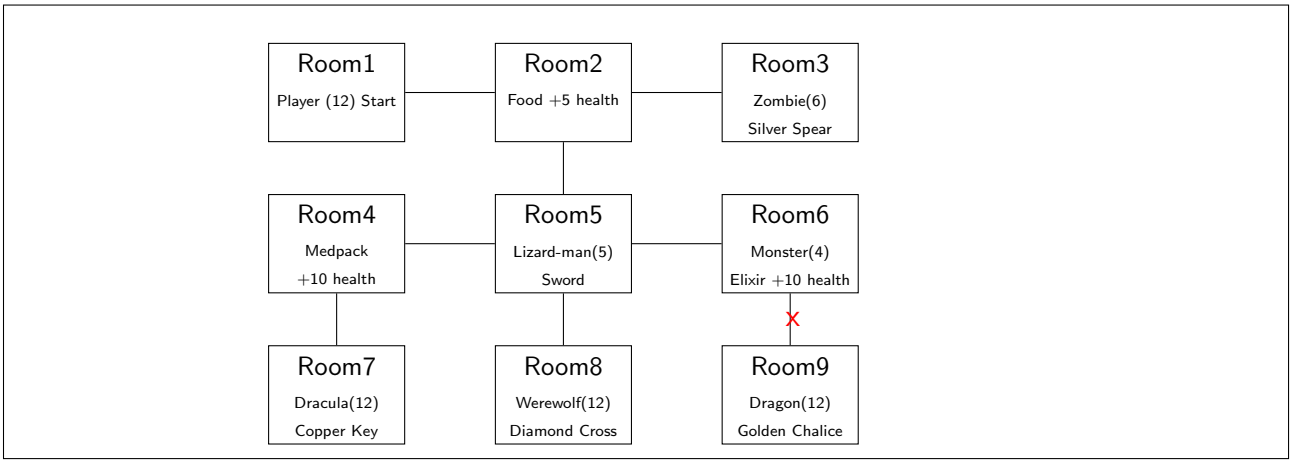

Figure 1: The map for the adventure game, x denotes a locked door.

The player starts in room1 with an initial health of 12. Monsters exist in various rooms with their healths shown in brackets in Figure 1. The player moves around the map in the adventure game, fighting monsters and collecting/using objects and artifacts. The adventure game should include the following general traits and features

1. It implements the map depicted in Figure 1. Choose appropriate descriptions and names for the rooms in your world.

2. There are multiple enemies with various strengths and weaknesses in the various locations depicted in the map

3. The player and the enemies now have health status number (HP) rather than just alive/dead. The player can also die and a score (or XP points) for the player should

be recorded. When the player dies, wins or exits the game, his score is reported before the program exits.

4. There are now artifacts and weapons added to the game that the player can use or gives the player certain advantages during the game-play.

5. Include a `help` command. When typed, a list of instructions telling the available commands known by the program is displayed.

6. The final goal of the game is for the player to get the Golden Chalice and exit the game by returning to the starting room with it. The player will then receive a congratulatory message and his final score (made up of treasures and monsters they have killed along the way) is displayed.

2 Expectation on details of the program/game

2.1 The main program calling routine

You are to program the adventure game world using the techniques shown to you in the practical sessions. In particular, the main calling line that interfaces between the user and the rest of the program should resemble the following:

1
```
int main() {
  adventureGame ag; // class for our adventure game.
  cout << "Welcome to Adventure Game!\n";
  while (ag.getCommand()>0) { // get a command from the user
    if (ag.processCommand()==0) break; // process the command given by the user.
  }
}
```

This method of calling the program will be used in the evaluation process and thus you should follow this code. Note: the call to the `processCommand()` function may be modified to include additional variables that you may need to pass to the function, so this code is just a template, but the general method to call the `processCommand()` function shown in code-block 1 above should be followed.

2.2 Monsters health, player (HP), experience (XP) & player can die

Each monster should be assigned a health or HP. The player should also be assigned a health or HP and experience XP. Previously the player only killed the monsters, in this assessment, you need to give the monsters the ability to fight back and if the players HP drops to zero, he dies and the game ends. During the combat between the monster and player, the health/HP of each side should be reduced using an algorithm that you come up with yourself. You should create the algorithm such that the player has a good chance to complete the mission. You can choose to include some randomness with a random number generator into the fighting subroutine. The player also gains XP when killing monsters or getting artifacts. These XP will determine the players score which is reported at the end. You may choose the XP to assign for killing monsters/getting artifacts yourself, but keep the numbers reasonable. The player should be updated on the status of the monster as well as of his own health throughout the course of each battle.

