Ryan Tran 105384304

CS32 Project 3 -- Report

**The name of your recursive goblin movement function and the name of the file it's implemented in, or else a statement that your goblin movement function is not recursive. (Make this the first thing, to help the grader find the function.)**

The goblin recursive movement consists of two functions named **monstersMove**, which calls the recursive function **goblinSmell** that checks for a path to the player in the **Dungeon.cpp** file.

**A description of the design of your program. For each of your classes, indicate its purpose, what behaviors it implements, and how it relates to other classes. documentation of non-trivial algorithms. Use pseudocode where it helps clarify the presentation.**

The program is split into 11 files, 2 of which were given fully implemented (utilities.h and utilities.cpp), these files consist of randInt and clearScreen that is used in displaying the game and calculating random variables and probability in the game.

**(Game.cpp and Game.h)** First let’s talk about the game files. The main routine, we create a Game and play that game. This refers to the files Game.h and Game.cpp, which has class Game, and two different constructors for Game as well as a destructor for game, a play function, and a private variable of type Dungeon which functions as the CURRENT level of the game. For our empty Game Constructor, we call the constructor of a Dungeon and create a new one to initialize to our private variable m\_dungeon in game. Then we initialize the range in which Goblins can track a player to 15 using a dungeon function called setGoblinSmell. Then we have a do while loop, that gets random row-column coordinates for a player and places the player at that row-column spot in the dungeon if the spot is empty. To check if it is empty or not, we use a dungeon function and pointed from m\_dungeon called getCellStatus, and we pass in the random coordinates. If it doesn’t equal to a global constant integer variable called EMPTY, then we reroll the coordinates. Once it finds a non-empty spot, we place the player using an addPlayer function in dungeon that takes in the row-col coordinates. The game constructor that takes in a parameter for goblin’s tracking range would do the exact same thing except set the range to what we give it. Game’s destructor deletes m\_dungeon since we dynamically allocate it. Now that we know how a game is made, let’s see how we play it. We have a play() function that uses a function called display() in dungeon and outputs the information of the dungeon level. We then will be in a while loop that will only exit once the player’s hitPoints is 0 or a boolean variable called Win is true. Inside the while loop, we take in a user’s input and pass it in a switch statement that will lead to different actions and functions. Most of these are handled in a class player function in Actor.cpp, which will then call a dungeon function called monstersMove that will let monstersMove in the turn the player does so the game is able to run turn wise. Once we exit the while loop, it means the game is over so it outputs the result and an empty while loop waits until the user inputs q to exit.

**(Actors.cpp and Actors.h)** Here we set global const ints that represent the ASCII value of the arrow keys. This allows for the user to input arrow keys and let the player move instead of using h, j, k, l to do so. So first off, we have an Actor class with private variables that hold the stats and information of an actor; these are the following: Hitpoints, strength, dexterity, armor, sleeptime, row, column, weapon, dungeon, name. Each of these has a respective getterFunction and setterFunction except for the dungeon pointer variable which is only set when an actor is constructed. The weapon variable is also a pointer to type weapon in the GameObjects files since it can be modified. The actor constructor takes in parameters to all of its private variables EXCEPT weapon to initialize them. The virtual actor destructor deletes the weapon private variable that will be dynamically allocated in its derived class.

There is class Players which are made from the public actor and the constructor for player initializes with the respective stats using the player constructor, but takes a 3 parameters, that being a row, col, and a Dungeon\* that will link a player to a game level. Inside the constructor, we create a new ShortSword weapon and push it to the player’s inventory. The inventory is a list of gameObject pointers. The player also has a maxHP variable that is initialized to 20 but can be modified with a setMaxHP function. There is also a getter function for that and the inventory. The virtual destructor for player iterates through the inventory list and deletes the pointer and erases the items for all items EXCEPT for the currently wielded weapon (which is set to m\_weapon) since that will be deleted by the actor destructor. There is also a monster class that is derived from actor. Then monster has 4 derived classes for each separate monster. Their constructors passes in the same parameters as a player and initializes the respect string name, randomizes stats, and its weapon that is dynamically allocated. For Dragon, like player, it has a maxHP private variable and a setter and getter for it since it can heal.

In our Actors.cpp, we have two notable functions. First let’s handle the easier shorter function. This is called void **readScroll** and is of Actor type and takes in a scroll pointer. It uses the scroll input to get it’s name string and there are 4 if statements that will check for what scroll it is and carry out that respective effect. Inside readScroll, we set a player Pointer variable to the m\_dungeon’s getPlayer. m\_dungeon is a part of the actor and links the actors and dungeon game. Now that we have access to the dungeon’s player and a scroll passed in. We modify the stats or position of a player depending on the scroll input.

