**PROJECT 3**

**MINIROGUE**

**REPORT**

**NAME: SANCHIT AGARWAL**

**UID:105 389 151**

All the requirements and particulars of the spec are implemented . There are no assumptions whatsoever . The program is complete in all means

***GOBLIN MOVE***

Goblin Move can be found at the very last of Actor.cpp . The function is implemented as a combination of four different functions namely pathExists , move() , movableSpot and clearData .

***IN THE FOLLOWING REPORT IT CAN BE FOUND ON PAGE NO. 17***

I SUGGEST THE RESPECTED GRADER TO RUN THE PROGRAM ON **MAC OS TERMINAL BY EXTRACTING THE EXECUTABLE FILE FROM X-CODE** FOR BEST PERFORMANCE . HOWEVER THE GAME RUNS PERFECT IN X-CODE AND G32 TOO .

**In the following 5 pages I have given a comprehensive program design with the responsibilities and purpose of each graph stated**

1) PROGRAM DESIGN

CLASS DUNGEON

--> setting up the GameBoard -> an 18-70 2D ARRAY

--> creating rooms to give a corridor like feel

--> Adding Player to the Board

--> Adding Monsters and GameObjects on the Board

-->It has pointeres to Various other classes such as Players,Monsters and Objects which in turn help to examine different spots on the grid and also to Bridge different classes

For eg-> if a player is at a particular spot and presses g then the player class asks the dungeon class that is there a object here and Dungeon which serves as a bridge further asks the GameObject class .

This Bridging Property of the dungeon helps in Various tasks such as

-->Checking Validity of a position so that a player/Monster/Object can be moved or placed there

-->Checking the presence of Various items such as Monster/object /Player so that attack or going to next level can be executed

-->Stores all gameObjects on the board and Monsters vectors with pointers to each of these things ...This helps in removing Gamobject/Killing Monsters from the board by using Polymorphism in a simple for loop

The dungeon class , not suprisingly enough, happens to be the only link between all the classes

At each level a new Dungeon is created by the Game class as a parameter of level is passed in the dungeon's Constructor .The dungeon adds appropriate number of Monsters according to the level .

SO BASICALLY THE DUNGEON CLASS RELATES TO ALL OTHER CLASSES BY HAVING POINTER TO EACH CLASS . THIS PROVIDES A BRIDGE BETWEEN DIFFERENT CLASSES AND ALSO PROVIDES DUNGEON ACCESS TO ALL OF THEM SO THAT APPROPRIATE CHARACTERS CAN BE PRINTED AT APPROPRIATE LOCATION .

A STYLISTIC NOTE: Dungeon is just a grid with walls and rooms all the other objects that are shown in the display are investigated at print time by using the pointers to different classes and comparing their position attributes to the current spot being printed ..

CLASS ACTOR AND ITS SUBCLASSES

CLASS ACTOR

BASE CLASS

Contains attributes that are common to both Player and Monsters

Derived Classes -> Player , monster

COMMONALITIES OF ALL ACTORS

1. They all lie in some Dungeon and so Actor Class contains a pointer to a Dungeon ^^
2. They all have some position on the Dungeon
3. The all have four Hit points (that need to be accessed and updated)
4. They all have a sleep time (that need to be accessed and updated)
5. They all have a printable symbol
6. They all have a weapon (that need to be accessed)
7. They all have a name and all of them attack (printed during attack)

All of this is accounted in the actor class and the details are in the section below called emphasization

Class Monster

Derived from Actor

RESPONSIBLILITIES

1. Passing the individual Data of all the monsters to Actor’s Class
2. Moving Monsters (smell parameter)

NOTE : In the movement of Monsters , the move function is Inherited by various classes …However as Snakewoman and Bogeyman have a similar kind of Move function , the Monster move function is programmed according to them and is overridden by the other classes

CLASS PLAYER

Derived class from Actor

RESPONSIBILITIES

The basic responsibilities are

1. Picking Object
2. Moving
3. Reading scroll
4. Wielding weapon
5. Checking Inventory
6. “CHEATING”
7. Bump in max hit points

So Basically the purpose and responsibility of this class is to handle everything UNIQUE to the player apart from the commonalities that are handled by the Actor class

Class Bogeyman

Derived from Monster

RESPONSIBILITIES:

Passes its data such as name, symbol and points array to the Monster class

Inherit the MOVE FUNTION

Only appears from level 2

Class Snakewoman

Derived from Monster

RESPONSIBILITIES:

Passes its data such as name, symbol and points array to the Monster class

Inherit the MOVE FUNCTION

Appears on all levels

Class

Derived from Monster

RESPONSIBILITIES:

Passes its data such as name, symbol and points array to the Monster class

Overrides the MOVE FUNCTION

Have a special Recursive MOVE FUNC.

