1 Overall Structure

1.1 Player

The `Player` is an abstract base class that is the root of all the player races. This object has all the essential methods for the players. Except for all the accessors and mutators, we have

- `virtual void move()` is responsible for player movement and notify the Observer to update its position.
- `virtual void attack()` attack the enemy if there is enemy within one radius of the player
- `virtual void usePotion()` use the potion if there is a potion within one radius of the player.

We set these methods to be virtual because it is convenient for us to implement the special abilities for each player races. However, these methods are not pure virtual methods, since most of the action of the races should be the same and no need to override these methods for every races.

For example, the special ability for `Vampire` is restore 5 health every successful attack, to implement this ability, we only need to override the `attack()` method from the `Player` base class, the `move()` method in the `Vampire` will add a condition in the method which will check if the attack is successful, and then call the mutator to add 5 HP.

If we need to add new races with new special abilities, it is suffix to add more virtual method and override them in a certain race that some method implementation is different from others.

In addition, we have implemented a new method for player called `trade`, use the command `t + direction`, can trade with a merchant if there is a merchant within one radius of the player. We will discuss the merchant in the `Enemy` class.

1.1.1 PlayerFactories

The `PlayerFactory` is a base class and it is responsible for generating certain races. Each Player Races has its own factory, for example: `Shadefactory` is responsible for generating shade, and so on. This follows the Factory Design Pattern.

1.1.2 Q&A

- **Question**: How could your design your system so that each race could be easily generated?

  Answer: For generating each race, we have a `Playerfactory` class. When we need to generate a certain race, we need to initialize a factory for the certain race, then call the `createPlayer()` method in the factory and it will return the exact race which we want.

- **Question**: Additionally, how difficult does such a solution make adding additional races?

  Answer: To add an additional race, the only thing we need to implement is its class inherit from the `Player` class and implement the special ability, the method is discussed above. Then we need to create a factory for this race which is responsible for generating this new race.

1.2 Enemy

The `Enemy` is an abstract base class that is the root of all the enemy races. It has three virtual methods that each type of enemy can override in order to implement special abilities(we can have more virtual methods for more special abilities).

- `virtual void move(PRNG& prng)` is only responsible for enemy movement. It takes in a random generator and randomly decide which direction will the Enemy move.
- `virtual bool attack(Player* player, PRNG& prng1)` is only responsible for attacking the player. It take in a raw pointer to the player which will call the HP accessors to the player. It take in a random generator only because Enemy will have the chance to miss the attack. This method will return a bool indicate that if there is a Player around the Enemy.
- `virtual bool beattacked(Player* player, PRNG& prng1)` is responsible for recieve the attack form the player.

For example, the special ability of `Halfing` is player has 50 percent chance to miss the attack. So when implementing its special ability, we only need to override the `beattacked(Player *palyer, PRNG& prng1)` method, and add a decision functionality (randomly generated) in this method to determine whether the player's attack is missed or not.

1.2.1 EnemyFactory

The EnemyFactory is responsible for generating enemy races. It is similar to PlayerFactory.

1.2.2 Merchant

The Merchants are neutral to the player if the player does not attack them. The player can trade gold with potion and other special goods from the merchant.

1.2.3 Dragon

Dragon can attack player even if the player is near the dragon hoard. We decide to bundle the dragon and dragon hoard together. We generate the dragon hoard first, then generate dragon beside the dragon hoard. Specifically see Treasure below.

1.2.4 Q&A

- **Question** How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

  Answer: Generating different enemies is similar to generating player races, each races has it's corresponding factory. The only difference from generating player character is the generating process for different enemies is completely random, meanwhile generating player is determined by the client.

- **Question** How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races?

Answer: We use the same techniques as for the player character races. In the base class, we provided the virtual method which as `attack(), move()`, etc. In the specific enemy race, we just need to override the method that need to specific change in order to implement the special ability. The example is provided above.

