CS246 FINAL PROJECT

STRAIGHT

Student: Nguyen Vu

ID: 20831324

Fall 2020

Overview

Straight is a game written completely in C++. Debugged with gdb and Valgrind. It can be compiled and run in a terminal.

It is a card game played with a deck of 52 cards, played with 4 players. This project has a text interface, so users needs to type commands to play. This game also allow to have computer player when there is not enough 4 people to play.

There are a few std libraries used to implement classes.

- <vector>: most use in CardList class and StraightGame class
- <string>: used across the classes in the program
- <algorithm>: used in shuffle() method in Deck class
- <random>: used in shuffle() method in Deck class
- <chrono>: used to shuffle the deck based on time in shuffle() method in Deck class.
- <memory>: used unique pointers in Player class and StraightGame class
- <iostream>: used to input and output messages in the interface, also output error messages.

The project contains those files:

- card.h and card.cc: implement Card class.
- cardList.h and cardList.cc: implement CardList class
- player.h and player.cc: implement Player, Human, and Computer class.
- deck.h and deck.cc: implement Deck class
- straightGame.h and straightGame.cc: implement StraightGame class
- main.cc: implement the game interface and in/output.
- makefile: to compile the program

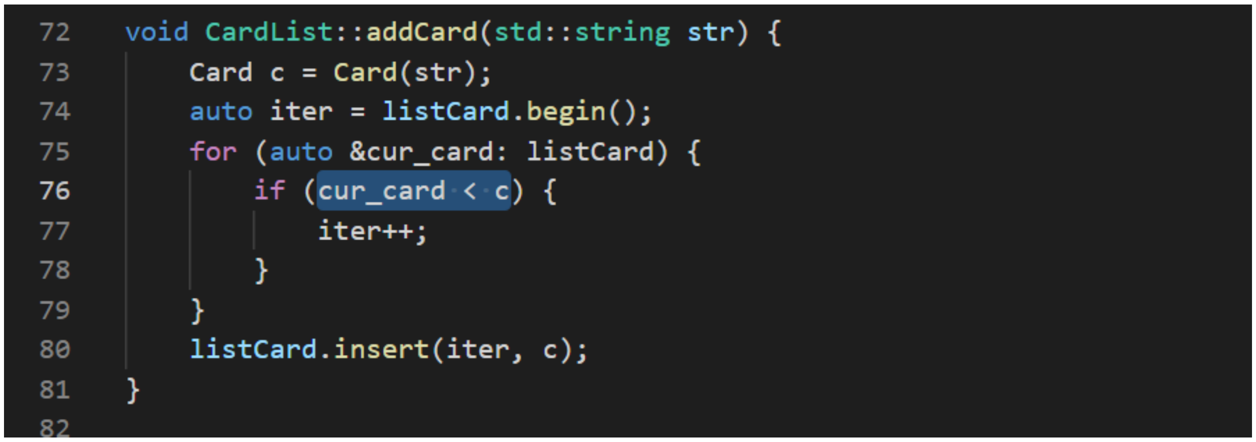
Design

- **Cohesion:** for easy management and easy upgrade in the future, several classes are created, with only fields and methods related to the class. For example, Card class only has getName(), getRank(), getSuite(), getOrderedSuite(), and operator<. It does not include anything unrelated such as playcard() function. I also tried my best to name fields and variable at most related to what they do. This way, when I have a problem somewhere in the code, I can easily navigate to the potential files that has the problem easily.

- **Coupling:** if compile from scratch, the program will need around 15s to compile. This is long compared with normal compiling time of assignment. My solution to reduce compiling time is building my classes depends on only one or two other classes, so each file only depends on one or two other files, so when re-compile, we only need to compli the changed file and its dependency. In particular:

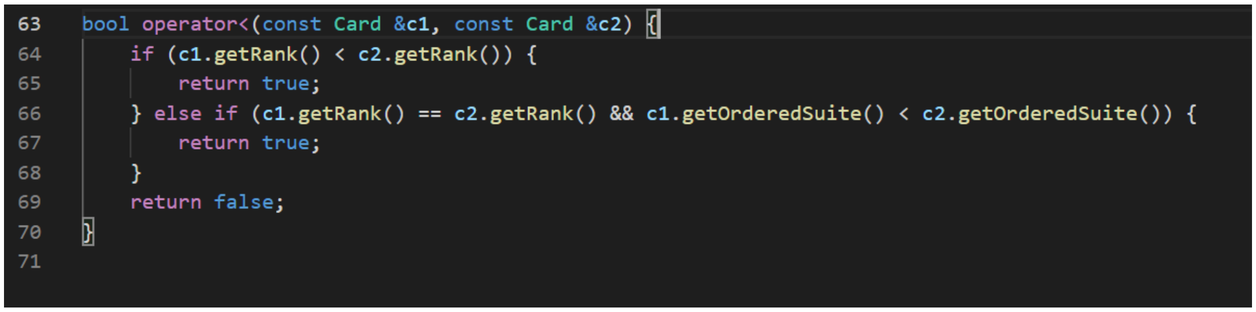
- main.cc depends on straightGame.[h|cc]
- straightGame.[h|cc] depends on deck.[h|cc] and player.[h|cc]
- deck.[h|cc] depends on cardList.[h|cc]
- player.[h|cc] depends on cardList.[h|cc]
- cardList.[h|cc] depends on card.[h|cc]