Appears on all levels

Class Dragon

Derived from Monster

RESPONSIBILITIES:

Passes its data such as name, symbol and points array to the Monster class

Overrides the MOVE FUNCTION

Ability to bump up Hit Points

Only appears from level 3

EMPHASIZATION ON DESIGN OF ACTORS

In the Base class Actor all the common attributes are stored .. All can be accessed by the dungeon as dungeon has a pointer to player and Monsters .

The Base Class Actor has two constructors . The one without the position and the dungeon parameters is exclusively for the player as due to different levels the player for obvious reasons is a member of class Game and thus needs to exist and be created before and separate from the dungeon . The base class also has some exclusive function to set dungeon and position which are utilized by the player once the dungeon is set up.

Apart from the trivial accessors and mutator functions a very Important function is the function called attack . To bring to your notice Attack is a string type function that is defined in such a generalized manner that it takes pointers to two actors and then update the points of the attacker and defender and returns the appropriate string too . Instead of having two functions or overrides the attack function accommodates for both attack of player on monster and of monster on player .

Dynamic Casting is used in player class while wielding a weapon or reading a scroll as the pointer to a GameObject needs to be converted to a weapon or a scroll pointer . Player class also has boolean functions like didPlayerWin etc. to account for different individualities of the Player such as winning the Game.

Monster Class in order to move uses a multitude of Functions …

1. isPlayerinRange->checks in Snakewoman/Bogeyman style for the presence of player in the area of Influence
2. distancefromPlayer->checks the distance of Player from a given monster position
3. attempt move -> give move proposals and in case of Snakewoman and Bogeyman valid proposals are randomly selected

Goblin and Dragon override the Goblin Move function for movement ..

Class HealthScroll

Updates MAX\_HitPoints

Class Scrolls

Derived from GameObject

When read have a property to updatePlayerStats and so there is a virtual function present which is overridden by various scrolls

The values of the above quantities for all the weapons is stored here

Can be picked and read

Class ArmorScroll

Updates Armor points of the player

Class StrengthScroll

Updates Strength points of the player

Class DexterityScroll

Updates Dexterity points of the player

Class TeleportScroll

Changes player position randomly

Class MagicFangs

Passes its data such as name, symbol and points array to the Weapon class

Special Booleans for this in other classes as it can put a Actor to sleep

Class MagicAxe

Passes its data such as name, symbol and points array to the Weapon class

Class LongSword

Passes its data such as name, symbol and points array to the Weapon class

Class ShortSword

Passes its data such as name, symbol and points array to the Weapon class

Class Mace

Passes its data such as name, symbol and points array to the Weapon class

Class Weapon

Derived from GameObject

Has two special values not common to all GameObjects

-->DamageAmount

-->DexterityBonus

The values of the above quantities for all the weapons is stored here

Can be picked and Wielded

A screenshot of a cell phone

Description automatically generated

CLASS GAME

***CLASS GAME***

The Game Class Basically sets the Game up...

In the constructor of Game Class

The Player is added

The dungeon is created

The dungeon is populated

And Various other booleans are initiated

Apart from this the Game class have pointers to both player and Dungeon ....

Using the pointer to dungeon the game class gives command to dungeon and various other monsters etc(as Dungeon has those pointers) and the Dungeon executes these commands .. To make the code clear and simple there are many functions in the dungeon class itself like add monsters , add GameObject etx . The game just calls these functions to get the work done ..

The entire playing of the game takes place in the Game class . Here the user enter commands , those commands are interpreted and then the necessary functions are called . The game here can easily be called the Master class that handles everything by calling functions of other classes .

Important things like Player winning the game and levels getting updated are handled by the Game Class . After every level the Old Dungeon is deleted and a new Dungeon is created in the Game Class .The player is then added to the newly created dungeon . The Game class calls the constructor for dungeon and passes in Player pointer and level as parameters to ensure smooth functioning of new levels .

One of the very Important functions in the Game Class is the Take Player Turn which accounts for every possible user command and calls appropriate functions according to the command.

