RPG Quest Log

Name:_____

Total Rupees:_____/300

General Tips

- During or after completing each core quest it would be VERY wise to refactor your codebase to ensure that everything that can be a function is a function, and every object that exists in-game corresponds to either a class or instance of a class in your codebase.
- Before making any changes to your codebase, create a branch on github and work on the branch. Once your changes are good, merge your new code with the old.
- Note that points on this quest log should NOT be treated like points on standard quest logs. The assignment categories are different, and the points associated with these quests are worth more in terms of overall grade. Hence why they're called rupees instead.

Core Quest 0: Getting Set Up for Success – The Tester's Menu

[Reward: 10 rupees]

There are many features that will be added to this game, such as an in-game shop menu, a battle sequence, inventory management, dialogue trees, and a medley of other things you will decide on. To keep your code modular and easy to test, this quest has you build a simple menu listing all the things you will eventually be able to do in game. Later in the semester once you have built your game overworld, this menu will morph into your in-game menu (like what you'd get when you press "pause"), with most of the options you use for testing getting removed.

Create a menu that displays the moment your game runs that lists the following options:

1. Return to Game
2. Enter Battle
3. Manage Inventory
4. Enter Shop

5. Talk to Someone
6. Quit Game

After the user enters the number corresponding to their choice, the following functions should run:

1. returnToOverworld()
2. enterBattle()
3. manageInventory();
4. enterShop();
5. chat();
6. quitGame();

For now, the implementation of each function should do nothing more than print out what choice was made and then return to the main menu until the user selects "quit". For now, these functions and your main() can be implemented all in the same file.

Show the instructor your codebase once this is done to earn a stamp.

Core Quest 1: Battle Sequence!

[Reward: 80 rupees]

Check out the RPGDemo.exe in the dropbox for a model of what this part of the project should be doing.

Build a turn-based battle sequence where it is the player versus an enemy to replace your current "enterBattle()" main menu function.

Front-End Requirements:

- One of (at least) three possible enemies should appear in the battle sequence. Which one appears is based on a dice roll (random number generator).
- Must display decision-based stats of the player (i.e. stats that can change in battle, such as health, magicka, stamina, etc)
    - The player must have at least three battle stats
- Must display health status of enemy. Other stats display is optional. HOWEVER! The enemy must have at least three stats

that are used in battle (again, such as magicka, stamina, health, etc.)
- The battle sequence must alternate turns until either the player or enemy either runs out of health or runs away.
- All outputs ABSOLUTELY MUST BE TIDY AND READABLE
- Anytime an action takes place, the screen should appropriately refresh and display relevant information for the next action choice.

Back-End Requirements:

- Inheritance and polymorphism must be used to handle the creation of different enemies. All enemy types must come from a base class called "enemy." Each enemy type must have their own different implementations for at least one base class function.
    - Suggestion: Have a base class called "character" from which you can derive both a player class and an enemy class, which can then be used to derive yet more specific classes.
- The enemy action engine must be probabilistic, in that different actions are chosen at random based on a weighted probability scheme (e.g. 40% likely to attack, 20% likely to run, 30% likely to use heals, etc).
- In all things, stay as modular as possible.
    - If different objects do the same thing, consider inheritance, and/or functions.
    - If the game or player must make decisions, consider using functions with appropriate inputs and outputs… and good naming based on what the function does.
    - Adding new features ideally should be not too much harder than just adding new functions and classes in different locations and doing a bit of copy-paste. Make this a goal at all times.
- Codebase must adhere to the C++ Style Standard (see Canvas Course Info under Modules for this)

- Implementation of all functions and classes must be separated from their prototypes using separate headers and .cpp whenever possible.
- ALL functions and classes must be properly documented. This means that the inputs, outputs and a brief description of what the function does should be above where the function is defined.

Tips

- There are many features you can add to this battle sequence. When developing it, try to keep everything as modular and function-based as you can, so that (ideally) new features can be added simply by dropping a new function call in the correct place.
- Have foresight about what features you might want to add later. This will help you think through how you can keep things modular and self-contained.

