Name – Praneel Jhunjhunwala

UID – 905394605

PROJECT 3 REPORT

**Goblin recursive function-**

The program does not have a recursive function for a goblin. For functional gameplay, the goblins move function programmed to be identical to that of the bogeyman.

**Design Description –**

**Inheritance structure -**

**Individual classes –**

* Game
* Dungeon
* stairs
* idol

**Base class** – Actor

**Derived classes** –

* Player

All monsters-

* Snake
* Goblin
* Dragon
* Bogeymen

**Base class-** gameObj

**Derived classes –**

* Weapon
* Scroll

**Description of classes –**

**Class Game –**

This class is responsible for interacting with the user of the program as it takes in a character and based on the character calls the functions required for the game to proceed.

The constructor of the class creates a dungeon object and a player object. The dungeon object is passed the level of the dungeon and the smell distance of the goblins and the player object is passed the starting location of the player.

The main function used in this class in game.play(). This function carries out the game as it takes in a character from the user of the game while the character is not a ‘q’. The Game class also has a list of strings as a private member. This list of strings is passed as a parameter to other functions which add messages to it which are all displayed after every turn by the user.

The Game destructor when called deletes the pointer to the player object and the pointer to the dungeon object.

**Class Dungeon-**

This class is what holds most operations in the program. As the dungeon is where the Actors interact with each other, all monsters and game objects are created in this class. The constructor of this class takes in two parameters, the level of the dungeon and the smell distance of the goblins. This class has a private two- dimensional array of characters which it uses to display the maze in the game. The constructor calls a function which generates a random maze and changes the characters in the two-dimensional array accordingly. It also creates a stair’s if the dungeon level is less than 4 or an idol if the game level is 4. This function is also responsible for creating the objects in the dungeon such as scrolls and weapons and adding them to a private member list of gameObj pointers.

The constructor then calls a function which takes in the level of the dungeon, and accordingly adds monsters to the dungeon by adding new monsters to the private list of actor pointers called ‘monstlist’.

Thus, the Dungeon class holds a list of Actor pointers used to access the monsters in the dungeon, a list of gameObj pointers used to access the various objects in the game, a pointer to a stairs object and a pointer to an idol object.

The Dungeon class holds the display method used to print out the updated maze after each turn. The Dungeon class also holds all methods which are used to move the monsters, make a monster attack a player and make a player attack a monster. These functions are called by the game.play() method according to the input by the user.

Class Dungeon also has a private struct called room and a list of room pointers used to generate the maze.

The Dungeon destructor, when called has deletes all the dynamically allocated objects, i.e. all the monsters in the list of monsters, all the weapons and scrolls in the dungeon and the stairs or the idol in the dungeon. The destructor also loops through the list of room pointers and deletes all the rooms used to create the dungeon maze.

**Class Actor –**

This class is the base class to all the Actors in the dungeon, including the player and all monsters. Thus, this class hold private variables for all the attributes of the actors in the dungeon. This includes their name (string), current row and column in the maze, max and current hitpoints, armor, strength, dexterity, sleep time and their weapon. As weapons are a type of gameObj, each actor has a pointer to a gameoBj, which is used to store the actor’s weapon. The Actor class further includes getter and setter methods for all these attributes which are used to change the attributes as the game progresses. The constructor of the actor does not include anything, rather the constructors of the derived classes (monsters and players) use the actor setter methods to assign values to all the actors attributes. The Actor class also includes a pure virtual getCharacter method which is used by all Actors to return the character that represents them in the game. For example, a player object will return ‘@’, a Goblin will return a ‘G’ and so on.

The actor destructor is used to delete the weapon that each actor holds.

**Class Player –**

The player class is a derived class to the Actor class, thus the player constructor called when a player is created at the start of the game, calls all the methods in Actor used to give values to all the player attributes. The player constructors takes as parameters the starting coordinates of the player. The constructor

The player is also the most complex actor in the game as it has its own list of gameObjs used for the player’s inventory. Thus, the player has a private list of gameObJ pointers which it uses to access the items in its inventory.

The player class has a playermove method which is used to move the player on the grid, an inventory method used to display the players inventory onto the screen, a method to add items to the player inventory , a method to read scrolls and a method to wield weapons from the players inventory.

The player destructor is used to delete all the items in the player’s inventory, except the weapon that the player currently wields. This weapon is destroyed when the Actor destructor is called for the player.

**Class Snake, Bogeymen, Dragon, Goblin (all monsters)-**

All monsters are derived classes from base class Actor, thus they have limited uses. The constructors for these classes are called whenever a monster is created, and this constructor is used to call methods in Actor which set values to the Actor’s attributes. The monster classes do not have any unique functions apart from getcharacter() which is used to return the character the monster is represented by on the screen. All other functions required are derived from the actor class.

