ChamberCrawler3000 - Documentation and Design

Introduction:

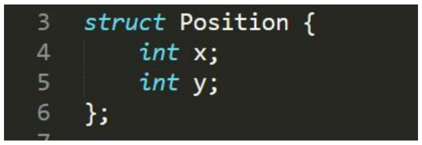
ChamberCrawler3000, abbreviated as CC3k, which is a simplified rogue-like that consists of 30 rows and 79 columns. In our project, we have created our own version of the CC3k game using the techniques and design patterns learned in cs246.

Overview

In our iteration of ChamberCrawler3000, we incorporated numerous design patterns, including the Model-View-Controller pattern, the Factory method pattern, the Visitor pattern and the Template method pattern. We also tasked ourselves to code the entire project without explicitly managing our own memory using vectors and smart pointers. There were three primary structs that were used to hold information and four primary classes that acted as the solid foundation in the structure of our project. Position, Name and Stats were three structures that enumerated data fields all over the project. The most important classes of our project were the classes: Character, Level, Game, and Display.

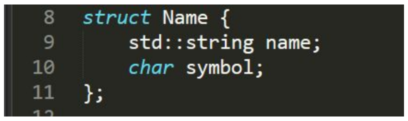
(Note: Pictures are not the entire code. They are simply for reference.)

Position:
A simple, but also the used structure in our program, consisting of two integer values, x and y. A pair used to specify a single point on the two dimensional map.



```
struct Position {
    int x;
    int y;
};
```

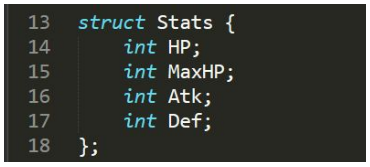
Name:
Every visible entity in the game has a symbol to be displayed on the map, and an actual name that helps with outputting information. Thus, this is a structure with a char named "symbol" and a string for its "name".



```
struct Name {
    std::string name;
    char symbol;
};
```

Stats:

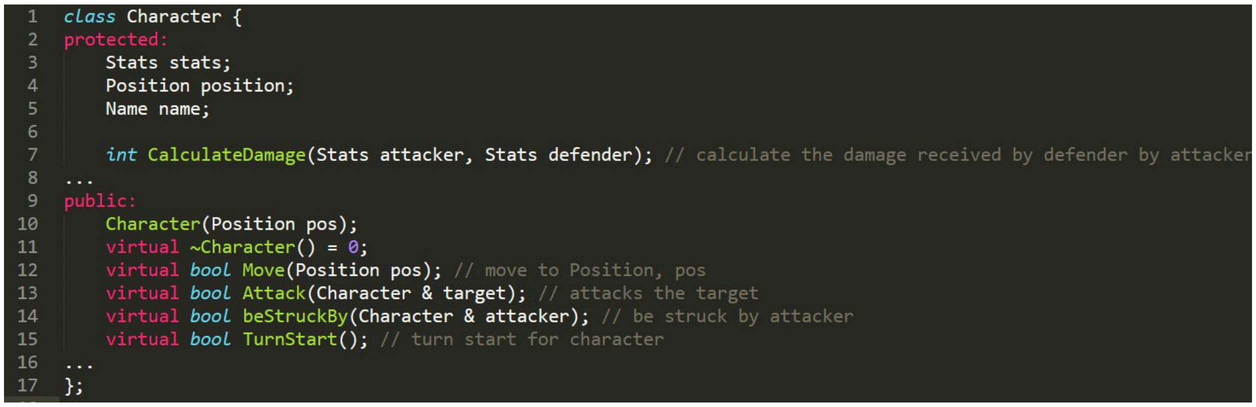
All characters (and potions) have stats HP, MaxHP, Atk, and Def. Instead of writing each variable individually, the stats structure can encapsulate all of these fields



```
struct Stats {
    int HP;
    int MaxHP;
    int Atk;
    int Def;
};
```

Character:

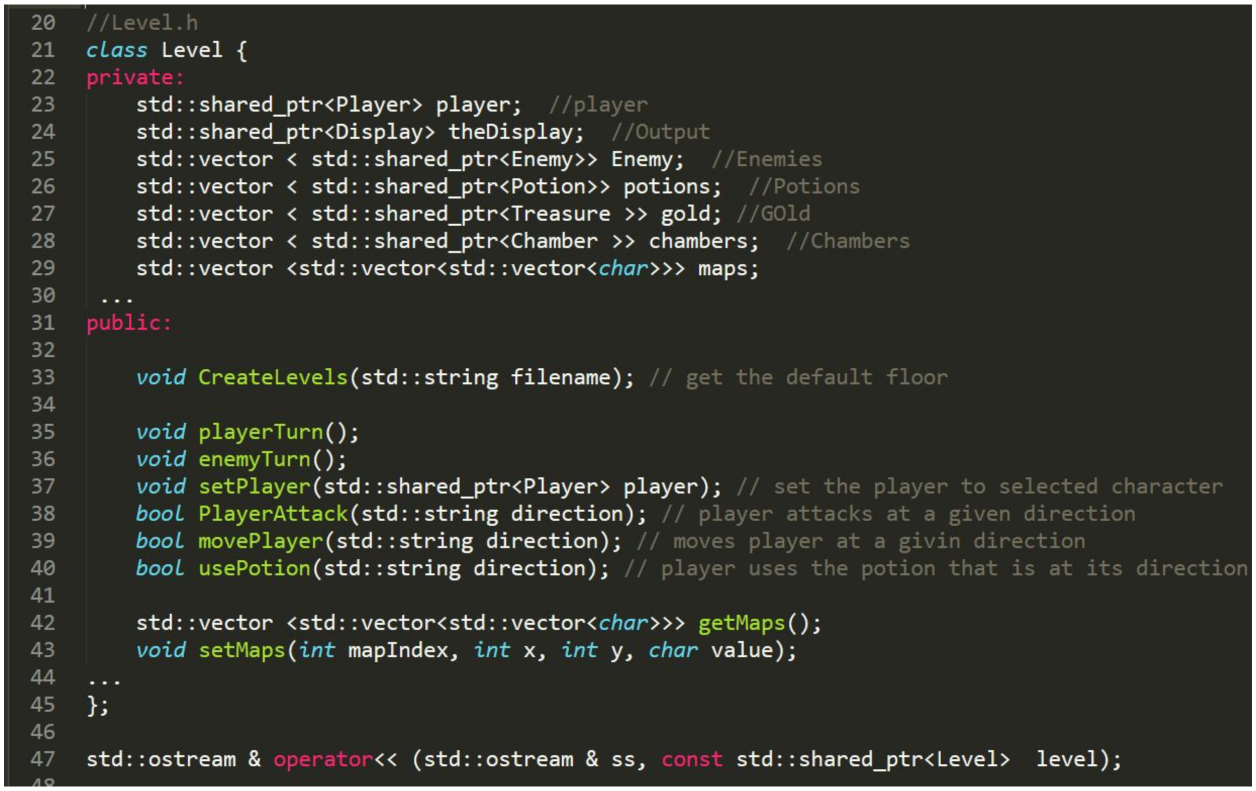
Every "character", that is- any entity that is capable of moving and attacking inherits the character class. In our iteration of CC3k, there were two subcomponents that fit this description: Players and Enemies. Thus, both players and enemies follow the Template method pattern and are abstract classes that also inherit Character. As mentioned before, the functions that all Characters have in common are Move() and Attack(). In addition, adding a beStruckBy(Character & attacker) function will also allow for different combinations of attacker and defender via the Visitor pattern. However, the movement and attacking of most enemies is mostly similar, therefore, instead making them all pure virtual functions we instead made them simply virtual functions with default behaviour. This allows for us to only have to override functions that have a specific ability in mind, for example, the vampire's life stealing ability. There are also other common fields amongst all characters, such as a Position, Name, and Stats.



```
class Character {
protected:
    Stats stats;
    Position position;
    Name name;

    int CalculateDamage(Stats attacker, Stats defender); // calculate the damage received by defender by attacker
...
public:
    Character(Position pos);
    virtual ~Character() = 0;
    virtual bool Move(Position pos); // move to Position, pos
    virtual bool Attack(Character & target); // attacks the target
    virtual bool beStruckBy(Character & attacker); // be struck by attacker
    virtual bool TurnStart(); // turn start for character
...
};
```

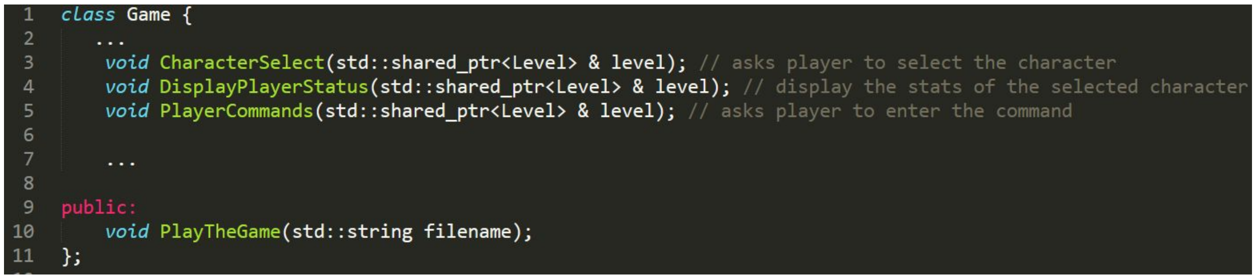
Level:

In our program the Level class is the "Model" in the Model-View-Controller design pattern. It is the primary structure that has accessors to all potential actions in the game (but it doesn't input/output anything itself). It connects the controller (Game) and the view (Display) together to control the state of the game, and has many accessor functions for various fields in the program. For example, in our Level class, there is a function createLevel(), which reads or randomly creates all five maps at the start of the program, setPlayer(Player * player) to set the player after selecting its race, a function movePlayer(Position pos), and a function enemyTurn(), which does all of the actions for all enemies in the level. It also has various important fields such as maps, which holds all five maps in the game, theDisplay, which holds an instance of the Display, and pointers to the Player, Enemies, and Items.