The player can increase his fighting ability with weapons and artifacts to help him defeat some enemies. In particular:

1. Having the sword increases the damage the player inflicts on most monsters
2. The player can regenerate health by eating food, using the medpack or drinking the healing elixir (back up to some maximum value that you should choose).
3. The werewolf is injured much more during fighting when using the silver spear
4. The sword has minimal impact on the vampires (although it is better than without it). They can be injured more only when the player is carrying or using the Diamond cross.
5. The Zombie prevents you from picking up the Silver Spear.
6. Dracula prevents you from picking up the copper key.
7. The Werewolf prevents you from taking the diamond cross.
8. The Dragon prevents you from taking the Golden Chalice.

2.3 Player can carry (at most 3) objects

There are various artifacts or objects in the game that the player can pick up and use. The following are things that should be programmed with respect to the objects:

1. Some of the objects are weapons that increase damage the player inflicts, or target specific monsters as described above. Examples are the sword, silver spear and diamond cross.

2. Some of the objects restore player hit-points such as the food, medpack and elixir. The player must `eat food`, `drink elixir` and `use medpack` in order to restore HP. Once used, the `food/elixir/medpack` are consumed and disappear from the game.

3. The copper key is needed to unlock the door to room9.

As in most adventure games, there is a limit to the number of objects the player can carry, the maximum number is 3.

2.4 Additional expectations on game features

1. If the player just presses enter without typing a command, the last command is automatically repeated. This will help during fighting when multiple `"killMonster"` commands need to be executed in succession.

2. All text prints out and formats nicely like a proper adventure game (no unsightly line-breaks or unwanted characters appear).

3. If the player tries to go to an illegal direction, he is informed as such

4. If the player tries to pick up something that is not there he is informed as such

5. If the player tries to kill monsters that are not there, he is informed as such

6. The monsters fight back.

7. Having appropriate weapon gives player an advantage in certain battles

8. Player has a record on HP, XP and can die
9. Player can heal with food/elixir/medpack up to a certain maximum
10. Keys are needed to access certain rooms
11. Monsters prevent player from picking up certain items

2.5 List of commands

Here is given a list of the commands that are expected to be known by the program:

Command	Example(s)
Movement commands	`north, n, south, s, east, e, west, w`
Look command	`look,l`
Fight commands	`killMonster,killmonster,km`
Alternate	`kill zombie, kill werewolf, kill dragon`
Get commands	`get diamond cross`, `get sword`, `get Sword`, `get elixir`
Drop commands	`drop copper key`, `drop food`, `drop golden chalice`.
Inventory command	`inventory, i`
Healing commands	`eat food`, `drink elixir`, `use medpack`
Unlocking door	`unlock door`
Exit game	`exit`

Note: Have your program able to accept both forms of killing commands.

2.6 Report

Write up a report (5∼10 pages) detailing and documenting what you have done in this assignment, highlighting certain codes that implement some of the above-described features. Have your report focus on the parts of the assignment that you had to create for yourself, such as:

1. The fighting algorithm: how do you determine how many points are deducted from the monster and from the player? Is it deterministic or random? What formula do you use

in either case?

2. How does your code implement the advantage given to the player when using the correct weapon against the targeted monster?

3. How does your code implement the monster preventing the player from picking up objects if it is alive?

4. How does your code implement the locked door and check if you have the key?

5. How does your code implement the XP and HP?

6. How does your code implement the restoration of energy?

7. How does your code implement the players demise?

Include some sample game-play screenshots in your report to demonstrate the above-mentioned features.

2.7 Code/game evaluation and submission

In order to standardize the evaluation of your game, your program will be checked using a fixed set of commands testing how well/whether it follows the instructions given above, such as being able to pick up items, kill monsters, open doors, etc... The `processCommand()` function will be given a series of sets-of-commands to follow and your programs reaction to each set will be checked to see whether it is executing the expected behaviour. Therefore your program must be able to take commands in the format as described in code-block 1. There will be standard tests of each of the features described above, and your score will be determined by how the program responds to each test. Thus the necessity in the modularity of your program, separating the functions (written by you) from the calling environment that will be used to test your program becomes clear.

By the due date/time, you should submit just the following files on Moodle:

1. `main.c` file: the main program, contains code simliar to the template shown in code-block 1.

2. `CW3.c` file: contains the source code for all the functions used in your program to execute the adventure game

3. `CW3.h` file: the header file that is included in both the `main.c` and `CW3.c` files. Any function declarations in `CW3.h` will be visible/callable from any function that `#include"CW3.h"`.

4. `report.pdf` file: the report.

I will create a set of `main.c` files that will execute a series of commands to test different aspects of your program.

In addition, the program will be checked for similarity with the programs of your classmates through a similarity-checking software to check for excessive duplicated code. Points will be deducted from both parties for duplicate code, and in severe situations, the case may be raised for further disciplinary actions. The exception will be for duplicated code that is given to you in the practicals. Each student is expected to come up with their own room descriptions to differentiate themselves.