Kyle Han
William Dang
Vincent Duong
Derrick Lin

Hell: The escape

Table of Contents

General Game Overview

Game Description

You are a skeleton who has been condemned to the depths of hell. However, hell just isn't to your satisfaction, so you're escaping back to earth. Explore the path, defeat enemies, collect organs, and set yourself up to become a real person again. Succeed, and live a life of

comfort back on earth. Fail, and become an unsuccessful ghost, forever stuck in the land of inbetween, neither satisfied on hell or reborn on earth.

Game Design

Floor themes:
F0: Hell
F1: Mystical Gem Cave
F2: Regular Cave
F3: Earth Forest.

There are 4 floors for the player to travel through. Each floor has 25 steps, where each step could result in a random enemy encounter, a random event, or nothing. Where these events occur are random and are generated by floor.cpp/floor.h, which generates the tiling via the constructor function. How many of each event occurs is controlled by the 2D array in floor.h `int floorSettings[4][4]`.

On the tile map, a 0 is considered our do nothing. This event is used as a storytelling device, and is meant to pull lines from F_Comments.txt.

On the tile map, a 1 is considered as an enemy encounter. When a user lands on this space, the playerCombat function (In main.cpp) should be executed. Random enemies are pulled from F_Enemies.txt.

On the tile map, a 2 is considered as a random event. Here, we pull a line out of the file F_RandomEvents.txt. Each line can adjust either HP or balance. As a signed integer, it could either increase or decrease each value.

The bulk of the program is in main.cpp, where we have the shop and the combat scenes. The shop can be accessed by entering the [m]enu, and then entering 3. This option is not available while the user is in combat.

The shop has items that can be brought an unlimited amount of times. These items affect both hp, max hp, damage, and balance. We decided that the game was too hard without the ability for the user to upgrade their own hp - both max and the HP itself.

Though the project description required a difficulty selector, we found that our game was hard enough without it, and didn't really make sense, given the type of game. Thus, we commented the whole difficulty selection out, and removed it from our code.

Master Program/Main Menu

Classes

Enemy

The enemy class holds the information in regards to all enemies fought within the game. It handles the opening, reading, and management of enemies' healths and such.
Enemies can be added to the F_Enemies.txt files using the following format:
`[Max HP] [Current HP] [Attack Damage] [Attack Multiplier] [Name]`

Attributes:

Active Attributes	Randomization Attributes
Name maxHealth healthPoints attackDamage Level enemyFileName enemyList	listOfNames listofMaxHP listOfHP listOfAD listOfLevel selection

Constructor

Because enemy now extends player, the default constructor is actually player::player.

THe other default constructor within enemy is enemy::enemy(**const** enemy& e1), to which we add the following to, in order to declare the enemy's values:

```
{
   name = e1.name;
   maxHealth = e1.maxHealth;
   healthPoints = e1.healthPoints;
   attackDamage = e1.attackDamage;
   level = e1.level;
}
```

One constructor requires an argument of the current floor level, which can be obtained in scoreboard.cpp. This floor level will decide which enemy file to open, since there are 4 floors. If ever at all this enemy file is missing, the program will quit.

This constructor then reads everything from the file into a vector (getLineCount in enemy.cpp). It then choose a random line out of the file (using srand, and rand), and moves all the corresponding information into the active variables. (Variables that are in use and can be modified with the getters and the setters).

getLineCount()

Used to read in all the information stored in F_Enemies.txt. It is stored in the following format: [Max HP] [Current HP] [Attack Damage] [Level] [Name]

The first 4 are ints, and the last, is a string. This is all pushed into a vector, in the randomization attributes.

modHealth/modDamage

These are used to modify the health and damage of the enemy themselves. We really only use modHealth, since the enemies don't change their damage.

File Operations

The File Operations class holds all there is to know about writing to files. That includes saving/creating/loading the scoreboard and the player files.

Attributes:

sbFileName
sb_myFile
pFileName
pmyFile
gsbFileName
gsb_myFile

Constructor:

This constructor deals with opening, closing, and reading the player files. This class adds all the scores and the name of the players into a file. Then the file is sorted depending on the score from highest to lowest.

save2File: Saves player's name and scores into a file.

chooseFile: Asks the player to choose a file name to store the information in.

closeFile: Closes all the files.

scoreRank: Outputs the scores onto the file.

getNames: Gets the names of the players in the form of a vector.

getScores: Gets the scores of the players in the form of a vector.

Floor

This class is used to create the floor tiling and track the user's position throughout the game.

Attributes:

floorLevel: The current floor that the player is on

floorSetup: A vector that holds the information about the current tiling.