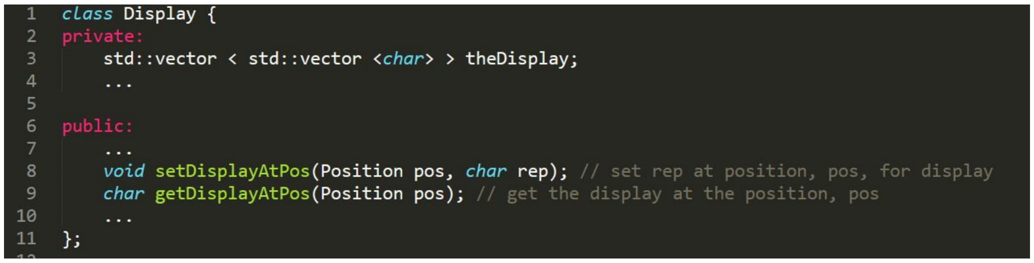
```cpp
//Level.h
class Level {
private:
    std::shared_ptr<Player> player;  //player
    std::shared_ptr<Display> theDisplay;  //Output
    std::vector < std::shared_ptr<Enemy>> Enemy;  //Enemies
    std::vector < std::shared_ptr<Potion>> potions;  //Potions
    std::vector < std::shared_ptr<Treasure >> gold; //GOld
    std::vector < std::shared_ptr<Chamber >> chambers;  //Chambers
    std::vector <std::vector<std::vector<char>>> maps;
 ...
public:

    void CreateLevels(std::string filename); // get the default floor

    void playerTurn();
    void enemyTurn();
    void setPlayer(std::shared_ptr<Player> player); // set the player to selected character
    bool PlayerAttack(std::string direction); // player attacks at a given direction
    bool movePlayer(std::string direction); // moves player at a givin direction
    bool usePotion(std::string direction); // player uses the potion that is at its direction

    std::vector <std::vector<std::vector<char>>> getMaps();
    void setMaps(int mapIndex, int x, int y, char value);
...
};

std::ostream & operator<< (std::ostream & ss, const std::shared_ptr<Level>  level);
```

Game:
If Level is the Model, then Game is the Controller in the Model-View-Controller pattern. It manages player input and calls functions from the Level class in order to proceed in the program. It has two primary functions: CharacterSelect(), which is where the player inputs their race, and PlayerCommands, which sends Player input when they are inside of the dungeon.



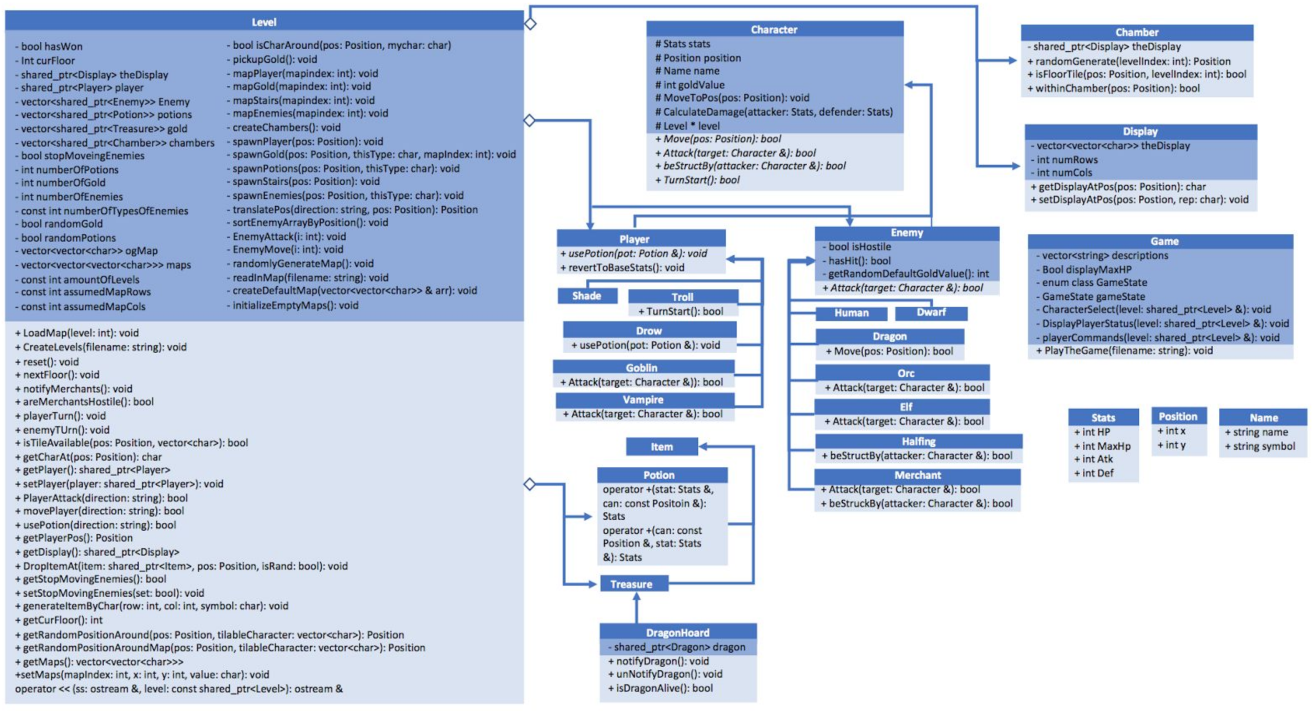
```cpp
class Game {
   ...
    void CharacterSelect(std::shared_ptr<Level> & level); // asks player to select the character
    void DisplayPlayerStatus(std::shared_ptr<Level> & level); // display the stats of the selected character
    void PlayerCommands(std::shared_ptr<Level> & level); // asks player to enter the command

    ...

public:
    void PlayTheGame(std::string filename);
};
```