This readScroll will be called in the next function **playerMoves** which takes in a character parameter. This char parameter always takes in a getCharacter() aka the user’s input in our program. Like readScroll, it refers to the player in m\_dungeon. Here, we first have a if statement that checks if a player is asleep (their sleepTime is greater than 0), if so we want to decrement the sleep time using the setter function of the player and then do 3 things that happen on every turn: 1) Monsters turn movement called monstersMove() from Dungeon, 2) the chance that the player heals and 3) the Display board to update the board. And then exit the playerMove since sleep prevents any user action. After that sleep check, we have a switch statement that takes in the character input from the user, remember how earlier we set the constant for arrows earlier? We will use it here.

***PlayerMove pseudocode:*** *takes care of all the player inputs and works in conjunction with the switch statement in the play function*

|  |
| --- |
| check if asleep  see above for the functionality of that  switch on the keystroke of the user  case if the input is either one of the arrow keys, j, h, k, l:  if the next spot in the direction of input is empty, then move and update the board  if the next spot in the direction is a monster, then get the monster at that spot and pass the player and monster into the attack function (in dungeon)  break out of switch statement  case if input is c  cheat so set the hp and maxhp to 50 and strength to 9  break  case if input is g  pass in the player pointer to our pickUpItem function in dungeon  break  case if input is i  show the inventory using the showinventory function and getcharacter so player can close it with any keystroke then move the monsters and display the (update) board  return out of the entire function  case if w or r  these are the read or write functions so moveMonsters and update the board, the function to actually read and write are taken care of in the game.cpp’s switch statement  …  there it shows the inventory and getCharacters again for input and tries to either wield  or read an item based on the 2nd input  case if >  calls nextlevel function which checks if player is trying to descend on a valid staircase  also moves the monsters then updates the board using display  -- outside switch statement --  roll for the chance to heal the player  move the monsters using the monstersMove() in dungeon and update the board |

**(GameObjects.h and .cpp)** These handle all the game objects in the game so all weapons, scrolls, stairs, and the golden idol. Here we have constant int global variables so it is easier for any switch statements using items. We have an overall class called GameObject which has 4 private variables that are initialized in its constructor and has getter and setters for these variables. There is a string m\_name which holds the object's name used for output log. Then there's an action string that pertains to when a user uses that object. Then there's row and col. There is also a second constructor that only takes and initializes the name and action string. The destructor is empty for all objects.

Now there are 4 direct derived classes, the two main ones being weapons and scroll. Weapons hold a damage private variable and another for dexterity. These will be used in attack calculations. There are then 5 different derived weapon classes of Weapon (mace, shortsword, longsword, magicFangs, and magicAxe) All the weapon constructors are straightforward, putting in the stats of the weapon and strings.

For the scrolls, the scroll parent class is straightforward with no private variables. All of its derived scrolls simply hold the name and action log.

Then there are the staircase and goldenIdol which are from gameObject, these are objects that only hold a string with its name, the other variables are initialized since they are never used.

**(Dungeon.cpp and Dungeon.h)** This holds the meat of our program, nearly all the game functions occur inside the dungeon.cpp file. A dungeon refers to the level in a game. The constructor initializes the size of a dungeon to a 2D array, which holds the if there are walls, actors, or is empty. It also sets the level to 0 and defaultly goblinSmell is set to 15, then win is set to False. We then calculate a private variable called monsterAmt which will set the number of monsters on a level. We do the same for objAmt which sets the number of items on the ground per level. Then the constructor calls createGrid() which generates a random map.

***CreateGrid***

|  |
| --- |
| set a random number of rooms from 4 to 6  create a vector of type room (a struct containing four ints being the height, width, and a x,y  coordinate inside the room)  nested for loop on the 18x70 grid and set it to all walls  for loop on the number of rooms in the vector  for each room, set each room with a random height and width and x,y coordinate  create a boolean initialized to false to track overlap  if we are not creating the first room,  nested for loop to check through the rooms every spot and  if at any spot, it hits another room (aka is already emptied out) then set overlap boolean to true and break  if statement to check one tile past the box to ensure a border  overlap is true if edges overlap and break  after checking, if overlap is still false,  then set the room’s spots to empty using a nested for loop  (here we hollow all of the room except for a 1 spot border around the room to keep an edge and separate rooms)  increment the vector only if we set the room, else do it again until it makes room  --- now let’s set our corridors ---  for loop on every room stopping at the 2nd to last  get the middle of the room and the NEXT room in the vector  if the room1’s center is south west of room2’s center,  dig out a tunnel west of room1 until it reaches room2’s center’s X coor  then dig out a tunnel north until it reaches room2’s center  repeat this for the other 3 quadrants (northwest, northeast, southwest) |

Now that we have a map, we call addObjects and addMonster which generate a random number of monsters and objects into a private list for each. For addObjects, we dynamically allocate and select a random object to place using a for loop on the objAmt which is initialized in our constructor, and a random integer function and switch statement decides what item to push onto the list. Then we always push either a staircase or idol onto the list, depending on the level. If the level is less than 4, there are always stairs, else there will be an idol. Then we call setObjects which randomly assigns the row and column for an item and uses the setter function of a game object to store that.