As a reminder:

0 = Do Nothing

1 = Enemy Encounter

2 = Random Event

currentPos: The current position of the player

totalPos: The total positions that are available. This can be edited to create longer levels.

floorsettings: A 2D array that will hold the number of times an event occurs. In the format {# of times Enemy Encounter}, {# of times Rand Event}

Constructor:

We need to be able to call this from the beginning of the game, whether a user is loading a file or creating a new one. Thus, we accept the arguments inpLevel and pos. These are just to set into the floor class to keep track of things.

Through this constructor, we also generate the tilings. There is a new tiling generated with every load of the game. Here, we use the attributes floorLevel, floorSetup, and floorSettings to generate the floor.

The program loops through the number of positions there are, adding 1s, 2s, and 0s depending on the floorSettings values. It then uses the shuffle function to randomize the order.

When this constructor adds a new enemy (Pushing 1 onto the vector), it'll also generate a new enemy (By way of enemy e1 (int floorLevel)), and then push this onto the queue. This was used in task 9.

takeStep: Increments the current position

(Task 0) Enemy derived class of Player Class

This task ended up being a troublesome one. Not to implement and understand, that was quite easy. `class enemy {}` was replaced with `class enemy: public player {}` and various repetitive elements, including `string name, int maxHealth, int healthPoints, and int attackDamage` were removed from the file. Everything worked out great!

(Task 9) Players fight enemies using queue

Kyle's Take on this:

This task is where the ugly head of a derived class started rearing its ugly head. The default constructor for player is as follows:

```
player::player() {
    name = "";
    maxHealth = 10;
    healthPoints = 10;
    attackDamage = 2;
    weaponName = "Hand";
    balance = 0;
}
```

This works fine for the player and the enemy as we have declared it in the past (Where a new enemy object is created every time a fight should begin. It makes sense to declare most things with a default value of unknown, since the functions following alter the data. For example:

```
enemy e1(p1Scoreboard.getFloor());
```

will cause the default constructor to run, but through the enemy(int floorLevel) routine, will replace each one of them with their proper values.

However, when adding an object to a queue, or even popping it off the queue, only the base default constructor (player::player) is run, since no parameters can be given. This none of the actual values besides the default values are loaded.

I tried to work around this by storing the queue in different areas, either main.cpp, or floor.cpp, where the floor generation occurs. I also tried to store in the queue the values themselves, in a queue with a struct as the date type. Thus, we would return the enemy object with the default values.

William's Take:

It was discovered that we needed to include

```
enemy::enemy(const enemy& e1) {}
```

which is what Kyle did. However, what Kyle did not realize was that e1 had values that could be modified, once the default constructor was loaded. Hence, the following:

```
enemy::enemy(const enemy& e1) {
    name = e1.name;
    maxHealth = e1.maxHealth;
    healthPoints = e1.healthPoints;
    attackDamage = e1.attackDamage;
    level = e1.level;
}
```

Player

The player class holds all information about the player itself, including name, weapon, HP, and attack.

Attributes:

Name
maxHealth
healthPoints
attackDamage
Balance
weaponName

Constructor:

This default constructor sets the default variables for the user. This includes the starting HP/MaxHP of 10, weapon of "Hand" with the damage of 2. The user also starts off with a default balance of 0.

Getters/Setters

Gets/Sets individual attributes of the player.

modDamage:

Modifies the damage of the user. Adds the input to the attackDamage of the user, and checks if the damage will fall below 0. If it does, then damage stays 0, since you can't heal the enemy.

Modbal:

Like modDamage, we accept an int as a modifier and add it to the users balance. From here, balance cannot fall below 0, so check for it, and correct it.

modMaxHP:

Using items in the shop, the player can modify their max HP. This will become essential as the player progresses through the game. Theres is no cap on this, and the item that modifies max HP is only positive, so we don’t have to do any checks here.

randomComments

The randomComments class creates a vector of lines read from the file F_Comments.txt. When called, we will get a random comment describing the environment that the user is in. The goal is to tell a story.

Attributes:

Lines: A vector that holds all the lines.
commentsFile: The file name that holds the comments.
Selector: The index to be adjusted by getRandom, and used in getEvent to access the lines.
Line: A variable used to temporarily hold values read from the file.

