Introduction

Group Members: Olivia Zhang and Angela Law
Project: CC3K+

Overview (describe the overall structure of your project)

The main function is the controller for the game. The controller owns the current floor and the player character. The controller starts the game and manages the turns. Each turn consists of reading from std in, calling PC's functions corresponding to the user's command, then calling the update method on the floor.

A floor consists of a vector of vectors of unique pointers to tiles. Tiles own enemies and items. That is, tiles have a unique pointer to an item and to an enemy. There are methods to ensure that no tile will ever have both an enemy and item at the same time. Tiles also have an enum to store what type they are (wall, door, moveable tile). The player character is not owned by the floor or a tile, but a floor has a shared pointer so it always knows where the player is. The floor calls the methods that generate enemies and items, updates the enemies to move/attack on each turn, and validates the user's input.

There are four distinct phases to the game: initialization, game play, transition, and ending.

1. The initialization begins when the program is run. The user is prompted to select a race, and the controller generates the player character. Next, the floor is generated. If given a command line argument, the controller reads and stores the first floor map as a string. This string is passed as a parameter to initialize the floor (in the Floor class), where tiles are identified and items and enemies are populated. If there is no command line argument, the controller reads a string from a set text file which stores the default map. In this case, the Floor class randomly generates the enemies and items. This occurs as Floor calls methods in EnemyFactory and ItemFactory, passing in a reference to itself. The controller only handles one floor at a time, and randomly generates a number to decide which floor the barrier suit will spawn on.
2. Game play involves the iteration of turns. Each turn begins with the user entering a command for the PC. If the command is invalid, the controller prompts the user to enter another command, everything else is unaffected. After entering and executing a valid command, the controller calls updateFloor(). This method iterates through all tiles in the floor. If the tile contains an enemy, the tile calls the Enemy class' methods to determine whether the enemy should attack or move. The floor is entirely responsible for moving the enemy. This is done by randomly generating a direction, checking if it is a valid tile to move on, and changing the ownership of the enemy to that tile. Then, the enemy is marked as hasMoved, and the floor finishes its iteration loop. This continues until PC dies, PC walks over stairs, or PC wins.
3. The transition phase occurs when the PC walks over stairs. The controller always checks before the PC moves if the next position is on the stairs. If this is the case, the PC's attack and defense stats are reset to erase the effects of temporary potions. The controller then generates the next floor exactly as in the initialization phase. The controller has a for-loop which iterates five times before the game is over, this keeps track of which floor the PC is currently on and when we are done.

4. The ending is one of two scenarios: PC dies or PC wins. Each turn, the PC’s HP is checked twice. Once before each turn to check if a potion killed the PC, and once after updateFloor() to check if an enemy has killed the PC. The PC wins when the for-loop in the controller is on the 5th iteration and PC reaches the stairs. The score is then calculated according to what race the PC is, and the win message is displayed.

Updated UML

Once we started coding, we realized that the items and enemies did not need to know their position on the floor. We handled this as items and enemies can only exist when they are owned by a tile, and every Tile object knows its coordinates on the floor grid. Accordingly, move for enemies is now managed by the floor, instead of being a function in the Enemy class. The floor is able to generate a random direction, check if the tile is a valid, empty tile to move onto, then switch the ownership of the enemy between the tiles.

We also realized that it was bad practice for the enemies and items to have an id integer to distinguish between them through the base class pointer. The original purpose of this was for printing the enemies out when we print the floor, however, we just added a pure virtual function getChar() for enemies and items that are overridden in each subclass for when we print the tiles. We omitted this from the UML because the “getChar()” and “getRace()” (for the PC) are not crucial to the implementation logic.

From a glance, our UML appears highly coupled around the State and Info structs. This occurred because when we implemented the pImpl idiom, many classes formed dependencies to these two structs. We realized that we could group gold and potions together into one class, Consumables, because the effect on the PC is the same. That is, we simply add all the fields of the consumable to the stats of the PC. For example, a smallHorde would have hp = 0, atk = 0, def = 0, and gold = 2. Thus, using a potion or gold involves the same arithmetic to the player’s stats.

Design (describe the specific techniques you used to solve the various design challenges in the project)

For the random generation of items and enemies, we use the factory method. Both Item and Enemy are abstract superclasses and have multiple concrete subclasses, however, the number of instances of each subclass on a given floor is determined (randomly) at runtime. To have this random generation occur separately each time the program is run, the EnemyFactory and ItemFactory provide an interface which will create these objects, so the floor class only needs to call this public method.