**Class gameObj –**

This class is the base class to the weapon and scroll classes. Thus, this class is used to store all the attributes of the scrolls and weapons. For Scrolls, the only useful attributes are their name and their row and column on the grid, while weapons also have a dexterity bonus, a damage amount and an action sound associated with them.

The gameObj class stores all these attributes as private variables and uses getter and setter methods to change and return their values.

The gameObj constructor takes as parameters the objects location on the grid and its name as a string. The name of the weapon/scroll is used to give values to its other attributes.

**Class Scroll and Weapon –**

The scroll and weapon classes are derived classes from the gameObj base class. As scrolls and weapons do not have much functionality, they only have a constructor and no other functions. The constructor takes as values the name of the Scroll/ Weapon and their row and column, and then passes these values to the gameObj constructor, which uses them to assign values to the object’s attributes. All other functions used are derived from the base class gameObj.

**Class idol and class stairs**

These classes are individual classes and don’t share any relationship with each other, however they share very similar attributes. As stairs and idols do not have any unique functionality and do not move, they only have two private variables used to store their location on the grid. They have a setpos method which takes two values and assign them as the objects row and column on the grid. These classes also have getter methods to access their location on the grid.

**Non- trivial Functions –**

Note – All non-trivial functions were included in three classes in the project. Class Game, class player and the most complex class Dungeon.

Class Game -

**void play() {**

call display function using the dungeon pointer and pass it the pointer to the player

set char dir = getcharacter();

loop while( dir is not ‘q’){

if((1/10) probability is met and player’s hitpoints are less than max hitpoints)

increase players hitpoints by 1

if (players sleep time is not 0)

decrease players sleep time by one

else if (player’s sleep time is 0){

if (dir is ‘c’, then cheat)

increase players strength to 9 and hitpoints to 50

set bool move = false;

if (dir is a direction(up, down, left , right)){

then set move = use player pointer to call playermove(dir, dungeon pointer) ;

// this sets move to true if the player moved and false if player did not move

}

if(move is false, i.e. the player did not move){

then check if the direction the player wants to move in has a monster in it .

use getcellstatus(int r, int c) method from Dungeon class to check if the cell the player wants to move into has a monster’s character in it. i.e. ‘D’, ‘B’, ‘G’ or ‘S’.

if monster is present in the cell

then call the playeraction() method and pass it the direction, the player pointer and the list of strings to add messages to.

}

If(dir is ‘>’, i.e. the player wants to descend){

Call the stairpos(int r, int c) function using the dungeon pointer to check if the player’s location matches the location of the stairs, if so then

increase the level;

delete the current dungeon pointer

create a new dungeon using the higher level

give new coordinates to player, making sure there are no actors on that spot

call the display function in dungeon

continue;

}

If(dir is ‘g’, i.e. the player wants to pick up an object){

Set string a = pickoBj(player pointer) method in dungeon

// this method returns the name of the object picked up, or returns a different message if there is no object at the location or if players knapsack is full

Add appropriate message to the list of string messages ,depending on string a.

}

If(dir is ‘w’){

display the inventory by calling inventory() function using pointer to player.

set string x = call wieldweapon() function using pointer to player

if string x is the name of a weapon

add message saying player has wielded weapon x to message list

if x is the name of a scroll

add message saying player cant wield x to message list

if x is something else

do nothing

}

If(dir is ‘r’){

display the inventory by calling inventory() function using pointer to player.

set string x = call wieldweapon() function using pointer to player

if x is scroll is of teleportation

give player new coordinates

if string x is the name of a scroll

add message saying player has read scroll to message list

if x is the name of a weapon

add message saying player can’t read x to message list

if x is something else

do nothing

}

If(dir is ‘i’)

Display inventory using inventory() function

Input another char using getcharacter()

If(dir is g)

Check if the players position has the idol on it using idolpos(int r, int c) function in Dungeon

If so ,

The output the message saying player has won

Call end() function

Break;

}

}

Call the monstersmove() function and pass it the message list to add to and the pointer to player

Call the display function in Dungeon

Call the displaymessages function

If(players health is negative)

Call end()

Break;

set dir = getcharacter();

if (dir is ‘q’ and player’s sleeptime is greater than 0)

change value of dir to ‘t’ ; // thus player will not be allowed to quit the game when asleep

**}**

**void displaymessages(){**

loop through the list messages

print every message

erase every message after printing

**}**

**void end(){**  keep inputting characters till getcharacter() is a ‘q’

**}**

**Class Dungeon –**

Note – The function used to create the randomly generated maze is called **dungeonstuf()**.