1.3 Items

The `class Items` is a pure abstract base on class, it is represent all the stationary items in the game, such as `Treasure` and `Potion`. It is polymorphism since `Treasure` and `Potion` are different type and they have different functionality, they are inherited from the `Item` class.

1.3.1 Potion

Since the restore and wound health is permanently added to the player, then we don't need to defect this potion. For buff/debuff atk/def, we add a private field in player, once the player reach the new floor, their buff/debuff will be set to 0. The method `usePotion` in each potion subclasses will call the modifier in the player to gain the special effects.

1.3.2 Treasure

The class `Treasure` is a base class for all kinds of treasures in the game. The specific types of treasure is inherited from this base class, the difference is the number of gold in each treasure.

The Dragonhoad is special, when the dragonhoard is generated, a dragon is spawned within 1 radius from the dragon hoard. The dragon is attached to the dragonhoard throw a pointer.

player cannot stole the hoard if the dragon is still alive. So we have implemented some method to check if the dragon is dead. If the dragon is not dead, the player can stand on the gold and get attacked from the dragon, but the player can not pick up the hoard. When the dragon is dead, the player can pick up the gold when stepping on it.

1.3.3 Q&A

1. **Question**: 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?

   Answer We could use Decorator design pattern to model the effect of the temporary potions. The subject is the player and every temporary potion can be a component. We can implement it as a linked list that recursively add the effect for each (Wound/Bosst ATK/DEF). In the abstract Component class, we can add two method called `getExtraAtk()` and `getExtraDef()`, and we will override these method in the specific potion. For example, in the Boost Atk potion, the `getExtraATK()` returns (`5 + next->getExtraATK()`) and the `getExtraDEF()` return (`0 + next->getExtraDEF()`);. So this can sum up all the extra effects in total. When the player goes to the next floor, these potions are automatically detached the linked list, free the enemy and set the pointer to nullptr.

   Unfortunately, we didn't use Decorator design pattern to implement special effect, we directly add two field called `exATK()` and `exDEF()`, the `usePotion()` method will directly mutate these fields, and destructed once the the potion is used. When the player goes to the new floor, these fields will be set to 0.

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

   Answer: Similar to generate players and enemies, we have an Itemfactory and an PotionFactory. When the need to generate specific type of item, just call the corresponding factory and it will generate the right type for you.

1.4 Floors

The `class Floor` is responsible for all the information in every floor:

- Player, item, enemies' generation
- Player, item, enemies' position
- Player, item, enemies' radius information (using to check what the player and enemies can "see")
- item, enemies destroyer

In the Floor class we will use `std::variant<Char, Creature, Item, Stair>` to create a variant called Cell, and in Floor we will have a vector Cell that stores all the objects that appear on this floor. And the getState function will return the Cell at that position. Therefore when players try to move to a direction or attack a direction, we can just use this getState(int, int) function.

If is a Char then we can check if is `'.', '#', '+'`. Then that means the player can move in that direction. If it is an Item then it means the player can walk on to that item and use it. If it is an Enemy then that means there will be combat. If it is a Stair that means the player can move to the next floor.

1.4.1 cell

cell class is the main logic that

1.5 I/O

In order to make our program can switch between ¡ncurses.h¿ and ¡iostream¿ we chose to strip out the input and out to create new classes to unify the management, when useDLC this bool is true our getInput() in the game engine will automatically read commands from ¡ncurses.h¿ and output them, and vice versa from ¡iostream¿. Instead, it reads commands and output from ¡iostream¿.

The `class GameInput` handles all the input from the player.
The `class GameOutput` is responsible for all the outputs.

1.5.1 `Map`

The `class GameMap` is a constent class that stores all the information for the output. The `std::vector<std::ve` stores the information of the chamber and positions. The row of the vector stores the index of the chamber, and each element of the row is the valid position for each chamber. Every time when player enter in to a new floor, floor will clear the old data, and read the new blank data that stores in GameMap, and then generate the objects in the floor.

