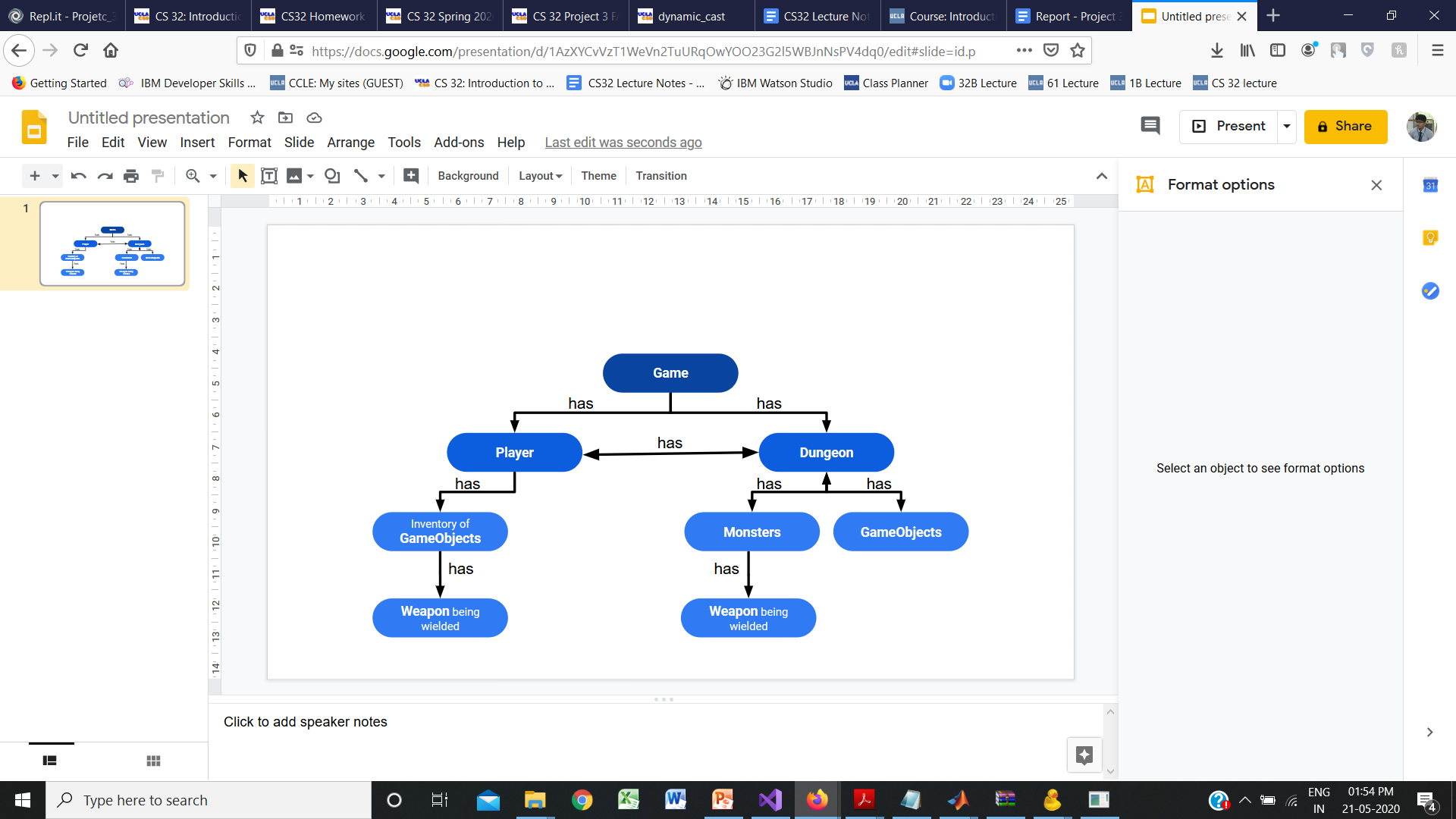
