Project 2

Title

Poker

Course

CSC-17C

Section

47468

Due Date

December 21, 2025

Author

Ivy Fudge

Introduction

I am coding a simple poker game. I chose poker because I enjoy playing it a lot and I made a blackjack game last semester, so I was already a little familiar with the general flow of a card game. I spent about 20 hours working on it, with about 1000 lines of code, 3 classes – the deck of cards, the players, and the dealer – and an enumeration for the possible hand types.

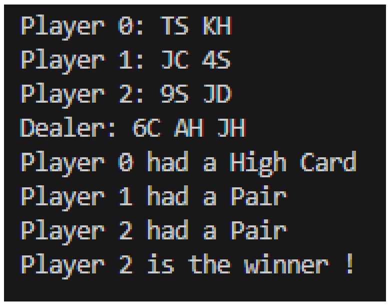
Game Rules

All players are initially dealt 2 cards each, then the dealer draws three cards for the table. Each player's score is then calculated based off of what hand type they are able to make using their cards and the table cards. The possible hands are Straight Flush, Four Of A Kind, Full House, Flush, Straight, Three Of A Kind, Two Pair, Pair, and High Card. To find the winner, the hands are first scanned for the best hand type. If only one person has the best hand type, they win. If there are multiple of the same type, then the hand scores, which are different per hand type, but are usually the highest face value of the hand, are compared. The person with the highest score wins. Now another game can be played.

Description of Code

The main file holds the game functionality and the game start. It creates a deck, 3 players, and a dealer. The player and dealer classes take a pointer to the deck, along with an ID number as arguments. The dealer class extends the player class to have a specialized show hand function. The deck is a queue of integers that act as the deck to pull cards from, with a set of the integers already in the deck. The player has a list of integers for its hand of cards, a counter for how many cards it has, a pointer to the deck, and an integer ID. In the main menu, there are 5 options available: play poker, get chips of each player, run Hash Table demo, run AVL Tree demo, run Graph demo, and exit. After selecting play, another menu is given, asking: play, display deck, shuffle deck, sort deck, and stop playing. During the game, 2 cards are dealt to each player, and 3 to the dealer. Then each player analyzes the cards available to them and finds the hand type they have, and an extra score alongside it. Then the hand types are compared between players, and whoever has the best hand wins. After the round, the deck will have less cards; If it does not have enough for another round, the user will be asked to choose how to reset the deck. This resolves to shuffling it or shuffling it then sorting it. This all is the majority of the functionality of the project.

Sample I/O



```
Player 0: TS KH
Player 1: JC 4S
Player 2: 9S JD
Dealer: 6C AH JH
Player 0 had a High Card
Player 1 had a Pair
Player 2 had a Pair
Player 2 is the winner !
```

Checkoff Sheet

Container classes

Sequences

List: The player’s hand is a list<int> to hold the card id’s, which hold the card face and suit.

Associative Containers

Set: The deck uses a set<int> to hold the numbers already put into the deck when shuffling

Map: The player uses a map<HANDS, int> to organize the outputs of the function that calculates a player’s hand type and score. This is then passed back to the main function to find the winner.

Hash: There is a simple demonstration of a hash table accesible from the main menu. It shuffles a deck, puts all of the cards in it into a hash table, displays the hash table, and then checks 8 cards to see how many carfds were in front of them.

Container adaptors

Stack: I use a stack of Player objects to hold the players that are in the game. This can be used to easily remove or add players from or to the game.

Queue: The deck uses a queue<int> of cards in the deck. This is used to easily deal new cards, that will not be repeated within a run of the deck. The deck persists between rounds so that it can be analyzed or used for more strategies. If the deck does not have enouygh cards for a round, it will prompt the user to shuffle it, and sort it if required.

Algorithms

Non-mutating algorithms

Find: Used in the Deck class when shuffling to ensure duplicate cards are not added to the deck. used.find([card]) is called to see if a card has already been added. If it has, a new card is generated. If not, that card is added to both the deck and the used set.

Organization

Merge: A merge sort algorithm is used for sorting the deck. It sorts the cards first based on suit, then by face value, which is just an effect of the cards' face and suit being derived from modding or dividing their card id.