Constructor:

The default constructor accepts the floor level as an argument, and selects a random comment file with it. If not found, the program will exit. The constructor then calls getLineCount.

getLineCount: This function will push all the lines read from the file to a vector.
getRandom: Seeds the selector with a random selection between it and the size of the vector
getEvent: Returns the selected line. Hopefully will properly educate the user as to the environment.

randomEvents

The randomEvents class reads and holds information from a text file that stores all the random events that can happen over the course of the game. That includes heath changes, balance changes, and the lines of text.
Random Events can be added in F_RandomEvents.txt, using the following format. Space is the delimiter.

```
[HP Change] [Balance Change] [Description]
```

Attributes:

lines: vector of all the lines from the text file. absolute.
hpList: vector of all the hp changes from the text file. absolute.
balList: vector of all the balance changes from the text file. absolute.
eventsFileName: the name of the file. relative.
eventsFile: initializes the file variable. relative.
hpChange: int variable that stores the health change.
balChange: int variable that stores the balance change.
Selector: int variable to help with line selection.
line: string variable that stores the whole string line that is displayed to the user.

Constructor:

The default constructor accepts the floor level as an argument, and selects and opens a text file for the appropriate floor. If not found, the program will exit.

getLineCount: This function will push all the lines read from the file to the appropriate vector.
getRandom: Seeds the selector with a random selection between it and the size of the vector.
getEvent: Returns the selected line. Tells the user what has happened during their step.
getBalChange: Returns the balance change from the random event.
getHPChange: Returns the health change from the random event.

Scoreboard

The scoreboard class displays the users info such as current floor, difficulty, and the current position of the player while in game. Score is calculated by collecting organs throughout the game. Everytime user collects an organ, the corresponding value point of the organ is added to the user's score. Class is also used to continue progress of player if he/she wishes to continue a preexisting game

Attributes:

sbScore: Current score of the player
sbDifficulty: Current difficulty level but we took this out so this part is just for looks
sbFloor: Current floor of the player
sbPos: Current position of this player
sbName: Current name of player

Constructor:

Default constructor initializes the score, difficulty, floor, and position of the player.

setScore: input from score from save file
setDiff: input from difficulty from save file
setFloor: input from floor from save file
setPos: input from position from save file
getScore: get current score
getDiff: get current score
getFloor: get current floor
getPos: get current position
getName: get name of the player
addScore: function to calculate player's score depending on organs collected during the game
sbOut: function to print out all the information for the scoreboard

(Task 1) List data structure

This task reads the scores on the scoreboard file and puts it as objects in the list. A search function is also made so that the program asks for a name and it searches to see if the name exists on the list. Overall this task setups for the next two tasks as this list can be manipulated to get different desired results.

(Task 2) Sorting scoreboard

The list from task one is now manipulated in order to sort the list by score. First an operator overload is created that compares two scores on the scoreboard. Then after this operator overload is created the built in sort function from the list library can be used in order to sort the scores on the list.

(Task 3) Averaged scores

A function averageScoreboard, takes in a list of each players' scoreboard the average score of any repeated names are added to a list. Three variables are created which hold the average, sum of scores, and how many times the name repeats. A temporary list is created which holds the same values as the original list. The names in the temporary list are searched in the original list and if the name is found, the scoreboard is removed from the temporary list, the count is incremented and the score is updated accordingly. Once the average is determined, it is added to a new list. Once all elements in the temporary list are removed, the function is completed and a new list with all names and averages is created.

weaponNode

The weapons class here is actually kind of a misnomer. Originally made to be a weapons specific class, this should really be called items.

This node establishes the framework for the doubly linked list, created and maintained by weaponsShopv2.

Attributes:

item: The name of the item. Absolute.

damage: The damage of the weapon. Can be set to 0. Absolute.

cost: The cost of the weapon. We will take this amount of currency away from the user

maxHP: Affects the max HP of the user. Called in the shop when the user buys “Dying a lot? Use this! Potion”. Relative.

HP: Affects the HP of the user. Adds more hp. Relative.

Constructor:

Loads the class with the arguments provided.

This class is mainly used as a data type. Similar to pokeNode, pokeDex, and Pokemon Trainer, this class will create a doubly linked list that can be traversed whenever the user enters the shop.

Setters:

setNext: sets up the pointer to point to the next weaponNode

setPrev: sets up the pointer to point to the previous weaponNode

Getters:

getNext: Returns the pointer to the next

getPrev: returns the pointer to the previous

getItem: Returns the name of the item.

getDMG: Returns the damage of the item

getCost: returns the cost of the item

getMaxHP: Returns the max HP adjustment

getHP: Returns the HP adjustment

Print: Prints out all the different information, depending on whether or not the item was a potion or not.

weaponShopv2:

Weapons are stored in FXweaponList.txt using the following format:
`[Cost] [Damage] [Max HP Change] [HP Change] [Name/Description]`

Attribute:

Head, tail, current: Pointers to weaponNodes, corresponding to their position.
Flag: This flag is set true if there was an error reading the file.