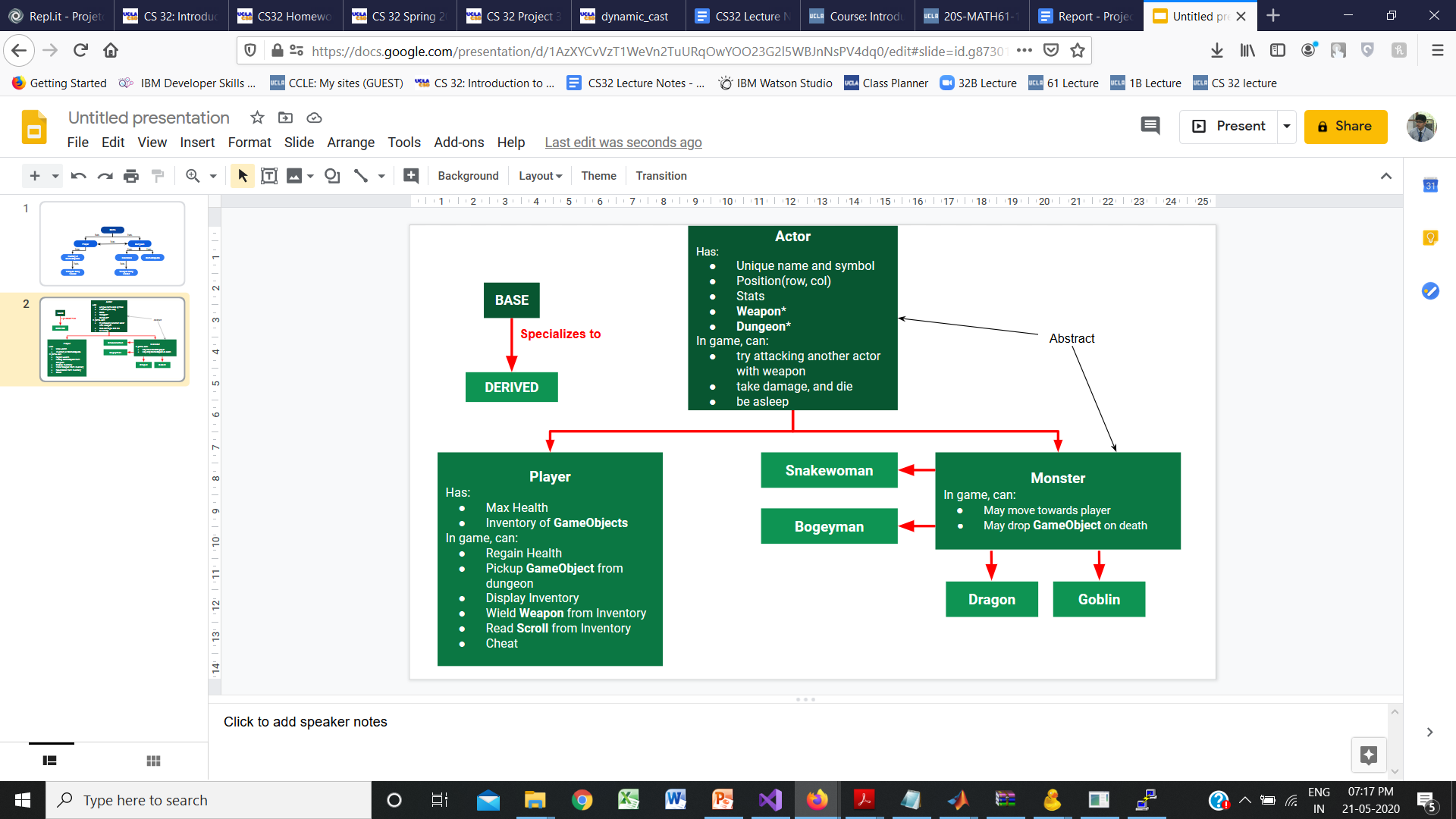