Display:
The Display class is the View in the Model-View-Controller pattern. It controls the primary output of the game - i.e. the map. It is a very simple class consisting of a two dimensional char array and appropriate accessor methods.



```cpp
class Display {
private:
    std::vector < std::vector <char> > theDisplay;
    ...

public:
    ...
    void setDisplayAtPos(Position pos, char rep); // set rep at position, pos, for display
    char getDisplayAtPos(Position pos); // get the display at the position, pos
    ...
};
```

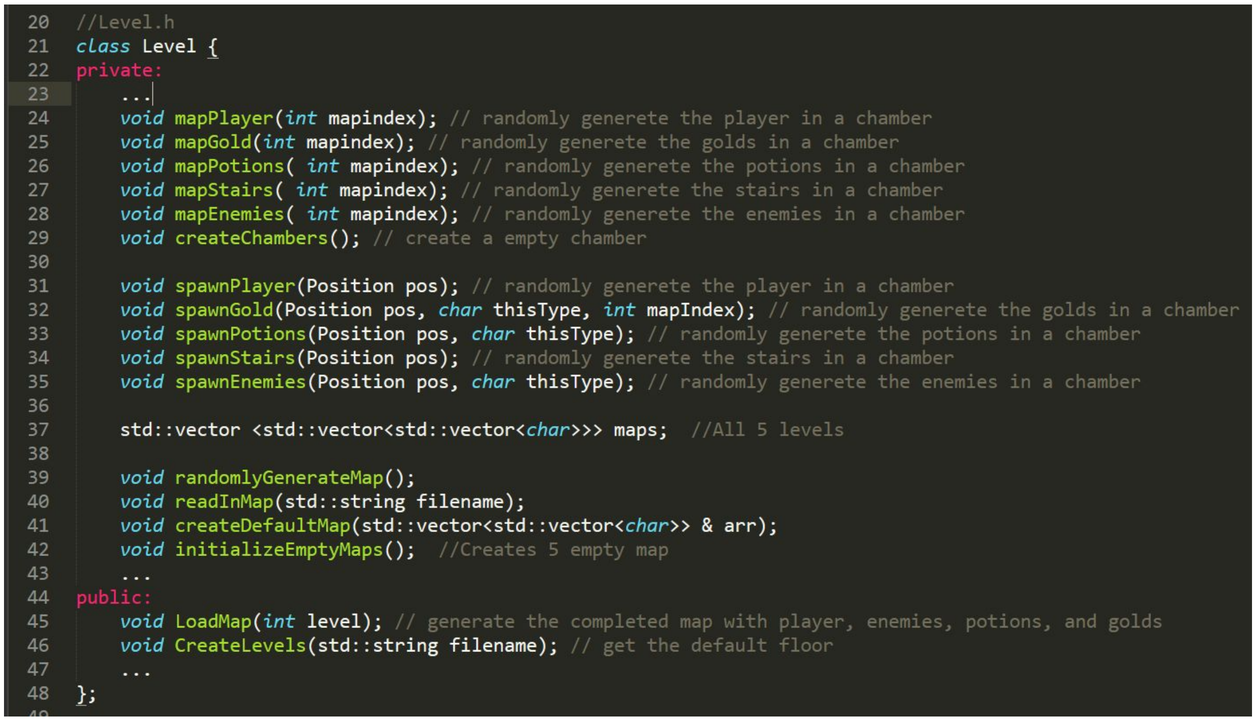
Updated UML


Level
- bool hasWon
- Int curFloor
- shared_ptr<Display> theDisplay
- shared_ptr<Player> player
- vector<shared_ptr<Enemy>> Enemy
- vector<shared_ptr<Potion>> potions
- vector<shared_ptr<Treasure>> gold
- vector<shared_ptr<Chamber>> chambers
- bool stopMoveingEnemies
- int numberOfPotions
- int numberOfGold
- int numberOfEnemies
- const int numberOfTypesOfEnemies
- bool randomGold
- bool randomPotions
- vector<vector<char>> ogMap
- vector<vector<vector<char>>> maps
- const int amountOfLevels
- const int assumedMapRows
- const int assumedMapCols
- bool isCharAround(pos: Position, mychar: char)
- pickupGold(): void
- mapPlayer(mapindex: int): void
- mapGold(mapindex: int): void
- mapStairs(mapindex: int): void
- mapEnemies(mapindex: int): void
- createChambers(): void
- spawnPlayer(pos: Position): void
- spawnGold(pos: Position, thisType: char, mapIndex: int): void
- spawnPotions(pos: Position, thisType: char): void
- spawnStairs(pos: Position): void
- spawnEnemies(pos: Position, thisType: char): void
- translatePos(direction: string, pos: Position): Position
- sortEnemyArrayByPosition(): void
- EnemyAttack(i: int): void
- EnemyMove(i: int): void
- randomlyGenerateMap(): void
- readInMap(filename: string): void
- createDefaultMap(vector<vector<char>> & arr): void
- initializeEmptyMaps(): void
+ LoadMap(level: int): void
+ CreateLevels(filename: string): void
+ reset(): void
+ nextFloor(): void
+ notifyMerchants(): void
+ areMerchantsHostile(): bool
+ playerTurn(): void
+ enemyTUrn(): void
+ isTileAvailable(pos: Position, vector<char>): bool
+ getCharAt(pos: Position): char
+ getPlayer(): shared_ptr<Player>
+ setPlayer(player: shared_ptr<Player>): void
+ PlayerAttack(direction: string): bool
+ movePlayer(direction: string): bool
+ usePotion(direction: string): bool
+ getPlayerPos(): Position
+ getDisplay(): shared_ptr<Display>
+ DropItemAt(item: shared_ptr<Item>, pos: Position, isRand: bool): void
+ getStopMovingEnemies(): bool
+ setStopMovingEnemies(set: bool): void