Constructor:

weaponsShopv2: Establishes all the pointers as NULL
addWeapon: Adds a new weapon to the end of the linked list
moveToNext: Moves to the next item, if it exists.
moveToPrev:Moves to the previous item, if it exists.
moveToHead: Moves to the head of the linked list
moveToTail: Moves to the tail of the linked list.
printAll: Utilizes the print() function in weaponNode to print out all the entries in the linked list
deleteAll: Frees up memory space by deleting the entries.
purchaseProduct: Adjusts the user's balance, HP/MaxHP, weapon name/damage accordingly. Returns weaponNode to add to the stack of brought items.

Shop:

Constructor: Sets up the shop according to the floor itself. If no file is found, adjusts the user's stats to default. It'll then add everything read from the file onto the linked list.

runShop: The driver function for the whole shop. Uses the wasd keys to navigate and buy.

(Task 4) Shop list for weapons

The weapons and weaponsShop files have been revamped to include weaponNode, weaponShopv2, and shop. Together, these act like lab 8's pokemon trainer system. The weapons file contains information about the node itself, the weaponShopv2 creates the doubly linked list, and the shop is the overall manager of the linked list.

Just like lab 8, the user can move up and down the list, selecting which one to buy.

(Task 5) Shop tracker

Once the user has brought something from the shop (Or even if they didn't) under the purchaseProduct function within namespace weaponShopv2, that node will be pushed onto a stack called itemStack. Once the user presses A to return from the shop, the user will receive a printall function of everything that has been purchased.

This is done by checking to see if the stack is empty. If not, it will print out the contents of the popped element, and continue in a loop.

Main.cpp

The main file is where all the magic happens. We have several functions here that are essential.

WeaponsShop: Once again, a sort of misnomer. This is actually the item shop. It will read from the file every time (Originally, there was supposed to be a new weapons shop for every floor, but we forewent that because it didn't make too much sense).

The program reads all the items from the file weaponsList.txt, in the format:

```
[Cost] [Damage] [Max HP Change] [HP Change] [Name/Description].
```

And pushes it to a weaponsList vector, using the template weapon.cpp/weapon.h.

We then enter the continuous loop, where the user is presented with all the options from the file. We can compare the user's selection to the index of the vector, and extract all the necessary information needed to modify the user values.

We can check for potions by checking if the damage is 0. Potions should do nothing to damage, but they affect HP/maxHP of the user.

After various checks to ensure the user actually wants to purchase, we can adjust the user stats and move on.

enemyCombat: This interfaces the combat system from the enemy pov. Mods the health of the player depending on the dmg of the enemy and mods the health of the enemy depending on the AD of the player.

playerCombat: Whenever the user encounters an enemy, this function is called. It will provide the user interface and carry out the various instructions that the user has asked. Both examine the enemy and examine yourself call up on the overloaded operator << in the player and enemy files.

The users attack is measured to see if the enemy might die. If it does, we output some user interface and add 1 to the scoreboard (To keep track of the organs collected/enemies killed). This is another requirement that we changed: since we believe the user to "successfully" win the game when the skeleton has collected all the organs. Killing an enemy also mods the balance of the enemy based on the maxHP.

If the user does not kill the enemy, we will just user interface it and enter the enemyCombat phase.

Differing from the updated project description: Once again, we implemented a slight difference from the project description. The ability to run away should be removed, however, I believe it to be a part of the strategy here. Instead of completely removing this feature, we implemented a random chance of it working/not working. If the enemy has prevented you from running away, you must stay and fight. However, upon attacking the enemy, you have a renewed chance of running away. Thus forces the user to interact with the enemy, but still offers the user semi-control over the manner. It's probably the worst when a game's RNG decides that you MUST die, which is what removing the runaway feature will do, especially since the user is not allowed to enter the shop while in combat. We gave the run away feature a 60% chance of working.

playerOptions: The user interface the user can call when out of combat, essentially a pause menu. Allows the user to examine themselves, see the scoreboard, enter the shop, exit the menu, and save and quit the game. This function runs continuously until the user decides to exit the menu or decides to save and quit the game.

Game: This is the main game loop. If anything else got deleted, it would throw compiler errors, but if this got deleted, it would break everything about the game.

Each loop of the game is essentially a new floor. There is a sub loop inside, where every loop is a new step. Combined, this covers everything in the game. Therefore, at the beginning of each floor, we can construct all our classes with all new informations.

