**Blackjack**

**Introduction**

I coded blackjack for my project due to it was a WIP that was for my 17B class this semester. When I originally created it, most of the code ran on vectors. I decided to retrofit it so that it’d run on queues, maps, sets and etc. or other containers. I spent about a week retrofitting the code so it’d work with the containers asked. There’s approximately 1200 lines of code in the project. It contains 7 classes which are Player, Login, Deck, Card, Hand, Bet, and Table. The login allows the user to create their own ID to save their score/money. Player is the unique ID. Cards form a deck, which is used in table in order to play the game. Hand handles the game logic for what a hand can do, and table is what brings it all together for a menu. The code can be found at [https://github.com/albuut/Blackjack17c](https://github.com/albuut/Blackjack17c%20).

**Approaches**

How I approached the project was I started from a base, like player and card. The player will be used by login, so most of the variables and functions will be to modulate the player. I then created cards, which I know would be used by a deck and hand since they’ll hold these values separately with their own logic. Afterwards I created a bet class in order to track my bets. All of these will go into table and allow the game to play out with it’s rules.

**Game Rules**

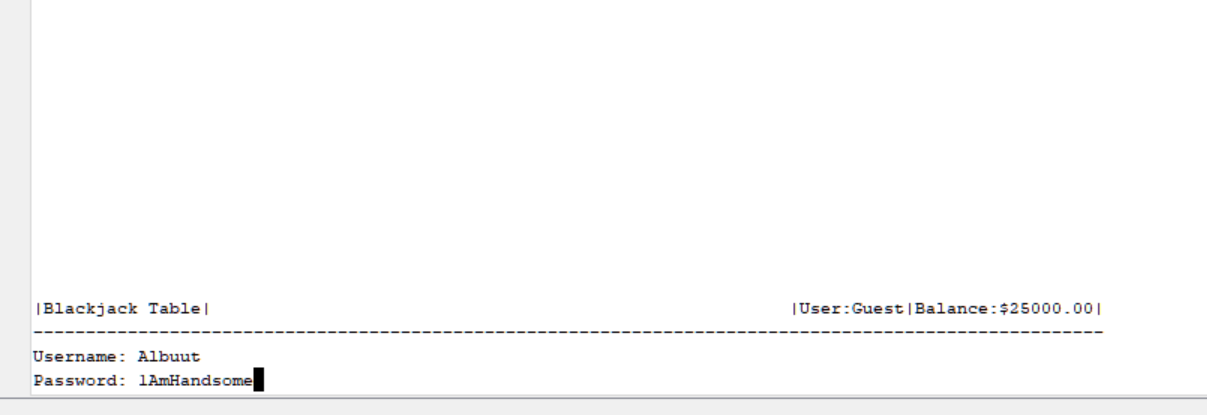
The goal of blackjack is to out-score the dealer’s hand without going over the value 21. When you first play, you enter with an account, afterwards you choose a value $100-$1000 for the size of your bets. Then, you would have chosen how many hands you would want to play, 1 through 5. You would then be dealt two cards for each hand. You’ll be given an option to double down, which is to double your current bet and only draw one more card, or you could choose to hit which would add another card to your hand. The values of each card are the same as their number shown, Ace has a value of 1 or 11 and the face cards have a value of 10. If you are to get blackjack, 21 on draw, then you automatically win that hand with a pay out of 3:2. The dealer also draws two cards in the beginning, but one is hidden from you. If the card that’s revealed is an ace. It gives you the option to do even money or insurance which would essentially, you’re gambling if the dealer has a blackjack. You keep hitting your hands until you decide to stop, when you finish hitting all your hands. The dealer will reveal their card and deal as many cards as they can until they reach the value 17. Once they hit a value over 17 they’ll stop. At that point there will be a showdown between your hands, if your hand value is larger than the dealers without going over 21. You win your bets.