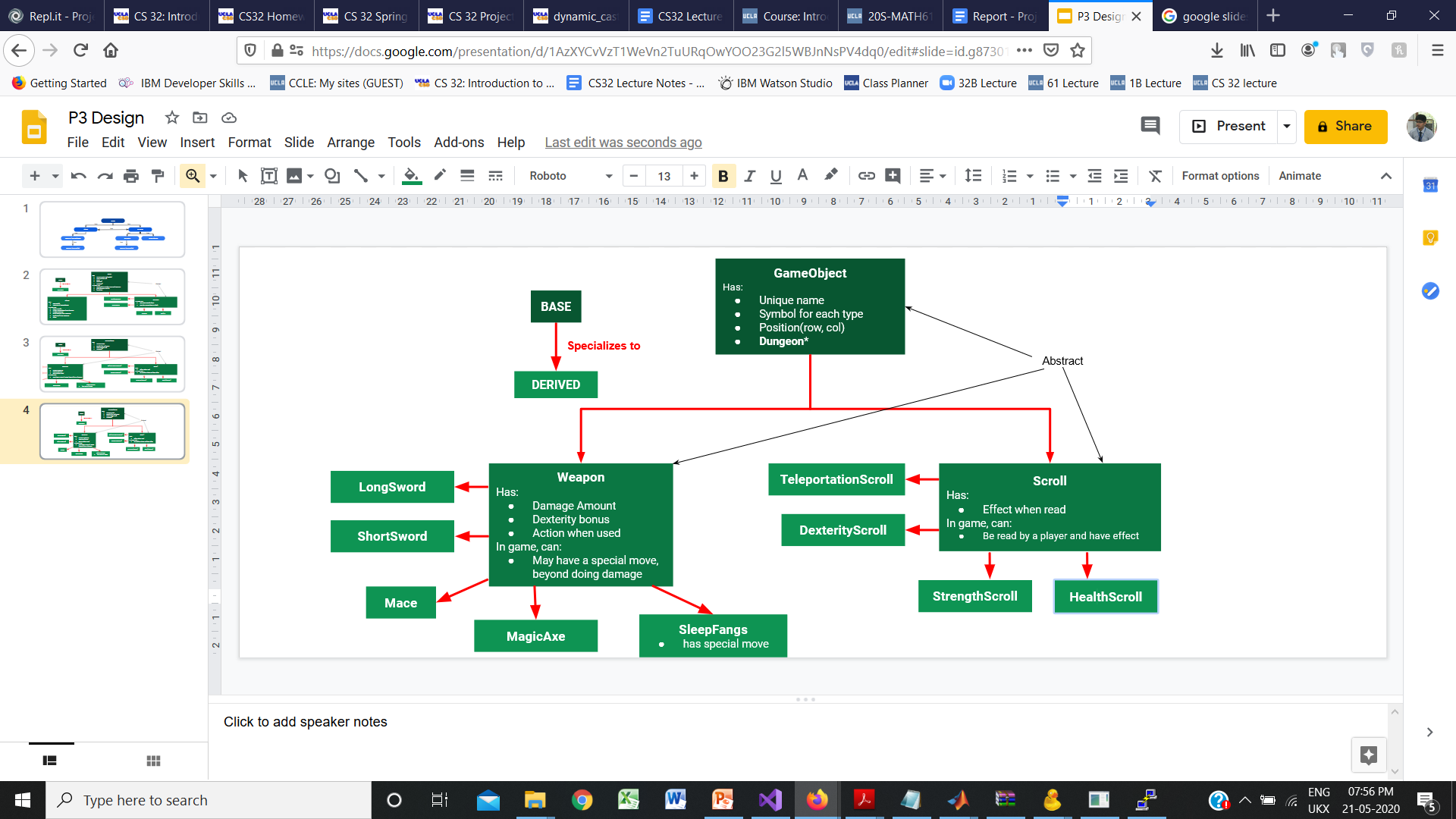
**Report**