In the inner loop (while the current position is less than the max positions available), we ask for the user's input. Here,the user can either take a step or enter the menu to see more information. If they take a step, then we can ask the class floor.cpp for the tilemapping. From there, we can expand on what should happen.

As a reminder:
0 = Do Nothing
1 = Enemy Encounter
2 = Random Event

We then have a switch statement that will call different functions based on the tile mapping.
Do nothing will call the randomComment class.
Enemy Encounter will call on playerCombat. It will also use a constructor to generate a random enemy out of the floors enemy file.
Random event will call on the randomEvents class.
AlphabeticalScore: Function that pops up at the end of the game that alphabetically lists the scores of all the users that have played the game so far.

Main: The start of the program. This initializes all the setup functions - getting the user's save files ready and the scoreboard files ready. Once all the files are all prepared, it calls game().

(Task 6) Stack for enemies defeated

Created a stack called enemystack that stores the defeated enemies and their levels using pairs. To push an enemy onto the stack, we created a function called addtoenemystack() that passes the enemy class so we can grab the necessary attributes. Using make_pair(), we are able to store the two different data types into a stack of pairs. In this case, we are storing a string and int data type values. Because stacks store from bottom to top we need to reverse the stack in order to get the correct order of enemies defeated to print at the end. For that we created a separate function called reverseStack(stack<pair<string, int>>& st) that passes a stack of pairs of string and int. With this function we are able to reverse the stack and print the proper order of enemies. To print, we created a function called printenemiesdefeated() that calls reverseStack() and prints the data from the stack, as well as clearing the stack.

(Task 7) Side quests

In game(), we tell the user of their side quest on each floor using a switch statement. The switch statement input is the floor level and each floor has its own side quest. After each floor we run the checkQuest() function to check if the quest has been completed. If the quest has been completed we award the player with bonus points.

(Task 8) Combat log

Added two stacks called combatloghealthstack() and combatlognamestack() to print out the enemy name and enemy health for the combat log. Added functions called addtocombatlogplayer(player& p1) and addtocombatlogenemy(enemy& e1) to push names and health points of both player and enemy onto combat log stack. Also created functions called reverseStackname(stack<string>& st) and reverseStacknum(stack<int> &st) to reverse the hp and name stack so that it can be printed properly later on. These two functions can be used to reverse any int or string stack. Finally created a function called printcombatlog() that calls the reverse stack functions and prints out the stack in the correct order. Also pops all elements of the stack at the same time so that the stack is cleared.

Highlights

Random Events that can give or take health or money from the player. This randomness removes the predictability of the enemies and structure of each floor and gives the game a higher replay value than just having the game be a linear path from one floor to another. The player cannot die due to a random event, that would make the game too hard and unbeatable, instead the player is reduced to 1 health point.

Random Comments are just comments that are non-related to the game and do nothing to the player or the story. It spices up the dialogue of the game and allows the game to “break character”. It also allows the developers to write about anything they want and be funny.

Run Away feature allows the player to run away from certain fights. This feature was not supposed to be in the game, but we felt like we had to include it as we added a game of chance when using the option. It adds more strategy to the game, as the user can only use the option once until they attack once, this way the player is forced to attack the enemy instead of spamming run away.

Operating systems used:

Kyle Han: Windows 10, build 19042.1288
William Dang: Windows 11, build 22000.282
Vincent Duong: Windows 11, build 22000.282
Derrick Lin: Windows 10, build 19042.1288

Work each has done for midterm:

Kyle Han

Created floor structure system.
Implemented random selection of enemies, comments and random events.
Added shop for user to buy items in. Removed only weapons, added more ideas.
Added combat system for main.cpp.
Implemented organ collection system, utilizes the scoreboard to keep track of enemy kills.
Added user stepping and menu systems
Created text files for the program to collect comments, weapons, and others from.

William Dang

Implemented randomEvents class
Implemented randomComments class
Worked on fileOperations class
Implemented money/currency system
Worked on combat system
Worked on player save files.
Debugging and testing
Added to randomEvents, randomComments, and weaponList text files.

Vincent Duong

Debugging and testing
Fixed issues in the various menus of the game
Helped with the text files for comments
Scoreboard class
Weapons class
main.cpp

Derrick Lin

Debugging and testing
Implemented a scoreboard file where all the scores and players are stored.
Sorted the file by scores and alphabetically.
Main.cpp
Scoreboard class

Work done for final:

Derrick

Task 1 & Task 2 & Task 3

Kyle

Task 0 & Task 2 & Task 4 & Task 5 & Task 9

Vincent

Task 6 & Task 8 & Task 9

William

Task 6 & Task 7 & Task 8 & Task 9