Core Quest 2: In-Game Inventory

[Reward: 20 rupees]

Create an in-game inventory. Make it possible for your character to add and remove things from their inventory OUTSIDE of battle using a menu. The player's inventory should be managed using a linked data structure (SLL, DLL, Queue, Stack, or even a hash table (useful for some features)). How the game goes about adding or removing an inventory item is up to you and will be determined later as the game is developed. For now, it is sufficient to simply write a function that adds stuff to the inventory and another that removes something from the inventory. Note that it should be possible to have multiple of the same item in your inventory. It is up to you if you want to maintain a weight cap or carry infinite items.

For the sake of this quest, there is no requirement that the inventory be accessible in battle sequences. However inventory "equipped" outside of battle should be reflected in battle. For example: if you equip a sword as your primary weapon outside of battle

Implementation Recommendations:

- Create an "inventoryItem" class and instantiate a few different items that can be selected from and added and removed from your inventory to test that the add and remove functions work correctly. Both of such functions should take an item object as input (unless you have in mind a completely different functionality for how things will be added or removed in your game).
- Consider carefully which features you want to add that involve your inventory menu.

Hint (if you dare): Perhaps the easiest way to handle an inventory is to have a list or hash table housing every item that it is possible to obtain in the game, along with the total number of each item you have (most items will have a zero). Then, your inventory could simply display the items that have non-zero values beside them. This also makes it easy to add new obtainables to the game (esp if you are using a linked data structure for the set of all possible items). It might even be good to have a generic "item index" that works in the same way, built so that all NPCs in addition to the player can use it. This way, all an inventory for ANY character/NPC/shop is is nothing more than a count of the number of each item they have.

Core Quest 3: In-Game Shop!

[Reward: 40 rupees]

Create an in-game shop to replace the main menu's enterShop() function. There should be minor dialogue between the player and the shopkeeper, and allow the player to:

- Purchase new items to be added to inventory
- Sell items in inventory for Gold (or whatever currency exists in your game).
    - This means that most, if not all, items in your inventory must have a gold value.

Your shop should have a mechanism for allowing several items of the same kind to be purchased at the same time (such as "I want 200 potions")

Naturally, you'd want it so that you cannot buy more products than you have gold or sell more things than you have. Display appropriate messages.

Furthermore, the in-game shop must be implemented using one of the data structures we've discussed (i.e. not just a vector). E.g. you might want to use a queue the full list of items (or various submenus). You do you.

To earn a stamp for this quest, show the instructor your functioning store AND show that your inventory has changed in some way after leaving the shop.

Recommendations

- Add a "gold" member to your player class. This will be massively useful with a bunch of different features you can add later.

Core Quest 4: Dialogue Tree

[Reward: 40 rupees]

The purpose of this quest is to exhibit how trees can be implemented quickly by combining them with the instant access time of arrays. That is, we will use arrays to implement our dialogue tree.

Dialogue in this game will work as follows:

- When an NPC that talks is "activated" in the game (i.e. you walk up to them and say "sup bruh"), they will print out a sentence or two of speech, followed by at least one or two responses that the player can choose. Once a choice is made, another dialogue from the NPC gets printed with yet more choices the player can make, and so on.

The idea for implementing this is simple:

1. Write a dialogue tree for the NPC and yourself on paper:
    a. The NPC says something, and you have choices of how to respond.
    b. Each choice you can make is associated with a branch of a tree that connects to the NPCs response to your choice
    c. Number every possible NPC dialogue on your paper tree, 0 through n, where n is however many things the NPC can ever say.
2. Store all NPC dialogues in a vector (or array), along with possible player response options to said NPC dialogue.
3. Dialogue starts by printing the first dialogue in the vector along with the associated player response options.
4. Once the player makes a choice from the response options, the game then goes to the location of the associated NPC followup dialogue in the vector (again, based on the choice made).
5. The NPC dialogue at that vector location is printed along with yet more choices, and so on.
6. This process continues until the player chooses to leave the dialogue or when there are no more responses for the player to choose (the convo ends naturally)

More of the nitty gritty details for how to make this happen. The classes, members, and methods are as follows:

Class **talkingNPC**: a base class class (perhaps derived from class Character) representing an NPC that you can engage in dialogue. All NPCs that you can talk to in-game will be classes that are derived from **talkingNPC**.

