**Recursive Goblin Function:** Coord smellPath(Coord current, Coord\*maze[][LEVEL\_COLS]) const;

Location: “Goblins.h” & “Goblins.cpp”

Note: I was not able to implement/finish the recursive function, so in my current game, Goblins just uses the same function as Snakewomen and Bogeymen, but has their “smell distance” set to 15.

Pseudocode:

Coord Goblins::smellPath(Coord current, Coord \*maze[][LEVEL\_COLS]) const

int r = current.r();

int c = current.c();

int d = current.d();

char dir = current.dir();

char directionToMove = dir;

int p\_r = dungeon()->player()->getRow();

int p\_c = dungeon()->player()->getCol();

if(distance at maze[r][c] is greater than the distance of the current Coord)

set maze[r][c] to point to the current Coord;

if (r == p\_r && c == p\_c)

return the Coord pointed to by maze[r][c];

if (distance at maze[r][c] is less than the distance of the current Coord OR d >

m\_smellDistance)

return a Coord with a very large distance and direction set to ‘f’

Coord outputCoord = Coord(r, c, 100000, 'f');

if (you are at the original position)

if (no Walls in ARROW\_DOWN direction)

Coord temp = smellPath(Coord(r + 1, c, 1, ARROW\_DOWN), maze);

if (valid move and temp distance is less than output distance)

set temp as the new output Coord

if (no Walls in ARROW\_UP direction)

Same code as above, but ARROW\_UP

if (no Walls in ARROW\_RIGHT direction)

Same code as above, but ARROW\_RIGHT

if (no Walls in ARROW\_LEFT direction)

Same code as above, but ARROW\_LEFT

else

Same as code in “if” statement, but set direction to “dir” instead of

original directions

return outputCoord;

I wanted the code to travel all paths and make each grid point have the shortest distance and original direction that was taken for the shortest distance as the output coordinates. The table to the left shows a simplified version (distance only) of what I hoped to achieve.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 1 | 2 | 3 |
| 1 | Start  (0) | 1 | 2 |
| 2 | 1 | 2 | 3 |
| 3 | 2 | 3 | End  (4) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
|  | Start  (0) | 1  Right |  | 5  Right | 6  Right | 7  Right |  |
|  | 1  Down | 2  Down | 3  Down | 4  Right |  | 8  Right |  |
|  | 2  Down |  | 4  Down |  |  | 9  Right |  |
|  | 3  Down |  | 5  Down |  |  | 10  Right |  |
|  | 4  Down |  | 6  Down | 7  Down | 8  Down | 9  Down |  |
|  |  |  |  |  |  | Down  (10) |  |
|  |  |  |  |  |  |  |  |

The table above shows something more similar to the project. Although the first few could be either down or right, at the end, the direction should be down since the down direction’s path to the end is shorter.

**Class Game**

* Class Game is the overarching class which controls the game. It only contains an array of Dungeon pointers (Dungeon\* m\_dungeon[5]--one for each level), which allows it to manipulate the game.
* The class will not make an exit on the fifth level. It will instead make an idol
* When destructed, Game deletes all dungeons

Most of the implementation to create the dungeon is in the Dungeon class; however, Game will call the functions that add monsters, items, exit, and the player to the game (void addLevel()), as well as move all the monsters and players for each turn (void play()). When each level is initialized, it will also place random monsters and objects (char randomMonster(int level) and string randomObject())

**Class Dungeon**

* Class Dungeon is much more complicated than Game. The monsters, items, and player all “belong” as variables of Dungeon, or, at the very least, point to Dungeon.
* The rooms in the dungeon are initialized in a somewhat random manner. There are always 5 rooms, and if the room is short (<(2\*LEVEL\_ROWS/3 - 2) / 2), they the rooms will stack. Pathways are also initialized in the constructor of Dungeon.
* The functions to add game objects, actors, and exits are all located within Dungeon. Items always “belong” to the Dungeon, even after they have been used or the monsters that wielded the weapon have been killed. The items are set to “picked up” if they are held by an actor, and scrolls are set as “can’t read” if they have already been used.
* The print function only outputs items that have not been picked up by an actor.
* There is a separate constructor for levels 1-4, because player must retain all items.
* Dungeon deletes all actors and items when it is destructed
* The exit is made recursively, and the rooms and pathways are made such that smaller rooms will be stacked, but not overlapping