- **Operator override**:
  In cardList.cc, addCard function, I need an way to compare cards so I can sort them in order:



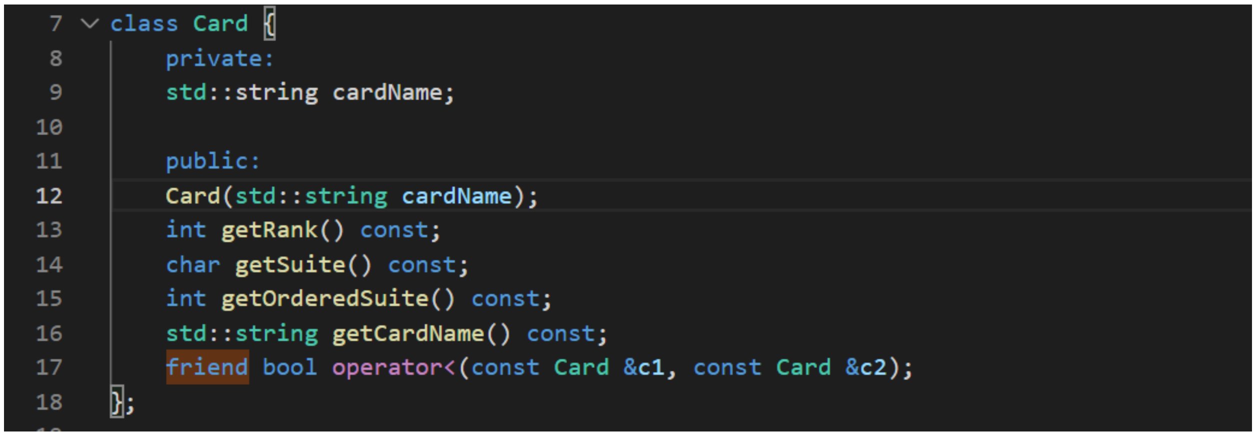
```
void CardList::addCard(std::string str) {
    Card c = Card(str);
    auto iter = listCard.begin();
    for (auto &cur_card: listCard) {
        if (cur_card < c) {
            iter++;
        }
    }
    listCard.insert(iter, c);
}
```

I realized Card is the class that I implemented, so I have to write a compare function myself. Since this is comparing two Cards, it should be implemented in Card class. I choose to write operators instead of a function because comparing two cards will be use a lot more in the future, and writing operator is more intuitive, much easier to remember the syntax.



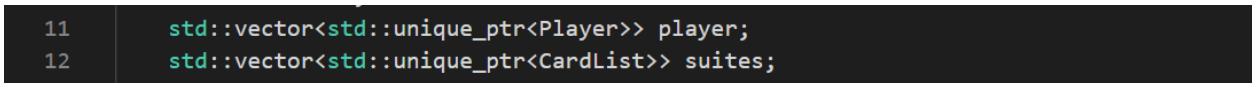
```
bool operator<(const Card &c1, const Card &c2) {
    if (c1.getRank() < c2.getRank()) {
        return true;
    } else if (c1.getRank() == c2.getRank() && c1.getOrderedSuite() < c2.getOrderedSuite()) {
        return true;
    }
    return false;
}
```

- **Encapsulation and the use of accessors:**
  I used a lot of getter and modifier functions, while leaving the fields private or protected to ensure encapsulation. For example, in card.cc, my cardName field is private, and the moteds are all gettors so extract information from the card.