Members and Methods:

- **Name**: the character's name (which will remain printed throughout the entire convo, just above the NPC's current bit of dialogue)
- **dialogueTree<dialogueNode>:** a vector or array of dialogue nodes (more on these below).
- **printDialogue(int currentDialogue):** a method that

1. takes the index of the next dialogue node to be printed,
2. then prints the contents of **NPCDialogue** and **playerChoices** at that location in dialogueTree.
3. Then, gets input from the user to choose an option from the **playerChoices** list (as a number, 0 through 2,3 or however many options there are).
4. Then, based on the number selected, goes to that location in the vector **currentDialogue.nextDialogues** to get the associated array/vector location of the next dialogue in the dialogueTree.
5. Lastly, calls the **printDialogue** function with arguments **dialogueTree** and the location of the next dialogue as found in the previous step.
    - Yes, this is technically recursion, and loads up the program stack perhaps unnecessarily. Go with it for now.
6. (Optional and useful for later), consider where other actions aside from speech can be added to this dialogue such as initiating a battle with this NPC,

- **talkingNPC(int treeSize)**: a constructor that does the following:
    1. Initializes dialogueTree to have treeSize number of slots (filled in with NULL for now).

Now, when you create a talking NPC, it will derive from the talkingNPC class. Suppose our talking NPC is called roadsideBeggar. Then, we build a class called roadsideBeggar, inherit from talkingNPC, and build a constructor that first calls the talkingNPC class constructor, then does the following:

1. For each dialogue node in roadsideBeggar's dialogue tree, instantiate a dialogueNode, passing the constructor for said node the string roadsideBeggar will say, the player's response choices, and a vector/array of integers corresponding to the indices of each of the next bits of dialogue in the dialogueTree (matched up with the choices the player can choose).

    a. All of the actual dialogue stuff is hardcoded into roadsideBeggar's constructor. The roadsideBeggar's constructor only will take the size of the dialogue tree as an argument, which is immediately passed to the talkingNPC constructor.
2. Then, add the dialogueNode you just created to the correct location in the dialogueTree

WARNING!!! If you put your dialogueNodes in the wrong location in your dialogueTree, or if your nextDialogues in any node refer to the wrong indices in your tree, you will be VERY confused as to why the NPC isn't responding to your choices correctly… or get a happy little seg fault. Make sure your dialogue node locations match the indices in your tree drawing.

Class **dialogueNode**: a node containing NPC dialogue and choices that can be made by the player

Members and Methods:

- **string NPCDialogue**: What the NPC says when this node is activated
- **string playerChoices**: a multiline string with each possible choice the npc can make
- **vector<int> nextDialogues**: a vector of integers, where each of the indices 0 through nextDialogues.size() corresponds to choices in the string **playerChoices.** The int values located at each of these locations correspond to the location of the NPC dialogue follow-up convo in the main dialogueTree.
- **dialogueNode(string npcTalk, string playerChoiceList, vector<int> nextList):** constructor that takes as input
    - a string npcTalk that holds what the NPC says at this node in the tree
    - a string playerChoiceList that holds the multiline output of choices for player dialogue in response to what the NPC says here.

- a vector (or array) of integers corresponding to the indices of the NPC dialogues in **dialogueTree** that follow from each of the choices the player can make after the dialogue at this node is printed.
- The constructor then saves these inputs to the dialogueNode members.

To complete the quest, create two or more characters derived from the talkingNPC class, where each has unique dialogue trees with at least 6 nodes each. The dialogue tree must have height at least 3. In your int main() function, instantiate these characters and start the dialogue tree for each, allowing the player to interact with these characters individually. Show the instructor your dialogue tree drawing for each character and your functioning conversations to earn a stamp!

Note: There is a picture on the dropbox to show you diagrammatically how this all fits together.