Author: Jagrit Digani

**Program Design**

****

****

****

In the following report, the simpler non-trivial functions are marked by a bullet point and only described to give an idea of their use. Pseudocode is written for any significant or complex function, and these are not marked by a bullet point.

**Game Class**

The Game class connects all parts of the program, interfaces with the player and plays the game.

* A game is constructed with a player and a dungeon of level 0, and makes note of the given goblin smell distance
* The play function takes in and deciphers commands, and calls on the functionalities of other classes in the program to execute the commands and proceeds to other levels according to the game requirements

**Game Class Non Trivial Functions**

Constructor:

* Game::Game(int goblinSmellDistance): make a Player and level 0 Dungeon

Helper Functions:

* void Game::proceedToNextLevel(): make Dungeon of next level, clear older Dungeon
* bool Game::commandToNewPos(char command, int& newRow, int& newCol):

If valid direction command passed, return true and indicate the new position if player moves in that direction, else return false

Gameplay:

void Game::play()

*display first level*

*take first command*

*repeatedly, while the player has not quit*

*if player is asleep*

*skip executing player command and reduce player sleep time*

*randomly, with 1/10 chance, regain player’s health*

*if direction command*

*if player walking into a wall*

*do nothing*

*if player walking into a monster*

*attack monster, update result*

*kill monster on dungeon if needed*

*if player moving to a valid position*

*change player’s position*

*if pick up command*

*if player is at the idol*

*Indicate game won, update result*

*if player at some object*

*attempt picking up object, update result*

*else do nothing*

*if display inventory command*

*display player inventory, and get some command to exit*

*if wield weapon command*

*display player inventory*

*take command from player*

*try to wield item from inventory, update result*

*if reading scroll command*

*display player inventory*

*take command from player*

*try to read item from inventory, update result*

*if cheating command*

*make player cheat by giving player stat boosts*

*if go downstairs command*

*if player is on stairs, proceed to next level*

*try moving the monsters on the dungeon, update result*

*if game won or player dead*

*display updated dungeon and result*

*wait for player to give quit game command*

*quit game*

*display updated dungeon and result*

*get next command*

**Supporting Class Structures**

* The other classes in the program are summarized in the diagrams.
* The functions in these classes are broadly divided as follows
  + Accessors: Allow other classes to get private data member values. Trivial
  + Setters: Change to private data member values after ensuring validity. Trivial
  + Game Functions: Provide the functionality required for game play. Mostly non-trivial
  + Helper Functions: Support functionality of class implementations

**Dungeon Class**

* Each object of a Dungeon class represent a level in the game and holds all the relevant actors and objects
* An object of Dungeon class has
  + number rows and columns in the dungeon
  + level of the dungeon
  + the goblin’s smell distance
  + A vector with pointers to GameObjects on the dungeon
  + A vector with pointers to Monsters on the dungeon
  + A pointer to the player
  + A base grid of characters that holds the room structures (where the walls are) and the stairs or idol depending on the level
    - Dungeon Class uses simple Room structures that hold a room’s height, width, and coordinates to its top left corner to help generate rooms on the grid. These structures are not used anywhere else
* In a game, a dungeon can be used as follows
  + It can be displayed
  + Remove an object from a dungeon
  + Kill and remove a monster from a dungeon by calling each Monster’s dropDead()
  + Move the monsters around on the dungeon by calling each monster’s move() or have them attack the player using Actor’s attemptAttack function
  + Find out if at a given spot, there is a Monster, Object, Stairs, Idol, or Walls

**Dungeon Class Non-Trivial Functions**

Constructor:

Dungeon::Dungeon(int level, Player\* player, int goblinSmellDist,int rows ,int cols)

*Set level, rows, columns, and goblin smell distance*

*generate rooms*

*randomly decide number of monsters depending on level*

*add as many random monsters to the dungeon*

*add stairs or idol(level 4) to the dungeon base grid in a valid position*

*add between 2 or 3 random objects to dungeon*

*add player to dungeon*

Helper Functions:

* Dungeon::produceEmptyPos(int& row, int& col) const

produce a position that’s totally empty (can’t be the location of an object)

* Dungeon::produceEmptyPos(int& row, int& col) const

produce a position that an actor can occupy (can be the location of an object)

* Monster\* Dungeon::monsterAt(int row, int col) const

If monster at (row, col), return a pointer to it, else return nullptr

* GameObject\* Dungeon::objectAt(int row, int col) const

If object at (row, col), return a pointer to it, else return nullptr

* GameObject\* Dungeon::addNewObject()

On a random, valid pos. on the dungeon, add a new dynamically allocated object

* Monster\* Dungeon::addNewMonster()

On a random, valid pos. on the dungeon, add a new dynamically allocated

monster depending on the level of the dungeon

* bool Dungeon::addPlayer(Player\* p)

Give the player a random, valid pos. on the dungeon and link player and dungeon to each other

void Dungeon::generateRooms()

*Set entire base grid to be walls*

*Randomly pick the number of rooms*

*Depending on the number of rooms, decide maximum height and width of a room*

*Repeatedly:*

*decide a width and height for the room keeping in mind the max.*

*accounting for previously generated rooms, decide bounds for new upper left corner so that new room shares apart of a wall with the prev room*

*decide random upper left corner for the new room in the bounds*

*using the upper left corner, width and height, clear a rectangle on the base grid to be the new room*

*looking at the right wall of the previous room and the left wall of the new room, randomly decide a shared neight*

*clear a corridor between the rooms at that height*

*Update the previous room to be the newly generated room*

Game Functions:

* bool Dungeon::removeObject(GameObject\* object)

remove object from the dungeon if there such an object on the dungeon

void Dungeon::display(string result) const

*create a display grid and copy onto it the base grid (with rooms and stairs/idol)*

*add all objects on the dungeon to the display grid*

*add all monsters on the dungeon to the display grid, on top*

*add player to the display grid*

*clear screen*

*print display grid*

*print stats*

*print result (from the last move)*

bool Dungeon::killMonster(Monster\* monster)

*find monster in dungeon*

*if monster is dead*

*if the monster drops any items on death, add them to the dungeon*

*remove monster from the dungeon and deallocate it*

*return true*

string Dungeon::attemptMoveMonsters()