1.6 The Game Engine

The `class GameEngine` integrate all the modules together, create the control flow of the game.

- `void handlePlayerCmd(PlayerCmd cmd)` take in a command from the client and give the Player Character instructions to do all the action.
- `void handleEnemiesAction();` automatically give instructions to Enemies.

1.7 Main.cc

In the `main()` function, we have set up command line argument for the program: `./cc3k -DLC` is for downloadable content, `./cc3k -m filename` is for inputting pre-generated map.

2 Design

2.1 Polymorphism

Polymorphism allows the game to handle different types of characters, enemies and items uniformly through their base classes.

The base class Item provides a common interface for all game items. Concrete classes like Potion and Treasure implement specific item behaviors. This design lets the game interact with different items in the same way, using polymorphism. For example, a single function can call the use method on any Item object, allowing for dynamic and varied item interactions during gameplay. This approach simplifies the code and makes it more extensible and easier to maintain.

The factory pattern, combined with polymorphism enhances the game's flexibility by allowing dynamic creation of various characters and items at runtime. Abstract factory classes like CharacterFactory and ItemFactory define methods for creating objects, while concrete implementations such as ConcretePlayerFactory and ConcreteEnemyFactory instantiate specific types of characters and items. This approach encapsulates the creation logic and decouples the client code from the instantiation details. As a result, adding new character types or items to the game requires minimal changes to the existing codebase, as the factories can simply be extended to include new types.

2.2 Design patterns

2.2.1 Observer Design patterns

We use observer design patterns for displaying the gameplay. The Observer is the `GameOutput` class. The Subject is the `GameEngine`. The `GameEngine` will notify the `GameOutput` every gameloop so it can get the updated state from the game and display on the screen. This Design pattern allow us to have multiple display system. Although in this project we didn't add a way to display on the window, the design will give us the chance to easily add this new feature and displaying the gameplay at the same time.

2.2.2 Factory Design patterns

We mostly use Factory design patterns to generate different types of Player Characters, Enemies and Items in every floor. For example, the CharacterFactory class creates different types of characters

(e.g. Shadow, Raven, Vampire, Troll, Goblin)s. This encapsulates the creation logic and makes it easier to add new character types in the future. The benefit of using Factory Design patterns is adding new character types requires minimal changes in the existing code. Just add a new class for the new character and update the factory. It also reduce code duplication and enhance readability.

2.3 Coupling and Cohesion

- **Couping**: We uses factory patterns to create characters and items. This means that the client doesn't need to know the details of how these objects are created, just that they are created. This reduces the direct dependency between classes.

  We also uses abstract base classes like Player and Item. We have also separated Abstract Input and Output Classes. This allows different types of player characters and items to be used without the game needing to know their specific types. It only interacts with them through common methods defined in the base classes. such as `usePotion()` etc.

  Because of these patterns, the classes in the game are loosely connected. Changes in one class have minimal impact on others, making the system more flexible and easier to debug.

- **Cohesion**: Each class in the game has a specific job and their own responisiblities. For example, `GameEngine` manages the game control flow, `GameInput` handles user inputs, and the player class defines the behavior of that character, such as `trad(), move(), attack()` methods. The `Floor` class is responsible for random generate positions for all entities and so on.

  By organizing behaviors into specific classes and defining common behaviors in abstract classes, the game design keeps things tidy and understandable. For example, all Treasures and Potions are inherit from *Item*, and all Player Character Races are inherit from Player, with each subclass implementing its specific details

3 Resilience to Change

The biggest difference from the first plan to the final solution is we use more abstractions to decrease coupling and increasing cohesion. We planned to use decorator design pattern to implement the Display of out game. But we find out this design to really hard to achieve, because we need to make every Entities to be a component from the map. This contradict the low coupling and high cohesion principle, since the map, item, player, enemies will have completely different responsibilities and functionality, it is also difficult for us to abstract the classes. So we decided to redo the project five days before the deadline. We create all the classes set up strict responsibility and functionality for each classes. In addition, we strictly follow the object oriented design principle, and we have found it is beneficial for us.