**Dungeon(int x, int y) {**

Set level = x;

Set gobsmell = y; // used in the goblin move function

Call dungeonstuff(); // creates the dungeon and its objects

Call makemonsters(level); // creates the monsters according to level

If (level is 4)

set the pointer to idol m\_idol to new idol(); // create an idol if level is 4

give the idol an appropriate location on the grid

**}**

**void display(player\* pl){**

loop through all the characters in displaygrid[][]

if the character , is not a wall, empty space , or stairs, set it to an empty space

if (level is 4)

set the appropriate spot in displaygrid[][] to ‘&’ , representing the idol

loop through the dungObjs list using an iterator {

for each object, if the object has an action sound “none”

set the appropriate spot in displaygrid[][] to ‘?’; // representing a scroll

else

set the appropriate spot in displaygrid[][] to ‘)’; // representing a weapon

}

Loop through the list of Actor pointers monstlist{

For every monster

set the appropriate spot in displaygrid[][] to the monsters getchar() function; // returns the character use by that monster.

}

Using the player pointer passed as an argument

Set the appropriated spot in displaygrid[][] to ‘@’

Use the clearscreen() function;

Use a nested for-loop

Print out every character in displaygrid[][] to show the maze

Print the dungeon level and player statistics;

**}**

**Note –** This is the function used to create the randomly generated maze

**void dungeonstuff(){**

using a nested for-loop fill the entire displayGrid with ‘#’s

use a random integer from 4 to 6 to determine the number of rooms in the dungeon

loop while the size of the list of pointers to rooms has not reached the number of rooms required {

chose valid starting row and column for the room using randint

use a while loop to choose a valid length and breadth for the room // choose new values till the room isn’t entirely contained inside the grid.

set bool intersect to false ;

using the starting coordinates and length and breadth of the room loop through displaygrid[][]

if any of the places in the room is an empty spot, this implies that the room overlaps with another.

If room overlaps, then set intersect to true

If room does not intersect with any previously generated room. (i.e. bool intersect is false)

Then create a new room pointer passing it values for the starting and ending coordinates of the room

Loop through all spots in the newly generated room and set all those places in displaygrid to the empty character

}

Now the rooms have reached the number required.

To make corridors, loop through the list of room pointers called ‘dun’

Set iterator p = dun.end();

Do p--; // this will set p to the last element in the list

Loop while (p is not dun.begin){ // loop from the last to the first item in the list

set the starting r and c to the centre of the room using variables midr and midc in the room struct

decrement p to access next room . i.e. p--;

set the ending r and c to the centre of the room using variables midr and midc

if (starting column is less than ending column){

make an empty row from the starting column and row to the ending column

if (starting row is less than the ending row)

make an empty column from the starting row to the ending row

else if(ending row is less than the starting row)

make an empty column from the ending row to the starting row

}

else if (if ending column is less than starting column)

make an empty row from the ending column and row to the starting column

if (starting row is less than the ending row)

make an empty column from the starting row to the ending row

else if(ending row is less than the starting row)

make an empty column from the ending row to the starting row

}

Else if (the starting and ending columns are the same){

if (starting row is less than the ending row)

make an empty column from the starting row to the ending row

else if(ending row is less than the starting row)

make an empty column from the ending row to the starting row

}

// now corridors between rooms have been made

If the level is not 4 , then create new stairs and assign it to stairs pointer m\_stair

Choose valid coordinates for the stairs, i.e. one without another non-actor

Set the position of the stairs using m\_stair->setpos(rs, cs);

//now creating 2 or 3 game objects for each level

Using a for loop create 2 or 3 new objects for the dungeon {

Choose valid coordinates for the object, i.e. one without another non-actor

Use equal probability for each object and create a new weapon or scroll accordingly using a gameObj pointer

Set the position of the game object

Push the pointer onto the list of dungeon object called dungObjs.

}

}

**}**

**string pickobJ(player \*pl){**

loop through the list of pointers to objects in the dungeon dungObjs{

if the players position matches an object in the dungeon{

erase the object from the dungObjs list

call(addtoinvent(pointer to gameObj) function in player)

if it returns false; // this means player inventory is full

push the object back into the dungObjs list

return appropriate message // convey that knapsack is full

if(object is added to players inventory)

return appropriate string; // depicting what object the player has picked up

}

if the player is not on top of an object, return message “none”

**}**

**void makemonsters(int level){**

depending on the level of the dungeon

create that many number of monsters using appropriate range

considering which monsters appear on which level and using equal probability for each monster

create the number of monsters required randomly

push back a pointer to each monster created into the list monstlist

**}**

**void playeraction(char d, player\* pl, list<string>& a){**

// this function is called only if the player can attack a monster, i.e. moves in its direction

Loop through the list of monsters monstlist

Depending on the direction char d check what monster is on that spot

If you find a monster on the spot

Set a pointer to an Actor to that monster

Call the attack() function, passing it a pointer to the player, a pointer to the monster and two bool variables by reference, which show if the player has been hit , or put to sleep