+ generateItemByChar(row: int, col: int, symbol: char): void
+ getCurFloor(): int
+ getRandomPositionAround(pos: Position, tilableCharacter: vector<char>): Position
+ getRandomPositionAroundMap(pos: Position, tilableCharacter: vector<char>): Position
+ getMaps(): vector<vector<char>>>
+setMaps(mapIndex: int, x: int, y: int, value: char): void
operator << (ss: ostream &, level: const shared_ptr<Level>): ostream &
Character
# Stats stats
# Position position
# Name name
# int goldValue
# MoveToPos(pos: Position): void
# CalculateDamage(attacker: Stats, defender: Stats)
# Level * level
+ Move(pos: Position): bool
+ Attack(target: Character &): bool
+ beStructBy(attacker: Character &): bool
+ TurnStart(): bool
Player
+ usePotion(pot: Potion &): void
+ revertToBaseStats(): void
Shade
Troll
+ TurnStart(): bool
Drow
+ usePotion(pot: Potion &): void
Goblin
+ Attack(target: Character &)): bool
Vampire
+ Attack(target: Character &): bool
Enemy
- bool isHostile
- hasHit(): bool
- getRandomDefaultGoldValue(): int
+ Attack(target: Character &): bool
Human
Dwarf
Dragon
+ Move(pos: Position): bool
Orc
+ Attack(target: Character &): bool
Elf
+ Attack(target: Character &): bool
Halfing
+ beStructBy(attacker: Character &): bool
Merchant
+ Attack(target: Character &): bool
+ beStruckBy(attacker: Character &): bool
Item
Potion
operator +(stat: Stats &, can: const Positoin &): Stats
operator +(can: const Position &, stat: Stats &): Stats
Treasure
DragonHoard
- shared_ptr<Dragon> dragon
+ notifyDragon(): void
+ unNotifyDragon(): void
+ isDragonAlive(): bool
Chamber
- shared_ptr<Display> theDisplay
+ randomGenerate(levelIndex: int): Position
+ isFloorTile(pos: Position, levelIndex: int): bool
+ withinChamber(pos: Position): bool
Display
- vector<vector<char>> theDisplay
- int numRows
- int numCols
+ getDisplayAtPos(pos: Position): char
+ setDisplayAtPos(pos: Postion, rep: char): void
Game
- vector<string> descriptions
- Bool displayMaxHP
- enum class GameState
- GameState gameState
- CharacterSelect(level: shared_ptr<Level> &): void
- DisplayPlayerStatus(level: shared_ptr<Level> &): void
- playerCommands(level: shared_ptr<Level> &): void
+ PlayTheGame(filename: string): void
Stats
+ int HP
+ int MaxHp
+ int Atk
+ int Def
Position
+ int x
+ int y
Name
+ string name
+ string symbol

Design:

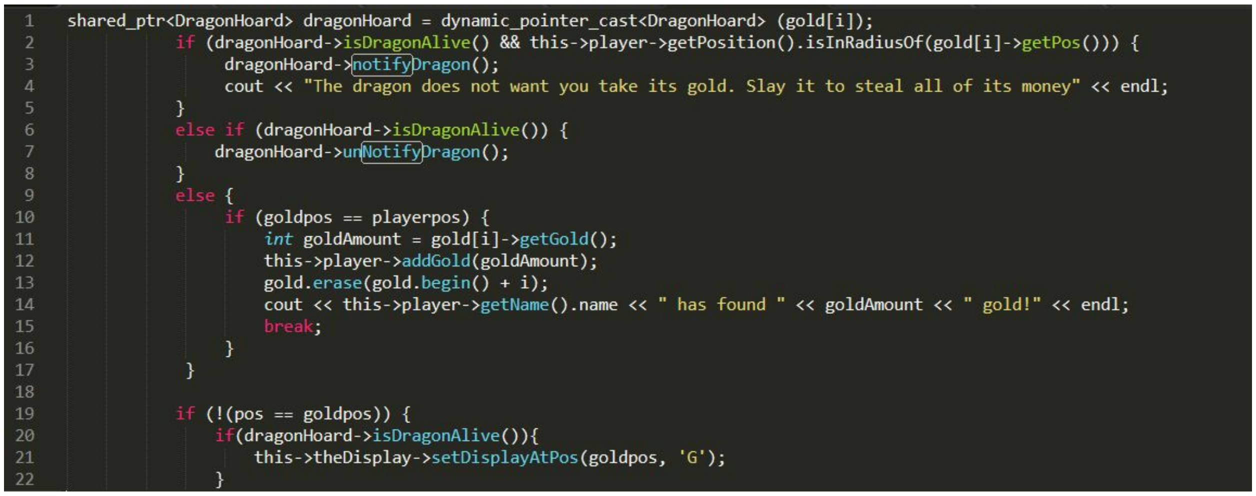
There were two primary difficulties with the implementation of cc3k which required some thought to fix. The two most difficult tasks were: designing the functions that read maps in a way such that it reuses as much code as possible, and designing a way to connect the Dragon with a Dragon hoard.