***LINKAGE***

AS THERE ARE TOO MANY CLASSES , DRAWING ARROWS TO SHOW LINKAGE WOULD BE VERY MESSY SO BELOW IS THE REPRESENTATION TO SHOW LINKAGE

GOLDEN IDOL

Uses Dungeon pointer

Pointer to Dungeon

GAMEOBJECT

SCROLL

Uses Dungeon pointer

WEAPON

Uses Dungeon pointer

STAIRCASE

Uses Dungeon pointer

Pointer to Player

Pointer to Dungeon

GAME

PLAYER

Pointer to

GameObjects

(For Inventory)

MONSTER

Utilizes the Pointers to Actors

Pointer to Weapon

Pointer to Dungeon

ACTOR

Pointer to GameObjects

Pointer to Monsters

Pointer to Player

DUNGEON

PSEUDO-CODE/EXPLANATION TO NON TRIVIAL ALGORITHMS

***DUNGEON***

The very nature via which I have Implemented the project is that apart from Walls and Path Nothing is Hard-Coded on the Dungeon Grid . The position attribute of every Actor/Non-Actor is stored in their respective classes and for each of them I have made a function in Dungeont called "is\_\_\_\_\_\_At" which using the pointer to the class accesses it's row and column . Thus it is only when I am displaying the Dungeon , I call the respective function for each Actor/Non-Actor on each spot and ask if they lie there . If the boolean result is true I call another function called "getSymbol" which tells me the character to print and that character is printed .

Create Rooms

**void Dungeon::CreateRooms();**

As you will note my room making algorithm truely ensures that the rooms and the paths are as random as possible .

As the create room algorithm is very long and detailed I would like to

represent it in a point format

Here two terms are used a lot

1) Linear Rooms -> rooms assigned the first on the basis of 1 per partition

2) Overlap rooms -> rooms assigned after the first set of (3-4) rooms are assigned ..Its basically a second room in the same section

* Divide the grid in 3 - 4 sections of Random length *within given threshold* vertically
* Decide *randomly* which two partitions will accomodate more than one room
* Make the probability of rooms more than 3-4 less than one. (Just for less mess and more vividity)
* For each Partition select a row and a length *whithin a given threshold*
* *If a partition contains more than one room* then select the row and length for the overalap too
* Find the width and col within the given threshold for the "linear" room and "overlap" room and then push them into 2 different vectors as "Room" structure
* "Room" is a structure defined in dungeon.h for the purpose of storing room data
* Call the function called mark rooms which clears the space in the grid and number codes the grid with the number being related to the vector it is pushed in
* Draw Path between *Linear Rooms*

1. select a random co-ordinate on the rightmost edge of the leftmost room and name it start
2. select a random co-ordinate on the topmost edge of the room right to the 1st room and name it end
3. start making paths horizontally in the row start lies in
4. if you reach the column of the end , check by room number coding that are you inside the destined room
5. if you are inside the destined loop then complete the loop iteration and deal with the next set of rooms
6. if you are not inside the destined room start moving up/down to reach the destined room

* Repeat the above process to draw paths between overlap rooms with their corresponding linear loops just this time instead of start moving horizontally and taking vertical turns , start moving vertically and take horizontal nodes

I have thorougly done the error checking for the rooms which ensure that they will never walk of the edge and always there will exist a path from one room to any other room .

**Dungeon Display**

**void Dungeon::display()**

* move over each spot in the grid

1. (printing items are arranged according to the precedence where gameObjects are placed below players and monster because of which if a particular position is occupied by both then the Actor will be shown)

* if wall is at a particular position print that else move on
* if player is at a particular position print that else move on
* if Monster is at a particular spot print that else move on
* if Object is there at a particular spot print that else move on
* if staircase is there at the spot print that else move on
* if idol is there at the spot print that else move on
* if nothing above is found print path

**Spot getters**

Almost all is\_\_\_\_\_\_At() functions are trivial except a few

**bool Dungeon::isObjectAt(Position pos , int& which) const;**

it checks if an object is present on a particular spot and also passes by reference the index at which it occurs in the vector ....

This helps in tracing of the object ..