Given the special attributes of each PC race, items and each race should interact in different ways. To apply an item's effects on the PC, we must first determine both the PC’s race and the item type. For this, we use the visitor design pattern to implement double dispatching. This provides a way to call the right inherited method through the base class pointer. In the item class, we overload the function useOn, which takes a parameter to a player subclass, that is a human, elf, dwarf, or orc pointer. As such, we have four overloads for this function.

This allows us to implement the special attributes of each race, for example, before we use gold on a dwarf, we want to double its value, whereas for an orc, we half the gold amount we add. We also override the function useItem(Item *) in each race subclass so when we call useOn(this), the “this” pointer is not a base class pointer to Player; it is a pointer to a race subclass.

One known challenge with the visitor pattern is the lack of scalability. For each new PC race we add, we have to write another overloaded function *useOn(newRace *p)* in every item class. We also have to override useItem, which calls useOn in the race’s class. Nonetheless, there is no reasonable alternative to work polymorphically and implement this functionality.

<u>Specifics:</u>
Dragon and its treasure (DragonHorde or BarrierSuit, which we call DragonBaby) posed many challenges. When the DragonHorde gets spawned (⅛ chance) in ItemFactory, the Dragon which guards it must also be spawned. Moreover, a Dragon and its DragonBaby must know about each other to some extent: a Dragon attacks only when the PC is within 1 block of its DragonBaby, so it must know its position, and a DragonBaby (item) can only be used once its protector is dead. Since a Dragon must be able to tell its DragonBaby it has died, Dragon has a pointer to a DragonBaby. DragonBaby only needs a bool field which has the alive status of the Dragon protector. This is set up right after the DragonHorde is constructed in the ItemFactory. We first check that the randomly chosen tile has at least one valid neighbour tile for the Dragon, if so, then we construct a Dragon, passing in a pointer to the DragonHorde and the position of the DragonHorde in its constructor. This allows us to override the shouldAttack function in Dragon to look at the location of the DragonBaby relative to the PC, instead of its own tile position. This is different from the other enemies' shouldAttack. When Dragon dies, it calls a method in Dragon Horde that sets the hasProtector bool to false.

When the Tile is called to removeEntities, this function first checks whether it is removing an enemy. This function asks the Enemy whether it was holding a compass, then if it is supposed to drop a Merchant Horde. In either case, the Tile will make a new item that the Item pointer in tile now points to, then the Tile removes the Enemy. If a Merchant is selected to hold the compass, it will drop a compass instead of a Merchant Horde when it dies.

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

Adding another subclass for Enemy would be a simple process. Enemy inherits all its functionality from the Enemy superclass, so what’s left is to create a new Enemy subclass and write the default constructor to initialize the stats. Afterwards, include it in the EnemyFactory array so it would be generated.

Similarly, adding another item would require creating another subclass for Item. This subclass must have the useOn function overloaded for every PC race. This item can change any member field in Player. To add another potion or treasure is even simpler.

Additionally, our program is compatible with any floor layout of any size, as long as a numbered chamber map is also provided. To change the number of floors in our dungeon game is also a matter of changing one number in the main file (how many iterations the for loop will have).

If we wanted to alter the special abilities of the different PC race's, simply update the useOn function which takes a pointer to the race as a parameter.

If we wanted to generate items or enemies with a different probability distribution, we only need to change how many times each subclass occurs in the vector which contains the item/enemy types. Similarly, we could easily generate a different number of items and enemies, or also a fixed number of a certain subclass.

- New map would only need to add a new numbered map

Answers to Questions

1. How could you design your system so that each race could be easily generated?

We declared classes for Human, Dwarf, Elves, and Orcs that inherit from an abstract Player base class that inherits from an abstract base Character class. We will always reference the PC through a base class pointer of Player. Player has a public interface with methods that implement the shared behaviour of all the races (e.g. move, getAttacked). Character class contains a pointer to info struct containing ints for HP, ATK, DEF. Each of Human, Dwarf, Elves, and Orcs will be constructed with their character specific stats by passing in a Info struct to a protected player ctor that initializes player and character fields. Races can be generated using the Factory Design Pattern because we don't know which player will be generated until runtime and a valid input is given. A player factory that has a method that takes a character representing the player type, and returns a corresponding player subclass object

2. Additionally, how difficult does such a solution make adding additional classes?

This is quick. Polymorphism is used in this simple inheritance; the base class pointer to a Player ensures we don't need to change the abstract Player superclass to implement a new subclass race. We do not need to change any floor functions and controller access to player because they have a shared pointer to the player base class.To define the new behaviour of the new race, the added class inherits from Player and overrides the applicable functions in Player's public interface (useItem). The new race will inherit the shared behaviour of the player class. We need to write a new constructor for each subclass of Player which gives us default stats for each race. We also need to add the option to generate that character in the playerFactory class.