For addMonsters, we run the same algorithm that we used to place our player, randomly assigning a spot and setting it if it is empty else randomize another spot. Depending on the level, we have if statements to select random monsters from a set of available choices as some monsters do not appear until later levels. These monsters are dynamically allocated into the monsterList with their row and column and we set the cell status to being occupied by a monster. Each monster has a global int stored in the 2D array so the display function will know what to display.

Remember how playerMove called monstersMove? Well that function moves all monsters if they can. Let’s get into it. Monsters move iterates through the list of monsters and sets a direction to empty for now. If the monster is a **goblin** it will call its goblinSmell function which calculates if it can move and where to move, using that, we have a switch statement almost identical to the player movement switch statement, the only difference is that it checks bool function called isNext (which outputs true if a player is next to the monster) if so, then it will attack the player.

For the goblin smell algorithm, I take in a monster pointer, row, col, and an integer that is the goblin’s smell distance. It is a recursive function with 2 base cases, if the smellDistance is 0 or less, then return false, if it passed a row and col that is the player’s spot, then return true. Now we compare the goblin’s spot to the player’s spot. If is north of the player, and goblinSmell doing down one spot returns true, then move the goblin and return true. It does this same thing for all directions north, west and east as well. Decrementing or incrementing the row appropriately for the direction. Every time we recursively call goblinSmell, it decrements the 4th parameter being the smell range so the goblins will not always move towards the player if it has to move more than 15 times.

For Bogeyman and Snakewoman, I wrote two functions for their movement. We have a function called smell that takes in their pointer and their smell range. Using the monster’s position and player’s position, we subtract the row and column values and absolute value it to see if they are within the smell range parameter we passed in, if it so return true else false. So if smell returns true, the moveMonster switch statement case for these two monsters moves forward. But the direction it moves in is derived from the second function called towardsPlayer. This is similar to the logic we used in creating the map. We have if statements for each quadrant around the players’ position, if it is Southwest, move either North or East towards the player. We do this for all directions and output what direction to head. Using that we can move the row and col of the monster and update the board cell status.

For attack, we take in an actor pointer for both the attacker and defender. We get the weapon of the attacker and do an extra calculation if the weapon is magic fangs as it can put the foe to sleep. We also need to dynamically cast the pointers passed in in order to correctly set death. We also output the right action logs if they hit or miss and have empty strings that get cacentated together through the attack function and get pushed onto the dungeon’s action log at the end. The action log is made up of a queue that outputs refreshes in display. After the attack functionalities and calculations that update the hitpoints and such, we need to register death. Our death is only for monsters since this cleans up their character after death and deletes them appropriate from the monster list. We have a function called isDead that returns true if the hp is equal to or less than 0. Then set the board spot to empty and iterate through the monster list to find them and delete them. When monsters die, it calls monster drops which is made of a bunch of formulas that push new items based on probability to the dungeon objects list.

To pick up said items and other items, we have the inventory functions in dungeon, pick up object is called by input g and iterates through the list checking if the player is over the item which allows pick up. First we check if the inventory is full, if the object is a staircase (which would exit), and if it is goldenIdol which would set the bool win variable to true and exit the game. Otherwise, we push the item onto the inventory and erase it from the objectlist and push action strings that indicate we picked something up. To show our inventory, we simply iterate through the inventory and cout the item and increment char a to display it a - z. To read and wield items, which gets called in the game’s play() input switch statement, we take a second input and take that minus ‘a’ to get the spot in the inventory. Then we iterate through the inventory list and try to either setWeapon or readScroll, we dynamically cast the gameObject pointer and would not let a weapon be read or scroll be wielded. If we read a scroll, we have to delete the pointer and erase the item in the list.

Lastly, we have the nextLevel and clearLevel functions which coincide and get called when the player inputs ‘>’ on top of a staircase. nextLevel takes in the player pointer and gets that if the player does this exact thing and if so it calls clearLevel, which increments the m\_level variable and erases and deletes the dynamically allocated monsters and items on the level we just ‘left’. Then it calls create grid to make a new map and recalculates the monster and item amounts on the next level. Then it runs the samn do while algorithm to place a player in a valid empty spot in the map and then calls addObjects and addMonsters. This basically is a combination of our constructor and destructor…. which is our last function! The Dungeon destructor deletes the m\_player which is dynamically allocated in the game's play function and it iterates through the monster and item lists and deletes and erases it to prevent memory leaks!

**a list of any known bugs, features not implemented, or serious inefficiencies.**

* The only bug that I am aware of is that the goblinSmell and movement is not fully functional. There are times where the goblins get stuck in walls when they should be able to fully traverse to the player’s spot.