--> which represents to the Index of Monster/object in the respective vector that is at the given point

**bool Dungeon::isMonsterAt(Position pos , int& which) const** has the same implementation as above just over a monster vector

**bool Dungeon::isEligibleSpot(Position pos) const**

checks if a spot is eligible to move onto or to be placed

**Position Dungeon::getEligiblePosition()**

1. Create random numbers for row and column within the range of the grid until and unless the spot is an Eligible Position

2. Return that position

**POPULATE DUNGEON**

**ADD MONSTERS**

**void Dungeon::addMonsters(int goblinSmelldistance)**

1. Randomly decide appropriate number of monsters to be created according to level
2. Though a switch case decide which all monsters will be there on the dungeon according to level
3. For the number of Monsters to be placed , under a for loop randomly decide which monster in the monster variety appropriate for the level shoul be placed
4. Create that Monster and push it in the monster vector of dungeon

**ADD GAME OBJECTS**

**void Dungeon::addGameObjects()**

1. Randomly decide number of objects to be placed (Threshold 2->3)
2. if level is less than or equal to three add a staircase
3. if the level is four add a Golden Idol
4. under a for loop using a switch case , randomly assign the different object upto the number decided in step 1 and place them on the grid

**MOVE MONSTERS**

**string Dungeon::moveMonsters()**

1. Loop over the monster vector in the dungeon
2. Call move function for each Monster (Polymorphism Used)

**void Dungeon::killMonster()**

1. Loop over the monster vector in the dungeon
2. If any monster returns true to the call of isDead function
3. delete that monster
4. erase the position from the vector
5. over a switch case drop GameObjects with the required probability for the Monster which is dead

***ACTOR***

**ATTACK FUNCTION**

I have a single attack function for both the Player and the Monsters . This was possible as a private member of actor I store the names of player and each monster and for each weapon I store the corresponding verb and name as the private member of GameObjects class .

**string Actor::attack(Actor\* attacker , Actor\* defender)**

1. Set up the Attack String (Who is attacking?)
2. Calculate hit stats by the formulae provided in spec
3. if the calculations are such that the attacker hits the defender
4. if the hit points of the defender drop below 0

* set the defender to be dead
* if the defender is a monster than call the Kill Monster Function
* concatenate the attack string accordingly

1. if the weapon of attack is Magic Fangs

* calculate the probability that will the defender be put on sleep
* if the odds are in favour update sleep time of the defender
* if the defender is already in sleep , set the sleep time to be the max of newly calculated sleep time and old sleep time
* concatenate the attack string accordingly

1. else

* concatenate the attack string accordingly

**MOVE PLAYER**

**string Player::move(char direction)**

1. Set a local variable equal to the position of the player
2. Update the local variable according to the input from the player
3. If valid path then move and set players position equal to the local variable
4. If a monster is present then attack
5. if wall do nothing

**PICK OBJECT**

**string Player::pickObject()**

1. If idol is there at the spot player enter pick command -> then set player to winner and return win message
2. for all other objects on the spot if the inventory is not full then pick up the object
3. Add the Object to inventory
4. delete the object from the grid
5. if inventory is full return appropriate string accordingly

**void Player::displayInventory()**

1. clear the screen
2. cout appropriate string
3. convert the inventory index to alphabets by adding 97 and print thm over a for loop

**string Player::weildWeapon()**

1. covert the inserted character to inventory index
2. fetch the pointer of the targeted object
3. dynamically cast it to a weapon pointer
4. if demanded index is greater than inventory size then return empty string
5. if casted pointer points to a weapon --> weild the weapon and concatenate weild string accordingly
6. else return you can't weild \_\_\_\_\_

**string Player::readScroll()**

1. covert the inserted character to inventory index
2. fetch the pointer of the targeted object
3. dynamically cast it to a scroll pointer
4. if demanded index is greater than inventory size then return empty string
5. if casted pointer points to a scroll --> read the scroll and concatenate read string accordingly
6. "Magically Destruct" the scroll
7. else return you can't read \_\_\_\_\_\_

**MOVE MONSTERS**

The Monsters in my program move with the help of the following functions

**bool Monster::isPlayerinRange(int steps , Position mon\_pos )**

This function basically scans the player in the area of Influence of the Monsters and then reeturns if the player was found

The area is found with the help of two for loops transversing over the area like below

//BELOW SHOWN IS THE AREA OF INFUENCE OF A SNAKEWOMEN

\*

\* \* \*

\* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \*

\* \* \*

\*

**int Monster::distancefromPlayer(Position p\_pos,Position m\_pos)**

Calculates the distance between two points

Say there is a coordinate (a,b) and a coordinate (c,d) ,

then the distance between two points is root of (c - a)^2 + (d - b)^2 ...

As we are just dealing with positive co-ordinate i omit the root .