*for every monster in dungeon:*

*if monster asleep*

*do nothing and reduce sleep time*

*else if monster is next to player*

*attack player, update result*

*else*

*let the monster move*

*return result string*

**Actor Class**

* Each Actor has:
  + Position(row, col)
  + Stats
  + A pointer to a weapon to use
  + A pointer to a dungeon
* Each Actor can:
  + Return a unique name and symbol (purely virtual functions to make Actor class abstract)
  + try attacking another actor
  + Take damage and die
  + Be asleep and reduce sleep time
* This class is further subdivided into monsters and the player

**Actor Class Non-Trivial Functions**

Game Functions:

string attemptAttack(Actor\* defender)

*Make result to reflect that actor tired to attack defender using weapon*

*determine attacker points from actor and defender points from defender to determine if the attacker hits the defender*

*if the actor successfully hits defender*

*decide damage done depending on actor and defender*

*do the damage to the defender*

*if defender is dead*

*update result accordingly and return*

*do the weapon’s special moves, if any*

*update the result for a successful attack and return*

*else*

*update the result for a missed attack and return*

**Player Class**

* This class is derived from actor
* A player is constructed with a nullptr for dungeon since the player can exist outside a dungeon so that it may go to multiple levels, and a ShortSword as a weapon
* Each Player has:
  + Max Hit Points
  + Vector of pointers to GameObjects which is the Inventory
* A Player can, beyond what it inherits from Actor
  + Display inventory
  + Pick up a GameObject and add it to inventory
  + Read a scroll from inventory
  + Wield a weapon from the inventory
  + Regain health
  + Cheat

**Player Class Non-Trivial Functions**

Game Functions:

* string Player::attemptPickUp(GameObject\* object)

add object to Player’s inventory if space is available.

Return result of attempted pick up

* void Player::displayInventory() const

Clear screen and display the player's inventory with small letter char keys

* string Player::attemptWieldWeapon(char key)

If given key is a valid key in Player’s inventory and corresponds to a weapon, change the Player’s weapon to it.

Return result of attempted weapon wielding

* string Player::attemptReadScroll(char key)

If given key is a valid key in Player’s inventory and corresponds to a scroll, apply the scroll’s effect to the player and remove it from inventory.

Return result of attempted scroll reading

* void Player::cheat()

cheat by giving player some stat boosts (max health and current health to 50, dexterity to to 9)

**Monster Class**

* Monster is derived from actor
* Each monster has a unique function and can drop a unique object on death
* Monster is specialized into
  + Snakewoman:
    - constructed with magic fangs as weapon
    - move by calling the dumbSmellMove() function with a smell distance of 3
  + Bogeyman:
    - constructed with short sword as weapon
    - move by calling the dumbSmellMove() function with a smell distance of 3
  + Dragon:
    - constructed with long sword as weapon
    - don’t move
  + Goblin:
    - constructed with short sword as weapon
    - make a smart move, described below

**Monster Class Non-Trivial Functions**

Helper Functions:

* int Monster::idealShortestDistToPlayer() const

Returns the ideal shortest number of steps to player, ignoring any obstacles like walls or monsters

void Monster::dumbSmellMove(int smellDist)

*Finding shortest distance to player*

*if player outside smellDist or monster right next to player (since monster is supposed to attack if right next to player)*

*do nothing and return*

*// Exploring horizontal direction of movement*

*getting the position next to monster in the horizontal direction and closer to the player*

*if the monster can move to this new position*

*move monster there and return*

*else*

*// Exploring the vertical direction of movement*

*getting the position next to the monster in the vertical direction and closer to the player*

*if the monster can move to this new position*

*move monster there and return*

Game Functions:

* GameObject\* Goblin::dropDead()

If Goblin is dead, it returns (a newly dynamically allocated) magic axe or magic fangs of sleep with 1 in 3 chance where it dies if the position is free, else return nullptr

* GameObject\* Dragon::dropDead()

If Dragon is dead, it returns (a newly dynamically allocated) randomly selected scroll where it dies if the position is free, else return nullptr

* GameObject\* Snakewoman::dropDead()

If Snakewoman is dead, it returns (a newly dynamically allocated) magic fangs of sleep with 1 in 3 chance where it dies if the position is free, else return nullptr

* GameObject\* Bogeyman::dropDead()

If Bogeyman is dead, it returns (a newly dynamically allocated) magic axe with 1 in 10 chance where it dies if the position is free, else return nullptr

**Goblin Class Non-Trivial Functions**

* The goblin class uses a helper structure point which stores the coordinates of a point in the dungeon
* The goblin class also uses an additional helper grid which is a grid of integers that runs parallel to the dungeon. This grid keeps track of if the goblin can go to a point (denoted by 0), if a goblin can’t go to a point (denoted by 1) and if the goblin has already visited a point (denoted by 2)
* Move as described below

Helper Functions:

* void Goblin::initializeHelperGrid()

Initializes the helper grid to have 0 in the spots where the Goblin can go and 1 where the Goblin can't (other monsters and walls).