3. How does your system handle generating different enemies?

We created an abstract Enemy Class, which inherits from the abstract Character class. Enemy species (vampire, werewolf, etc..) is a subclass that inherits from the Enemy class. The enemy class has virtual implementations of functions that have the default behaviour for an enemy subclass, but the subclasses can optionally override to implement special attributes. For example, default attack is implemented in Enemy class, and overridden in Merchant class (to only attack when hostile).

To generate the enemies, we will use a Factory Design Pattern. This is because we don't know which enemies to generate until run-time, where they are randomly generated with different probabilities. All enemies will all be accessed through the Enemy base class pointer. Our Floor Class calls the class EnemyFactory to generate 20 enemies randomly. The Enemy Factory contains a vector of 18 enum enemy types that correspond to the enemy we want to generate. The vector represents our probability distribution (ex. a werewolf has prob 2/9, so the vector contains 4 werewolf types). The enemy factory randomly generates 20 numbers corresponding to an index in the vector, then constructs an enemy pointer by calling the corresponding enemy subclass constructor to the enemy type. Enemy factory takes a reference to the Floor object. The enemy factory then randomly generates a valid position to a tile on the floor and moves the enemy pointer to that tile on the floor. Thus the tile now owns the enemy pointer.

4. Is it different from how you generate the player character?

We reference both through pointers to their abstract Superclasses (base class), that is the Player and Enemy classes. The player race is given by the client, which will input their choice through StdIn, which we take in through a controller, which eventually initialises our PC. We only have one player character. Both instances use the factory design pattern and are generated at runtime.

The player factory receives a char representing which race to construct and returns a Player pointer that points to a concrete player subclass. On the other hand, in addition to constructing an enemy race based on input and returning an enemy pointer, the enemy factory has another method that receives a reference to the floor and populates the floor with all the enemies according to a random probability distribution.

5. How could you implement special abilities for different enemies? (For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?)

The only two "different" abilities we will be considering are Dragons (which do not move), and Merchants, which only attack when hostile. For dragons, we will override the virtual move and attack methods from character. If move is called on an Enemy pointer which is a dragon, the dragon's move will not change its coordinates (dragons are always stationary). Dragon's will only attack if the PC is within one block of the DragonHoard. Dragon has a pointer to its DragonHoard, thus the attack is different from the other enemies' attack. For Merchants, we will have an additional static field in the Merchant class only, a boolean called hostile. Only when this is true will Merchants attack the PC, there, this is also different from the other enemies' attack. The Merchant class override the attack method in the Character class, so it can check if the Merchants are hostile

6. What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

We could use the Decorator pattern, but we're choosing not to. We could have an abstract class, Stats, which has pure virtual methods like: getAtk, setAtk, getHP... The simplest concrete instance of this is BaseStats, which stores the default HP, Atk, and Def in fields for a given race. We write implementations for our methods, returning the int in the corresponding field. The Decorator class has a pointer to Stats and overrides the virtual methods. We decorate BaseStats with the temporary effects of potions, overriding the methods again. For example, a potion that increases attack by 5 would look like:

```
int getAtk { return component->getAtk + 5; }
```

We believe that we can model the temporary effects in a simpler way. Similar to the Decorator implementation, our PC has a struct called info to store our HP, Atk, Def and gold stats. Instead of decorating stats, we will save the initial, default HP, Atk, Def and gold stats in another Info struct. We can then directly add or subtract the potions effects when using the potion. When we move to a new floor, we reset info->Atk to be the original default Atk, and do the same for Def. For potions that affect HP, we directly add/subtract the potion effects and compare it to the default HP to ensure races don't go over their max HP.

7. How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

Similar to enemies, our PC and tiles will interact with all items through the Item abstract superclass. Item has a public interface with pure virtual method *useOn*. We use the visitor patter to overload *useOn* to take a pointer to a player subclass (Human, Elf,etc). So we declare a pure virtual method "useItem" in the Player class that takes an item as a parameter. Then in the Player subclasses, we implement "useItem" by calling "item->useOn(*this)".

We organised the Item subclasses as such: Compass, an abstract DragonBaby class (for the BarrierSuit and DragonHorde) and Consumable (for gold and potions). To implement the effects of using each item subclass, we override each overloaded method useOn for each Player subclass. This will allow us to implement different item effects for different Player subclasses. For instance, potions and gold both affect a player's info, so one of: HP, Atk, Def, or gold. We will add all fields of the info struct in a potion or hoard to the player's info (Eg. a +5 Atk potion would have the fields HP = 0, Atk = 5, Def = 0, and gold = 0). We will have a base abstract item class that all items inherit from. The base item class has an identifier field and a use() pure virtual method and the concrete item subclasses implement the use() method. Thus, Tiles can own an item and players can use an item without knowing exactly what the item is.