4 Extra Credit Features

- We have done the trade system between Player and Merchant. The Merchant will sell the floowing: Telepot(instantly tranport to the next floor), Gold, Potion. The type of potion will be shown on the screen.

- to trade with Mercant t + Dir.

- Using the library `ncurses`, this allow us to read in the command instantly without entering `enter` for every command we input.
- to open DLC run " ./cc3k -DLC ".
- to load the ginven map run " ./cc3k -m "ginvenMapFileName" ".

5 Final Questions

1. **Question**: 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?

   Answer: Planning, coordination, and formatting are very important in team development, and in these two weeks of teamwork we encountered a lot of problems that were not code-related, such as inconsistent code formatting, inconsistent naming rules, and inconsistencies in the way each person implements a particular feature. We all deeply realized how important the code format taught in cs135 is. In these two weeks, the time we spent on writing code was far less than the time we spent on solving the problem of explaining to each other how to implement our own code, and due to the lack of clarity in the comments in our code, we were often unable to read and understand each other's code.

   In addition to this, another important issue we encountered in our teamwork was resolving GitHub , merge conflicts. We use GitHub to version our code, but because of the file naming inconsistency in our team, there was a situation where the modified code could not be detected by Git and could not be merged. Through this project, we learned that any collaborative project must have its workflow and work requirements before starting, which is not like homework or personal projects where you can write code at will. Any lazy idea can lead to hours of fixing at the end of the day.

   Regarding the assignment, We think we did a better job because we had a serious logic and class relationship problem in the middle of writing the code, so we chose to start all over again, and this time we did a good job with the tasks they were assigned to due to the experience we had the first time around.

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

   The reason why we have serious problems with the code for the first time, first of all, is the lack of experience, in the beginning each one of us do not have any idea in the head about how to realize a whole big framework, due to the previous cs assignment is the school as well as write a good framework for us, we just need to think about how to realize it. So this was the first time we had to build a whole framework by ourselves. After the first round of discussion, we found a lot of probable feasible solutions, at this stage we have a rough idea when we don't know whether we can realize it or not, for example, we have considered to output the whole game by layers through decorator at the beginning, and this idea comes from our a4q2. In the end, we found that although it's feasible, but we have to write all the objects in the game in a single layer, so we need to write all the objects in the game in the same way. Eventually we realized that although it was feasible, it would be too much work to write all the objects in the game as decorators to decorate the Blank Map.

So we changed our mind and chose to have a vector of chars to act as a buffer to output the game page. This idea is right, but due to the first time in our project resource management is too confusing, for example, obviously is a buffer used to store output, in our code as far as possible appear from the buffer to read the number to determine the situation. Because the buffer is updated uniformly, this leads to a situation where the enemies may overlap with each other, and the data he reads from the buffer is not the latest, but the last. So I think we need a separate place to store the data in the cells first, when any unit changes the cells will be modified inside, and when the turn is over the cells will be fed into the buffer, and the buffer will be output. which brings us to the current version! If we re-do this project then we will choose, clear our goals and needs, clear each class is responsible for and to achieve the function, clear the relationship between each class, try to avoid hard code, to ensure that our code is live when we want to add new code without having to modify the previous code, should not be done by the class is not allowed to do it. Every class, every function, should have a clear function and goal. We need to subdivide each function we want to realize at the beginning, and then put the ones with similar functions into a class for unified management. And then think about the relationship between each class, try to avoid appearing I need function but in my field but access can not. And then make sure that the above ideas are clear, is to carry out a clear code format requirements, comment requirements, git merge requirements. When all these steps are done, we think we can do any team project perfectly!