* Goblin::Point Goblin::dirToNewPoint(int dir, Point start)

Produces a new point if a step is taken from start in the given direction.

int Goblin::shortestPathFrom(Point start, int maxSteps)

*find ideal distance of start (in steps) to player, ignoring obstacles*

*if the player is outside the Goblin’s smell range*

*mark this point as visited // so it is not considered again*

*return invalid path indicidation*

*if the ideal to dist. to player is greater than max steps to be taken*

*return invalid path indicidation*

*if the ideal dist. to player is 1 step*

*return 1*

*mark the starting point as visited*

*get all the points next to the starting point*

*for each point next to the starting point*

*if the point is valid and not already visited*

*if a shorter path to player has been previously discovered*

*set max available steps to that path length*

*else*

*set max available steps to maxSteps - 1*

*find the shortest path length from this point to the player in max available steps*

*if there is a valid and shorter path from this point*

*note this path length as the new shortest path*

*unmark starting point as visited //to let other paths to consider this point*

*if no valid path found*

*return invalid path indicidation*

*else*

*return 1 + shortest path length found*

Game Functions:

void Goblin::move()

*for each direction of motion*

*if the goblin can move in that direction and the player is not outside the goblin’s smell range after moving there*

*initialize Helper Grid*

*find length of the shortest path to player if goblin moves in this direction*

*if the new length is valid and shorter*

*update the shortest length and best direction*

*if no valid path found*

*do nothing*

*else*

*move the Goblin in the best direction as determined*

**GameObject Class**

* Each object has a position and dungeon, and also can give a symbol and unique name (purely virtual to make the class abstract)
* They dropped on Dungeon, or carried by Actor. The drop implementation is in the Dungeon addNewObject() and Monster dropDead(), and the pick up function is in player attemptPickUp().
* It is further divided into Scrolls and Weapons

**Scroll Class**

* Scrolls all share a symbol but have unique names
* Each scroll, when read, implements a unique effect on the player and has a string describing this effect
* Scrolls are further specialized into
  + TeleportationScroll
    - Teleportation Scroll can only be dropped by a Dragon, and the implementation of Dragon’s dropDead() controls this
  + ArmorScroll
  + StrengthScroll
  + HealthScroll
  + DexterityScroll

**Scroll Class Non-Trivial Functions**

Game Functions:

* void TeleportationScroll::haveEffect(Player\* user)

The user is randomly moved to another place in the level that is not occupied by a wall or a monster.

* void ArmorScroll::haveEffect(Player\* user)

The user's armor points are increased by a random integer from 1 to 3.

* void StrengthScroll::haveEffect(Player\* user)

The user's strength points are increased by a random integer from 1 to 3.

* void HealthScroll::haveEffect(Player\* user)

The user's maximum hit point value is increased by a random integer from 3 to 8. This scroll does not affect the player's current number of hit points.

* void DexterityScroll::haveEffect(Player\* user)

The user's dexterity is increased by 1.

**Weapon Class**

* Weapons all share a symbol but have unique names
* Each weapon has a unique action string to be displayed when used to attack an actor
* Each weapon has a dexterity bonus and damage amount
* A weapon may have a special move, but most don't
* Each actor is constructed with a specific weapon
* Weapons are further specialized into
  + ShortSword
  + LongSword
  + Mace
  + MagicAxe
    - Not placed in the dungeon initially, only dropped by dying monsters. This feature is implemented through dropDead() of Monster class
  + SleepFangs
    - Not placed in the dungeon initially, only dropped by dying monsters. This feature is implemented through dropDead() of Monster class
    - Are the only weapon with a special move

**Weapon Class Non-Trivial Functions**

Game Functions:

* string SleepFangs::specialMove(Actor\* attacker, Actor\* defender)

Puts the defender to sleep with a 1 in 5 chance, if the new sleep time is greater than the defender’s older sleep time

Returns result of special move