This functions helps me to get the distance of player from the monster so that whenever monsters like snakewomen and Bogeyman move they can ensure that they are not moving away from the player

**bool Monster::attemptmove(char direction , int& distance)**

This function proposes a move ....All proposals are surveyed by the move function and then a move is made

it also returns distance of player from the proposed move spot

1. Set a local Position type variable equal to Monster's position
2. change the local variable according to the proposed direction
3. call the function getDistancefromPlayer to find the distance of player from the proposed move and set the Pass by reference variable distance equal to this
4. if wall is there at the proposed spot return false
5. if Actor is there at the proposeed spot return false
6. else return true

**string Monster::move()**

This is the main move monster function ...This function is virtual it is overidden by the the Goblin and the Dragon but is inherited by the snakewoman and bogeyman

**THIS IS NOT THE GOBLIN MOVE FUNCTION**

1. Set up the move string
2. if sleep time isnt 0 SUBRACT it by one and return empty string
3. get Current distance of the player from the goblin
4. if Player is there in Adjacent blocks call the attack function
5. else for each of the four direction if the player is in range

* call the atttempt move function for each of the direction and if the distance of the player from the proposed direction is less than the distance of the player from the current spot then push the proposed coordinate into a vector called move choices

1. from the movechoices vector randomly select one move
2. return move msg

**string Dragon::move()**

1. Update the hit points by a probabilty of 0.1
2. If the player is in any of the adjacent spot , attack the player

***GOBLIN MOVE***

THE MAIN GOBLIN HELPER FUNCTION WHERE ALL THE PATH CALCULATION IS HAPPENING THROUGH RECURSSION

**int Goblin::pathExists(Position player,int max\_smell)**

So in this function basically I am pushing coordinate data which comprises of the co-ordinate and the distance from the player in a queue and the recurrsing over the queue . Since I am dealing with a queue I willl search for player in a fashion which popularly is called the "BREADTH FIRST SEARCH" which involves evaluating all steps at a distance 1 dfrom the player , then all steps at a distance two from the player and so. on in till 15 .. So by this way as soon as I meet the player's co-ordinates , I know for sure that the path is the shortest .

**BASE CASES**

1. if the queue is empty return -1

* pop the top co-ordinate of the queue and save that in a local variable

1. if the distance part of any coordinate pushed into the queue exceeds max\_smell and player is not found return -1
2. if player is found then return the distance of the player from the starting co-ordinate

**MAIN BODY**

1. For each of the Four directions from the current spot , if the spot is a valid spot then

* mark the spot visited
* push it on the queue with the position and the distance increased by 1 as compared to the. popped item's distance

**CALL THE FUNCTION AGAIN --> RECURSSION**

**string Goblin::move()**

**This function basically calls the path exists function and s=does some calculations**

1. Set up the move string
2. if sleeptime is not 0 , reduce it by one and return empty string
3. if player is in adjacent blocks -> attack
4. Set up a steps array which will store the number of steps for each of the direction goblin moves in and initialize them all to -1
5. For each of the four directions

* Mark the first step goblin would take in that direction as visited
* push that in the queue which will be used in recurrsion
* call the recurrsive function path exists and store the value it returns in the array
* Unmark all the visited spots
* clear up the data queue

1. Select the direction with the minimum distance to the player
2. Move the goblin in the selected direction
3. return messsage

**SCROLLS UPDATE PLAYER STATS**

As we know that when a scroll is read various points of the player are updated and so the class scroll in my program contains a function called updatePlayerStats that is ovveridden by each of the scroll types . The function basically Updates the player's stats and then returns the action string related to that scroll . eg. --> Your armor glows blue

**UPDATE LEVEL**

**void Game::updateLevel()**

This function Basically Updates the level of the game . It is so generalized so that it is even called by Game's constructor . I start the level count from -1 as it is updated before being passed to the dungeon

1. if level is greater than or equal to 0 delete current dungeon
2. if level is equal to -1 create a new player
3. update level private member value
4. create a new dungeon and pass level, player pointer as parameters
5. add game objects to dungeon
6. add Player to the dungeon
7. add Monsters to the dungeon(pass goblin smell as parameter)
8. if level equals 0 , display the dungeon

**string Game::takePlayerTurn(char playerMove)**

1. set up player move string
2. if sleep time isn't 0 , subtract one and return empty string
3. Update hitpoints by a probability of 0.1
4. Call various functions of player's class according to the user command
5. if the player is on staircase and presses '>' then update level
6. if player is on the golden idol and presses 'g' . set the palyer to be winner

**void Game::play()**

1. While user dosen't inputs 'q' or Game is Over

* getCharacter from the player
* call the function takePlayerTurn and pass that character
* if player wins cout player move and break from the loop
* Move monsters
* clearscreen
* display dungeon
* cout player and monster move strings

1. if player wins or is dead

* while player dosen't enter q -> cout the exit message,clearscreen and display dungeon ..