One of the difficulties was designing the program in such a way so that it would can both randomly generate maps, and read in maps, while reusing much of the same code in order to avoid repetition of code, and consistency (i.e. random generation works if and only if reading input files for the map works) Our initial interface design failed to include this capability, so we had to rethink about our plan, and how to create the different types of maps. Initially, how we planned on tackling this problem is that we read in maps, one by one, as we needed them. In other words, at the start of the game, one level is create, and only when the player advances to the next floor does the next map get generated. The issue with this approach is that there are two different ways to read in maps, which essentially do the same thing, but are implemented in different ways. We fixed this by allocating some memory to hold all of the 5 maps at the start of the program (i.e. vector<vector<vector<char>>>) and wrote functions to convert this 3D array into actually spawning the objects. This way, if we wanted to randomly generate the map, we would create 5 randomly generated maps at the start, which is equivalent to the five maps given in the input file. Therefore, both randomly generated maps and predetermined maps share the exact same code and is more consistent.



```
//Level.h
class Level {
private:
    ...
    void mapPlayer(int mapindex); // randomly generete the player in a chamber
    void mapGold(int mapindex); // randomly generete the golds in a chamber
    void mapPotions( int mapindex); // randomly generete the potions in a chamber
    void mapStairs( int mapindex); // randomly generete the stairs in a chamber
    void mapEnemies( int mapindex); // randomly generete the enemies in a chamber
    void createChambers(); // create a empty chamber

    void spawnPlayer(Position pos); // randomly generete the player in a chamber
    void spawnGold(Position pos, char thisType, int mapIndex); // randomly generete the golds in a chamber
    void spawnPotions(Position pos, char thisType); // randomly generete the potions in a chamber
    void spawnStairs(Position pos); // randomly generete the stairs in a chamber
    void spawnEnemies(Position pos, char thisType); // randomly generete the enemies in a chamber

    std::vector <std::vector<std::vector<char>>> maps;  //All 5 levels

    void randomlyGenerateMap();
    void readInMap(std::string filename);
    void createDefaultMap(std::vector<std::vector<char>> & arr);
    void initializeEmptyMaps();  //Creates 5 empty map
    ...
public:
    void LoadMap(int level); // generate the completed map with player, enemies, potions, and golds
    void CreateLevels(std::string filename); // get the default floor
    ...
};
```

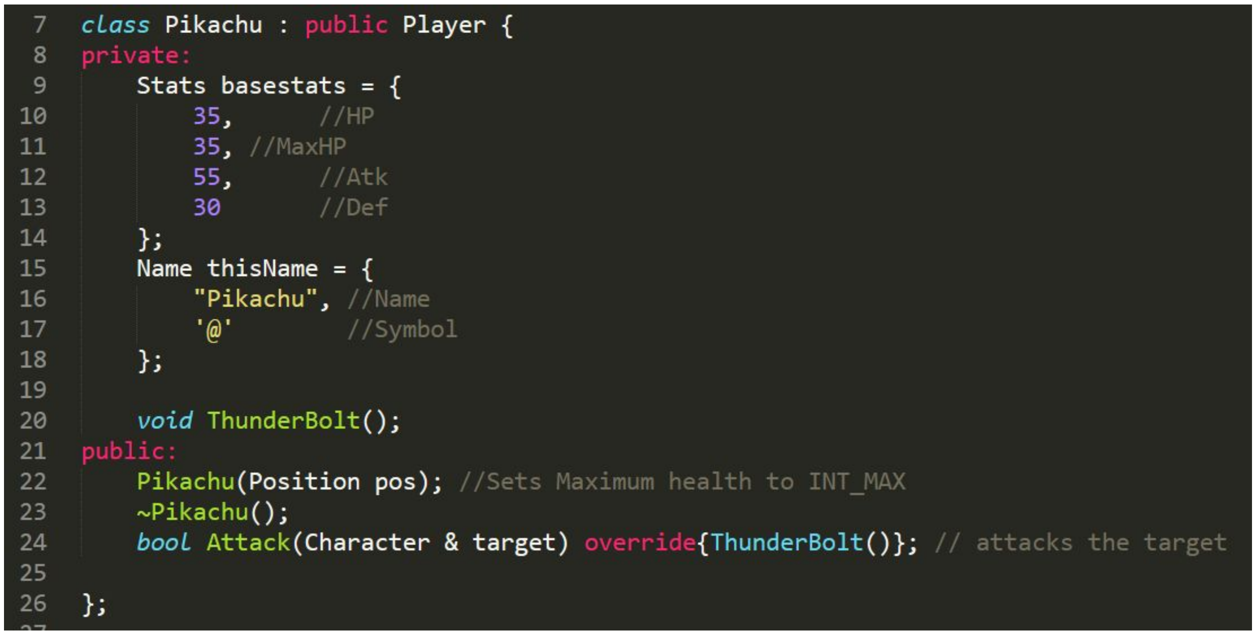
The second difficulty was trying to figure out how to have a Dragon connected to the DragonHoard. This is difficult because Dragon is an Enemy, but a DragonHoard is an Item. To fix this, we created a pointer to a dragon in the DragonHoard, so it can notify/un-notify the dragon as the player approaches the dragon's gold. In a way (not exactly), it is synonymous with the Observer pattern, as the DragonHoard acts as the Observer for the dragon, the subject.



```
shared_ptr<DragonHoard> dragonHoard = dynamic_pointer_cast<DragonHoard> (gold[i]);
            if (dragonHoard->isDragonAlive() && this->player->getPosition().isInRadiusOf(gold[i]->getPos())) {
                 dragonHoard->notifyDragon();
                 cout << "The dragon does not want you take its gold. Slay it to steal all of its money" << endl;
            }
            else if (dragonHoard->isDragonAlive()) {
                dragonHoard->unNotifyDragon();
            }
            else {
                 if (goldpos == playerpos) {
                     int goldAmount = gold[i]->getGold();
                     this->player->addGold(goldAmount);
                     gold.erase(gold.begin() + i);
                     cout << this->player->getName().name << " has found " << goldAmount << " gold!" << endl;
                     break;
                 }
             }

            if (!(pos == goldpos)) {
                if(dragonHoard->isDragonAlive()){
                    this->theDisplay->setDisplayAtPos(goldpos, 'G');
                }
```