**Class Actors**

* Actor is an abstract class that is the base class for Monsters and Player. It contains many functions to get and manipulate hit points, armor points, dexterity points, ect.
* Players can switch Dungeons (which occurs when you add a level)
* The nullptr cannot be passed to Actor, as it will cause the program to immediately terminate
* All actors spot are initialized to not stand on another actor or on a wall. Actors cannot be initialized in pathways, which is a personal choice since I did not like how it looked to have actors initialized in pathways.
* The initialize spot function is also called by the scroll of transportation and the constructor of Dungeon to move the player to an open spot when a new dungeon is made.
* The code of Actors is very straightforward, so there are not many comments

**Class Player**

* Player is a bit more complex than Actors. It contains an array of items (m\_inventory), which point to different items in the array of items in the dungeon. There is also a pointer (m\_currItem), which points to the item the player is currently wielding.
* The printInventory function takes the number of items held by the Player that can still be read (in the case of scrolls) and all weapons the Player has picked up. One can select an item and the Player will make that item his/her m\_currItem if it is a weapon, and will read the item if it is a scroll. If the item is the golden idol, it will print the phrase when the game has been won.
* The copyItems function will add items in the Players inventory to the total array of GameObjects in Dungeon.

**Class Monsters**

* Monsters sets a standard function of movement for all Bogeymen, Snakewomen, Goblins, and Dragons. Goblins was supposed to have a separate function for move, but I was unable to figure out the recursive function. The movement function takes the “smell distance” and checks if the player is within the proper distance using the playerNearBy() function. The playerNearBy function moves towards the player—the greater distance in the x or y direction is prioritized for the direction of movement, but if that movement is impeded by something else, it will move in the other direction.
* Monsters will attack if the player is next to it
* Monsters will drop items when they die
* Monsters is an abstract class
* DERIVED CLASSES: all derived classes set their own initial hit points, armor points, smell distances, ect. They all use the same move function. Goblin isn’t supposed to use the same move function, but as I mentioned earlier, I wasn’t able to finish the recursive function. Dragon also uses the same move function, but its smell distance is set to 0.
  + Class Bogeymen
  + Class Snakewomen
  + Class Dragons
  + Class Goblins

**Class GameObjects**

* GameObjects have several common features. They can be picked up, dropped, and cannot be initialized in a wall. Since Dungeon doesn’t contain separate arrays for GameObjects held by no one, monsters, and player, there are also function which will set the object as hidden, or allow it to be picked up. All GameObjects have an Actor pointer; however, only those that are picked up point to an Actor. The others point to the nullptr, which is how my function determines if someone is holding an object.
* A GameObject cannot be picked up if someone is already holding it.

**Class Weapons**

* Weapons are derived from GameObjects and have functions that can be used to attack actors.
* Weapons are an abstract class
* The actionString function will attack the victim and output the complete output string, which includes the result (HITS or MISSES)
* Weapon actions use a switch statement that is determined by what type of weapon that will be used.
* The code is pretty straightforward, so I didn’t comment as much
* DERIVED CLASSES: only MagicFangs had a different attack function, which required a different output string. Since MagicFangs can be used to both attack and put an Actor to sleep in one move, Magic Fangs calls Weapons’ attack function as well.
  + Class ShortSword
  + Class LongSword
  + Class Mace
  + Class MagicAxe
  + Class MagicFangs

**Class Scrolls**

* Scrolls is an abstract class which contains functions regarding whether a scroll can be read
* DERIVED CLASSES: each derived class has a different read function. Since only players can use scrolls, anytime it is read, it will affect the player.
  + Class SoEnhanceDexterity
  + Class SoImproveArmor
  + Class SoEnhanceHealth
  + Class SoRaiseStength
  + Class SoTeleportation

**Class Idol**

* Idols do not have any function except for the fact that they can be picked up. Their actor is automatically set to nullptr since idols are always placed on the Dungeon

**Known Bugs:**

Goblin move function.

I have to press enter after ‘q’ for the game to quit.

I am not too certain about some of the outputs. The exe file that I downloaded seemed buggy—I couldn’t read some scrolls nor did the cheat function always work