If ( monsters health is negative){

Push the appropriate message onto the string list ‘a’ passed by reference

Use the appropriate probability and the name of the monster that died to create a new object at the spot the monster died on

Push a pointer to that object to the dungeon objects list

Delete the monster that died and erase it from the list monstlist

}

If the monster has been put to sleep, or if the player hit or miss the monster

Push the appropriate message onto the string list ‘a’

**}**

**void attack(Actor\* attacker, Actor\* defender,bool& sl,bool& hits){**

calculate the attackerpoints and defenderpoints using the appropriate formulae and the pointers to the Actors passed as arguments

calculate the damage points

if (the attacker hit the defender){

set the bool hits to true;

reduce the defenders hitpoints

if the weapon used by the attacker is magic fangs

use the appropriate probability to check if defender has been put to sleep

increase the defenders sleep time appropriately

if true, set bool sl to true // implying that the defender’s sleep time has increased

}

**}**

**void monstersmove(player\* pl,list<string>& a ){**

loop through the list of monsters monstlist {

if (monster is a dragon){

increase the dragons health If the probability is met.

}

if (monsters sleep time >0)

decrease the monsters sleeptime by one

else {

for each monster, check if the player is next to it

if a player is next to it {

call the attack() function passing the monster pointer as attacker and player pointer as defender, and passing two bool variable by reference , which will show whether the player has been hit and whether the player has been put to sleep

add the appropriate message to the string list ‘a’ depending on whether the player is dead, whether the monster hit the player, the monster missed the player or if the monster put the player to sleep.

}

If(player is not next to the monster) {

Call the SnBmove(Actor\* monst, Actor\* player) function if the monster is a snakewoman or bogeyman;

Call the goblinmove() function if the monster is a goblin, and pass it a char variable

by reference;

Depending on the value of the char variable, move the goblin In appropriate direction;

}

}

**}**

**void SnBmove(Actor\* monst, Actor\* player){**

if(monster is a snakewoman)

set int range = 3;

else

set int range = 5;

set bool present = false;

depending on int range run loops to check each spot within the range to check if a player is present in the range;

if player is present within range

set the bool present = true;

if(present is true){ // implying that the player is within range

if the player is below the monster and if the spot below the monster does not have a wall or another monster, make the monster move down , change the monster’s row and column and set the monster’s initial position in the grid to empty.

similarly, check for all other directions and move in the first direction which gets the monster closer to the player

}

**}**

Class Player –

**bool playermove(char dir, Dungeon\* m\_dung){**

for direction ‘dir’

check if the player can move in that direction.i.e. if the spot in that direction does not have another actor or a wall in it.

Then move the player in that direction. i.e. change the player’s row and column and set the players initial position in the grid to empty.

return true;

If the player cannot move

return false;

**}**

**void inventory(){**

call the clearscreen() function

set char I = ‘a’;

loop through the players list of gameObjs called pObjs using an iterator

use char I alongside the name of the inventory object to print out each object

increment char I each time;

**}**

**bool addtoinvent(gameObj\* r){**

set int x = 0;

loop through the players list of gameObjs called pObjs using an iterator

increment x each time to finally see how many items are in the players inventory f

if the number of items is below 26

push the pointer ‘r’ into the player’s list pobjs

return true;

else

return false

**}**

**String wieldweapon(){**

input a character from the user, using getcharacter();

check if the character given is valid. i.e. check if the character is a letter from ‘a’ to ‘z’ and if for that character there exists a corresponding item in the player’s inventory. If so, set a pointer ‘targ’ to that item in the inventory

if(character is not valid )

return “none”;

else if(character is valid){

check if targ is a pointer to a scroll. i.e. if targ has an action sound of “none”.

i.e. if targ-> getaction == “none”

then return the object’s name

else if the object is a weapon

set the players current weapon to targ

return the weapons name

}

**}**

**readscroll(){**

input a character from the user, using getcharacter();

check if the character given is valid. i.e. check if the character is a letter from ‘a’ to ‘z’ and if for that character there exists a corresponding item in the player’s inventory. If so, set a pointer ‘targ’ to that item in the inventory

if(character is not valid )

return “none”;

else if(character is valid){

check if targ is a pointer to a weapon. i.e. if targ does not have an action sound of “none”.

i.e. if targ-> getaction != “none”

then return the object’s name

else if (targ is a scroll) {

set string name = targ->getname; // get the name of the scroll

run a loop through the list pObjs and stop when u reach targ

delete targ

erase the iterator to that scroll

}

using string name, check what the scroll is supposed to do.

If the scroll is of teleportation, just return the name;

If the scroll is any other apart from teleportation

Change the player’s attributes accordingly

Return the name

}

**}**

**Inefficiencies/bugs/Incomplete-**

**Incomplete –** The program does not have a recursive move function for the goblins.