Resilience To Change:

We believe that it is a very simple task to accommodate for changes in our implementation of cc3k. There are various additions that one may want to include, such as adding new characters, creating new items, and changing the input/output of the program. For example, if you wanted to add a new Player or Enemy, you simply have to create a class that inherits the Player or Enemy class accordingly and override any methods if you want a unique implementation. For example, if I wanted to make a Player race that attacks using a special move called Thunderbolt, which does 90 damage at 100% accuracy, simply inherit the Player class, override the Attack() method to call ThunderBolt(), and include it in the CharacterSelect() function in Game.cc. Another example is if you wanted to make a new type of Potion, simply change the implementation of the AssignThisPotion() function, and you're good to go! Similarly, if you want to change the input/output of the program (i.e. add Xbox controllers/XWindow display), change the Game class and the Display class in order to fit the new specifications.



```
class Pikachu : public Player {
private:
    Stats basestats = {
        35,      //HP
        35, //MaxHP
        55,      //Atk
        30       //Def
    };
    Name thisName = {
        "Pikachu", //Name
        '@'        //Symbol
    };

    void ThunderBolt();
public:
    Pikachu(Position pos); //Sets Maximum health to INT_MAX
    ~Pikachu();
    bool Attack(Character & target) override{ThunderBolt()}; // attacks the target

};
```

Questions & Answers:

- *How could you design your system so that each race could be easily generated? Additionally? how difficult does such a solution make adding additional races?*
    - Both the Player and Enemy classes are abstract classes of the base class, Character. We know that every character can move, attack, and do something (i.e. move/special effects) at the start of the turn, so this is why the Player and Enemy classes derive from it. For Players, the concrete classes are Shade, Drow, Vampire, Troll, and Goblin, each representing the race specified in the ChamberCrawler3000 document. We have another abstract function named, "SpecialEffect()", which allows the different races to have a strong level of control over its unique special ability. For example, the special ability for Drow is that it has a 1.5x magnifier for any potion it uses, and the special ability for Troll is that it regains 5 HP every turn. Overall, by this implementation, in order to add a new race to the game, we would simply write a class that inherits the Player class, and provide implementation for the abstract classes.

- *How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?*
    - **Initial Answer:** The system for generating enemies is essentially identical to the generation of the Player. Class Enemy inherits the base class Character, and the specific types of enemies inherit the abstract class Enemy. Much like Player, this makes it easy to implement new enemies into the game: You just create a new class that inherits the Enemy class, and provide the implementation of its abstract

classes. What is different about the Enemy class is mostly its fields. Enemy has a field determining whether or not it is hostile, it has a gold amount that the Player receives for slaying it, and cannot use potions, which is why it is a separate entity from the Player class. However, in general, most of the classes, Player and Enemy are identical in terms of the interface functions.

  - **Review:** The structure of Player/Enemy transactions remained the same, as planned.