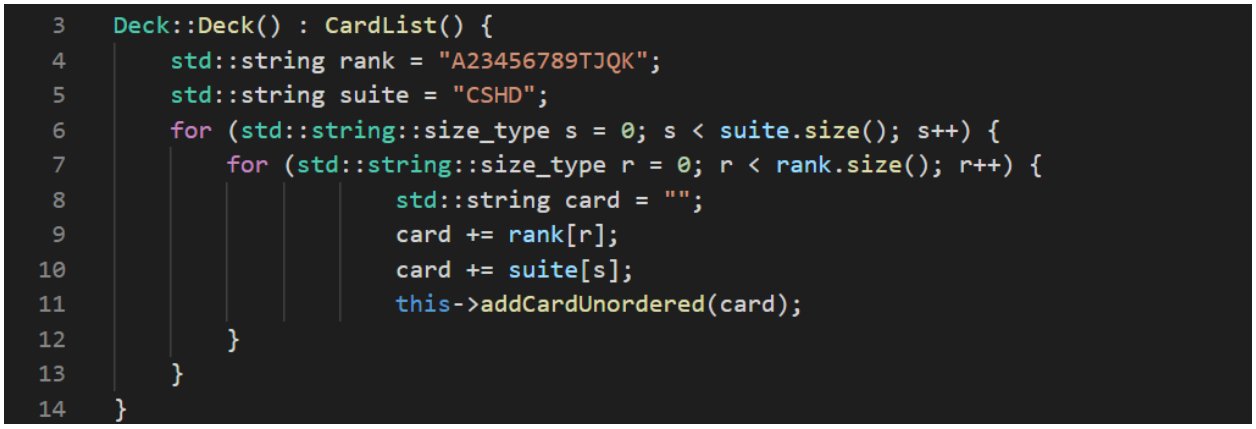

```cpp
class Card {
    private:
    std::string cardName;

    public:
    Card(std::string cardName);
    int getRank() const;
    char getSuite() const;
    int getOrderedSuite() const;
    std::string getCardName() const;
    friend bool operator<(const Card &c1, const Card &c2);
};
```

- **Prevent memory leaks**:
  I ensure to not leak memory by using only unique pointers throughout my program. This way, the pointer will be deleted automatically when it runs out of scope. An example is the pointers are in StraightGame object, points at Players and CardList objects specificly. This is a snippet of code in straightGame.cc


```cpp
    std::vector<std::unique_ptr<Player>> player;
    std::vector<std::unique_ptr<CardList>> suites;
```

- **Constructor:**
  Because of encapsulation, I can only modify object fields by getters, constructors, or by assignment operators. I prefer using constructor and modifier because it does not requires to write the full 5 operators li assignment operators. One difficulty Is when I build a 52 card deck, I must construct 52 cards, which may be a lot of repeated code. I managed to short it up to 12 lines of code by using for loop and constructor. This is a snippet of code in deck.cc.


```cpp
Deck::Deck() : CardList() {
    std::string rank = "A23456789TJQK";
    std::string suite = "CSHD";
    for (std::string::size_type s = 0; s < suite.size(); s++) {
        for (std::string::size_type r = 0; r < rank.size(); r++) {
                std::string card = "";
                card += rank[r];
                card += suite[s];
                this->addCardUnordered(card);
        }
    }
}
```

- **RAII:** I used unique pointers instead of raw pointers in my program.

- **Human and Computer player:**
  Originally, I wanted Human and Computer classes are is a generalization of Stategy class, to use strategy design pattern. However, my program changed, so I decided each player to have two unique pointers, one points to Human object and one points to Computer object, and there is a boolean value isHuman to decide if the player is a Human or a Computer. This way, the price is still relatively low since they are only pointers, and we can still modify isHuman filed to change the type of player dynamically during the game.

Resilience to Change

- **Change the number of players:**
  This change will take place in straightGame.cc. All players are managed by a vector of size 4 of unique pointers to a Player object. To change the number of players, we need to change the size of Player object pointers vector, then change all the loops limit related to players in straightGame.cc to the number we want.
  If we want the player to tell us the player they have before starting the game, ie we do not know how many player we want, in straightGame.cc, we can create a field call numberPlayer that stores the number the player want, then use it in all of the for loops related to player and set it as the size of the Player object pointer in Straight.cc

- **Change from computer player -> human player:**
  After a round without a player determined, but another human friend wanted to play, we may want that human friend to replace a computer. We can do that by asking in the beginning of each round if they want to change any computer to human player, and if yes, then we change isHuman boolean field in straightGame.cc which decide if the player is human or computer by using a mutator, let's say setSategy(). This way, we can ensure encapsulation.

- **Change rules:**
  The rules to decide which move is legal is in the legalMove() method in player.cc. If we need to change the rules, we just need to change the implementation of it.

- **Add GUI interface:**
  This is a tricky one. Since my display method are speeded among three different classes, I will first collect those display() method to one single class, call ViewController, then modify each method to show some GUI properties. I must also include a GUI library into the scope.