We randomly generate items using a ItemFactory class, which contains a generatePotions and generateTreasures wrapper method that calls a generateItems helper method. We have wrapper functions because we want to generate 10 items and 10 potions, each having different probability distributions. Thus, the generateItems method takes a reference to the floor, a vector of enum item types representing the probability distribution, and the number of items we want to generate for that distribution. Similar to the enemy factory, the ItemFactory randomly generates numbers corresponding to an index in the vector, then constructs an item pointer by calling the corresponding item subclass constructor. If the item type is a consumable (e.g. potions and treasures), the consumable ctor takes in an item type and is constructed with the corresponding stats/info. Item factory takes a reference to the Floor object and randomly generates a valid position to a tile on the floor and moves the item pointer to that tile on the floor. Thus the tile now owns the item pointer.

8. How could you reuse the code used to protect both dragon hordes and the Barrier Suit?

We will have a. abstract parent class “DragonBaby” that inherits from the Item class, since it does not implement the pure virtual “useOn” methods. The Dragon class has an aggregation relationship with DragonBaby. So Dragon will have a pointer to a DragonBaby object, but the Dragon does not own the DragonBaby. When the Dragon object is destroyed, the DragonBaby object is not and exists independently outside of the Dragon class. When the Dragon object dies, it sets the boolean “hasProtector” in DragonBaby to false. Its subclasses (DragonHorde and BarrierSuit) will also inherit methods to get the “hasProtector” member in the DragonBaby parent class. In addition, they will implement the pure virtual function useOn(Player *p), and other shared logic with the item superclass. The implementation will contain class specific logic to actually use the item. In the Dragon class, the dragon object will simply protect a DragonBaby object, and the Dragon object does not need to know what exactly the DragonBaby is, allowing the protection code to be reused for BarrierSuit and DragonHorde.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

We decided to try to complete the entire project without explicitly managing memory via vectors and smart pointers. Thus we did not write any dtor bodies and we do not have any delete statements in our program at all. We have some raw pointers for cases when we decided that there is no ownership, such as a dragon pointer to a dragon baby (rather, tile has ownership over the dragonbaby). We also adhered to RAII principles and avoided using the new keyword and instead used “make_unique”.

We came across many challenges with using smart pointers. For one, there was a bit of a learning curve with unique pointer syntax. Oftentimes our program would not compile because of the slightly different behaviour between a raw pointer and smart pointers (e.g. unique pointers). We also had issues with transferring ownership between unique pointers using various unique pointer functions (i.e move, release, reset). We solved this issue by reading the documentation for unique pointers and explicitly what each function did.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

This project taught us the importance of coordination and communication. Since we collaborated using Github, if we worked on the same part of the code at the same time, it would be redundant and we would have a merge conflict. We discovered what worked best for us was working on the code base at separate times and then communicating what we accomplished after each work session. This allowed for breaks to work on other things, discussions/input from a different perspective, and minimal merge conflicts. In addition, it was very important to communicate what part of the code we were working on, and tackle different problems. Thus we could reduce the probability of changing the same piece of code. We also tried to group similar tasks together and create our own areas of expertise, for example, Angela worked on the combat system and move, and Olivia worked on random generation and initializing the floor. In doing so, we could really understand our own code, making it easier to solve related problems. Thus, we could focus our efforts, reduce coupling, and not have to try to completely understand the other's code (abstraction!).

2. What would you have done differently if you had the chance to start over?

We think overall, we did a great job in delegating tasks and collaborating efficiently. If we had to change one thing, we would have tried to not focus too much on the small details during the design and implementation because we would spend a lot of time on inconsequential/bonus features when there were bigger problems to worry about. For example, we figured out how to easily count chambers with another text file, but Angela spent a long time trying to find chambers dynamically (which did not end up working). However, we were good at reminding each other to look at the bigger picture and work out the smaller kinks after we finished working on the main tasks. We also think that there were benefits to focusing on the small details because it made us think deeper about our design in order to accommodate those future changes. During the design stage, thinking about the implementation details helped us realize our flaws in understanding of the game and in our design.

Conclusion (if needed)

Overall, both of us really enjoyed this experience. At first, it was daunting to even read the assignment. Planning the UML and design patterns was the most difficult part. Approaching such a large task strategically required a holistic view of where every component would be used. The countless hours spent challenging each detail in planning allowed the execution to be a much smoother process. We are particularly satisfied with how we delegated the work. That is, the order of the classes and functions we implemented. Being able to print and see the floor right away was essential.