- *How could you implement the various abilities for enemy characters? Do you use the same techniques as for the player character races?*
  - **Initial Answer:** Every race is a separate class that inherits from the Enemy class (which inherits from the Character class) In the character class, there are abstract functions that every concrete enemy must implement. That is: Move(), Attack(), TurnStart(), and SpecialEffect() (The special effect class is just for organization). This allows for us to dynamically design the special abilities to fit the requirements for any race by overriding the implementation of the base classes. It is implemented similarly for Players. For example, if we wanted to implement the Goblin's special ability to steal 5 gold from every slain enemy we would be able to without too much trouble. In the implementation of the attack function, we would attack the enemy as normal, determine whether or not the enemy has been, and if it has, call the SpecialEffects() function, which increases the Player's HP by 5. We think that this is an effective way of using special abilities because it allows for control and organization.
  - **Review:** The SpecialEffect() function was much too generic of a function to be useful. It was replaced with private functions in the specific classes which have a special effect.

- *The Decorator and Strategy patterns are possible canidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work bet*ter? *Explain in detail, by weighing the advantages/disadvantages of the two patterns.*
  - **Initial Answer:** The strategy pattern allows for us to augment the implementation of and object at runtime. In contrast, in the decorator pattern, we are adding to the implementation of an object at runtime. Therefore, the more appropriate design pattern to choose for ChamberCrawler3000 would be the Strategy pattern, because you will never have two potions mixed together (though that would be great DLC) In other words, you there won't be a time (according to the specifications) where we have both a healing effect and a boost attack effect, therefore the Strategy pattern would be the more appropriate implementation.

However, for the purposes of our project, we simply created a Potion class that inherits from an Item class and added a use () function that modifies the player's Stats (HP, Atk, Def). We believe that a heavy duty implementation of the Decorator or Strategy pattern is not necessary, and actually less intuitive for the scope of this project, which is why we did this. Still, we do not believe that it is that difficult to add a new kind of potion, as you simply need to pick a name for the potion, and implement it into the use() function.

    - **Review:** Our way of implementing the types of different potions worked for the project specifications, but may not accommodate for change. We still believe it is easy to create a new type of potion, but would be difficult to create a unique potion (e.g. player moves 2 turns instead of 1)

- *How would 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?*
    - **Initial Answer:** Both gold and potions are derived from a base class called Items. The Items class has fields that contain its position and its name and symbol. Since both gold and potions inherit from the base class item, we can easily associate the two objects through an item pointer *, and possibly implement an iterator to traverse through a list of items if necessary. This association by inheritance allows for us to reuse code and associate similar classes together for any potential DLC.
    - **Review:** The structure of Treasure/Potion transactions remained the same, as planned, and seemed to be effective for the most part.

Extra Credit Features:

- Implemented the entire program without explicitly managing memory using vectors and smart pointers. (i.e. there is no "new" or "delete" keyword in the program)

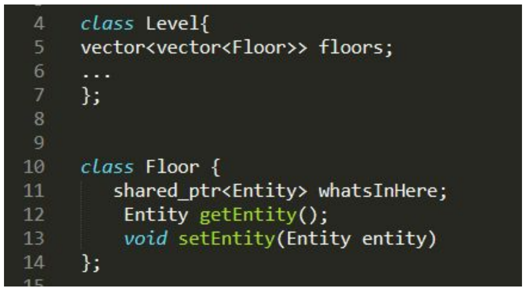
Final Questions:

- *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?*
    - There are various aspects involved in working in a team. One aspect is accounting for the team members' experience/skillset. Does the team member know how to use Git to add and commit files? We would have to weigh what would the team member prefer to do versus what has to be done. Another one is team coordination. Team coordination is often difficult while working on a single project. Take this example for consideration: A bug arises! Who fixes it? Should one person try and fix it while the others twiddle their thumbs? Or should all three have a try in fixing it, which will probably result in merge conflicts. Either way,

time is wasted. The final one is consistency/fluency of the code. Everyone has their own coding style, so when you try to merge them together, they don't always fit like pieces in a puzzle. It isn't always so straightforward reading someone else's code, and it can lead to a lower quality end product if not careful. These are a few of many more aspects of working in a team.

    ○

- *What would you have done differently if you had the chance to start over?*
    - With our implementation of cc3k, it would be a little difficult trying to find "what object is in this tile"? This is because Enemies/Items/Gold are managed through arrays of their respective type (i.e. vector<Enemy>, vector<Item>, etc) If we were to redesign the program, we would layout the "level" of the game through a 2D array of objects called floors. In these floors, we would have a "Entity". An entity is one of: Player, Enemy, Treasure, Potion, char. (i.e. Player, Enemy, Treasure, Potion inherit from the Entity class) This way, if we wanted to obtain the object inside of any 2D object, we would simply use the accessor inside of the Entity class, and perform a specific operation based on the object of the class. This would allow more freedom to more easily manipulate any tile in the game. Besides that, we believe that our code was sufficient given the scope of the project.



```
class Level{
vector<vector<Floor>> floors;
...
};


class Floor {
    shared_ptr<Entity> whatsInHere;
     Entity getEntity();
     void setEntity(Entity entity)
};
```

Conclusion:

Developing cc3k was a wonderful experience, and working in a team on a big project provided some insight on how a programming job in a team may be like. We believe that our code is relatively sufficient for the scope of the project. It incorporates many design patterns such as the Model-View-Controller pattern, the Factory method pattern, the Visitor pattern and the Template method pattern. It also written in such a way that potential changes can be accounted for if required. It's not perfect- there is still some room for improvement, but we believe it is sufficient for the scope of the project.