Answer questions

Q1: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

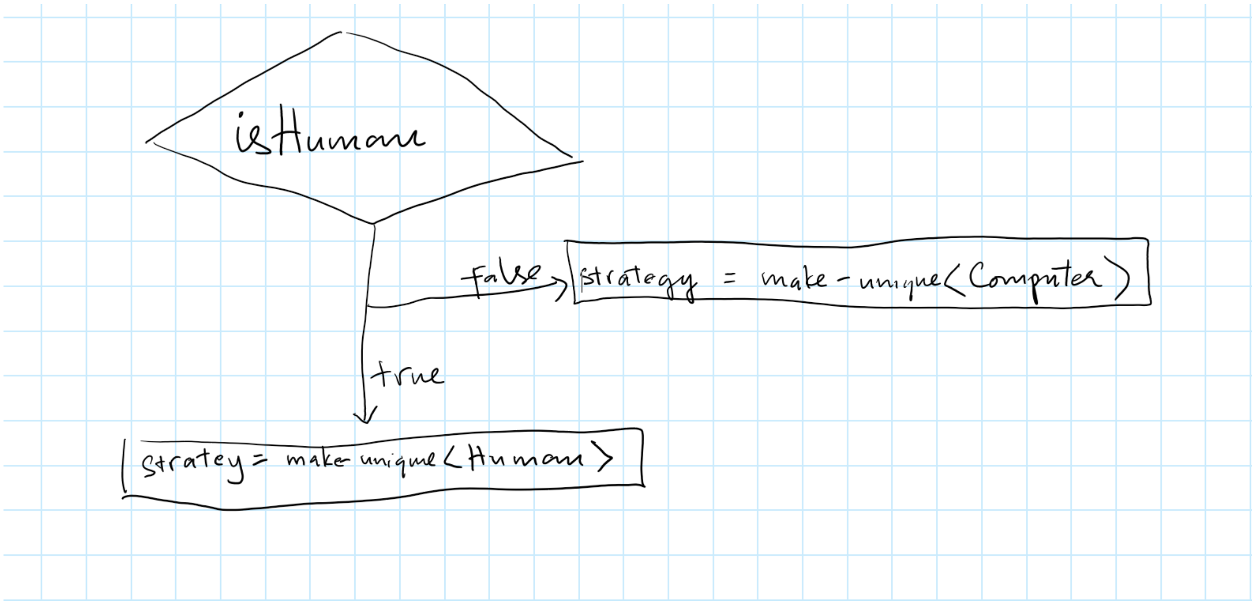
Answer:

I originally wanted to use strategy design pattern, however, due to some difficulties, I was unable to do so. My solution is each Player has two pointers, one points at a Human object, ones point at a Computer object, and have a boolean value to decide which pointer to use. By doing this, my design will not change a lot compare with the original plan, because by the time I failed strategy design pattern, I had perfect Human and Computer class setted up.

Q2: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

Answer:

Of course it is reasonable to replace a human with a computer when the human does not want to play anymore. I have isHuman boolean value that decides if a human is playing or not. To associated a human player to the computer player, I is need to set isHuman to false, then the player will switch to computer mode. A decision tree is shown below.


isHuman
false
strategy = make-unique<Computer>
true
stratey = make-unique<Human>

Extra credit

I did not do any extra rules or extra interface. I only used unique pointers instead of raw pointers to ensure RAII.

Final questions

Q: 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:

I worked on this project alone. This is my first time designing a multiple files program! This is such a big jump because the biggest question when I joined a hackathon a few months ago was how to connect files together to create one functional program. Now, did the whole "connecting" by myself, I realized there are so much more planning behind. Before coding, I needed to plan what exactly I would do, which file which class should be connected, which order to write so that we can test our functions of this class as soon as we finish, without waiting for other classes to be done. I needed to do decision such that use set or vector to use. The biggest lesson is that although you planed as detailed as possible, the plan will always be changed because there are so many things you do not know until you star coding. My original plan was changed at least 50% throughout the process. Also, debugging sucks, and it is okay. I also realised that doing a project is not that intimidating. I have always been procrastinated from doing a side project. I will definitely do a side project during the break.

Q: What would you have done differently if you had the chance to start over?

Answer:

Oh, oops, good question. One thing is I will not dodge the big 5 mutator rules so that my code can be cleaner. Also, I will try to build by design to corporate MVC design pattern into it, so I can easily implement a GUI interface. I will also spend time to make my exception cleaner, and my function has some kind of guarantee. Also, my strategy design pattern did not work. I wished I could spend more time on it to make sure it works perfectly.

Conclusion

I feel good with this project. I learned so much in this course. Thank you guys and have a great Christmas holiday!