Core Quest 5: Design the Map and (optionally) Write The Story

[Reward: 20 rupees]

Design your map and write a story! Story doesn't need to be elaborate. All it needs to do is give context to your map and locations thereon.

It is recommended to build your map on a sheet of graph paper or something similar, where each cell corresponds to a visitable location. It is also recommended that you label/name each location with either a unique name (if the location is special, or different from ordinary locations, such as a bland path in the woods) or xy coordinates.

Later you will be building the map into your game such that each location on your map is represented by a node on a graph.

At each location, you will want to establish the following:

- What does the player "see" at that location?

    - Think of a succinct but sufficiently detailed description of what the player is seeing and experiencing at that location. Sights, sounds, smells, etc. This will be what the player reads when they enter that location on your map.
- What can the player do at that location?
    - E.g. can they talk to someone at that location? Can they enter a shop? Can they challenge someone to a battle? Can a conversation lead to being challenged in a battle?
    - What directions can they go next?
    - Can they pick up something and put it in their inventory?
- Is there a possibility of an encounter/battle as you move from or to that location?

Show the instructor your map to receive a stamp. Be prepared to share some of the details from the above list.

Core Quest 6: Building the Map Into Your Game

[Reward: 70 rupees]

Take the map you designed in the last quest and build it into your game!

You should do this via a graph object/class, where each node in the graph represents a map location. You can use the graph class built in one of the standard quest logs, or build a new type of graph that works better for this situation than the usual representation. In fact, the easiest way to do this is by creating a multi-directional linked list with "up, down, left and/or, right" pointers, and keep a global index that points to your current location and possibly the next location (after execLocation() runs (see below). Another easy option is to just use a matrix (i.e. an array of arrays) of map nodes, where certain locations are accessible while others are not.

It is also recommended (for the sake of organization) to make it so that each node has a "execLocation()" function that does the following:

- Prints what the player sees and experiences at that location
- Prints a menu of choices that appear for that location
- Then, based on what the player chooses, executes the associated action.
    - If up until now you've kept all things your game can do modularized as functions, all this will require is object instantiation (such as bringing a talking NPC to life) and executing different functions based on user choice.
    - ***The best way to go about executing the correct function based on user choice is to use a vector of functions, where the index of each function in the vector corresponds to a number that the user can enter.***
        - All functions in the vector need to have the same return type, so you will need to resolve this problem as well.
- At the end or beginning of the execLocation() function, run a function that plays a random battle encounter (if applicable) with a certain probability. 10-25% likelihood of an encounter is just enough for standard random encounters. Might be higher for more enemy-dense areas of your map.

One of the other challenges of this quest is how to go about implementing each node's execLocation() function without a massive amount of overloading, overriding, and inheriting for every node, given that each node can have vastly different things you can do. There is a way to do this neatly if you are clever about how you organize and/or pass the contents and actions of, from, and to each node, and how you build the constructor for each node. If you are especially clever, each node in your map need not exist in-game until you visit it. If you need hints for this, talk with the me.

Unfortunately, because each location is different, you will in some capacity need to hardcode in what happens at each node when building your map out within the game. Each node's creation can be handled by your map class's constructor. It will be the biggest constructor you write for a while… unless you can find a cleaner way of doing things, or dynamically create each node as you arrive at it. In some capacity

though, your game will have to store all relevant information for each node somewhere, even if all of it is dynamically loaded in when a node is visited.

Finally, change your "return to game" function from Quest 0 to start the game at the beginning node of your map (i.e. starts the game fresh).

Core Quest 7: Menu Changeups

[Reward: 20 rupees]

Note that this quest may still be completed even if the last one is not completed (if you're in a time crunch).

Convert your current start-of-game menu into a proper start-of-game menu by removing all testing options (because they should now be thoroughly tested and integrated into your game), and leaving behind options like "start new game," "continue," and "quit game."

Secondly, when exploring the overworld, at any node, one option should be to "view menu."

The in-game menu should allow you to edit your inventory, quit the game, in addition to any other features you wish to have (like fast travel, or saving your game features). Best to have a separate function that manages this menu for you so it can be seamlessly integrated into each node's selection options.