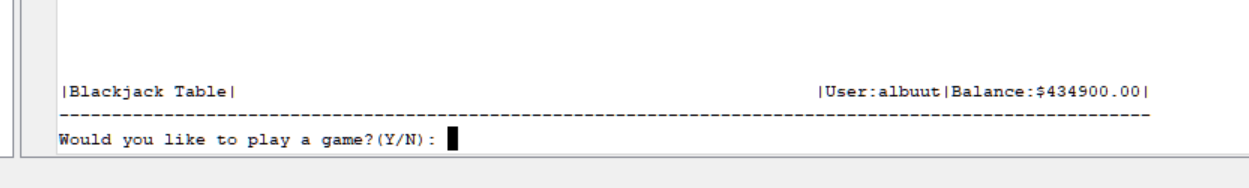
**Descriptions of Code**

The code is separated into 7 classes and a main. The 7 classes are Player, Login, Deck, Hand, Table, Bet, and Card. Card is what contains the point values and the suit of each individual card. Deck and hand are classes which hold the cards. The purpose of deck is to shuffle them and distribute the cards in a random manner, Hand holds cards that the player would use and it has different logic steps in order to evaluate what action you can do, such as hitting, stand, double down, insurance, or even money. It also has the ability to calculate your hand total. Bets are the values of which you decide to bet. It also determines how much you win or lose per hand, for example if you hit blackjack, you get a higher ratio of winnings rather than the 1:1. You get 3:2. It essentially holds the checks if you won a bet or if you tied. Player stores your username and password. It also stores how much money you have won. It also internally stores how many hands you’ve played and won. This could be useful for later on if you wanted to have a state sheet or if you needed to detect cheaters due to observing abnormalities in a user. The table is essentially what runs the blackjack logic. It passes allows you to pass the cards drawn from your deck into hands, it checks if your hand wins compared to the dealer, it also is the logic which allows your dealer to play cards. Afterwards it’ll have a run through in order to observe which hands you won and properly divvy up how much money you’ve won or lost.

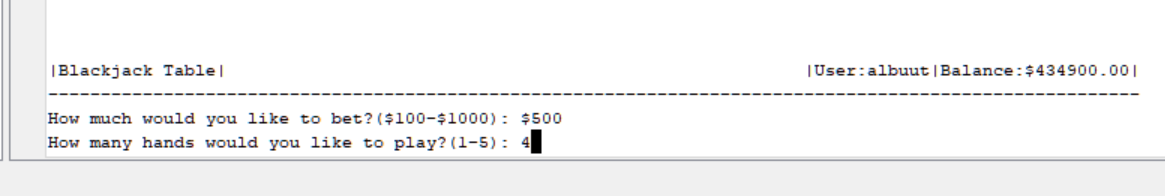
**Sample Input Output**

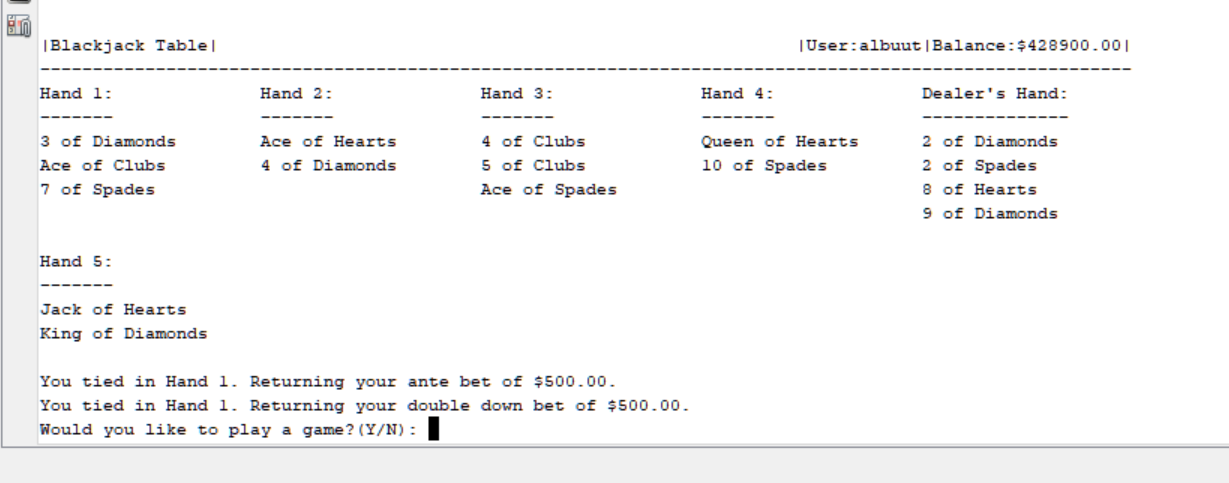
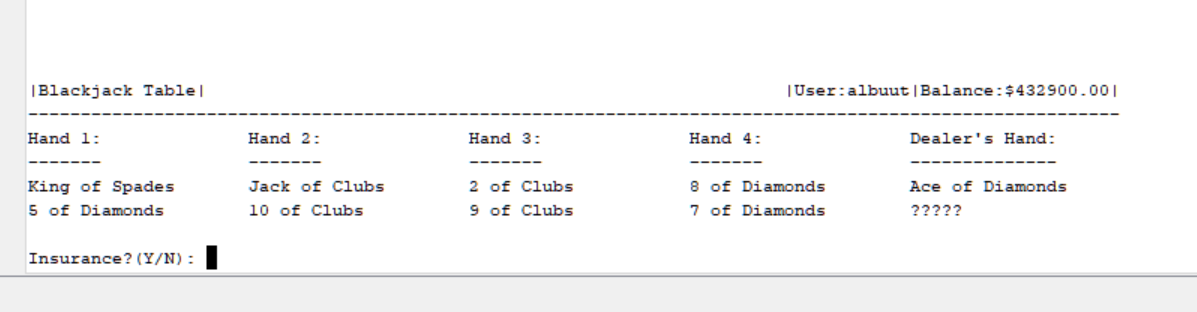
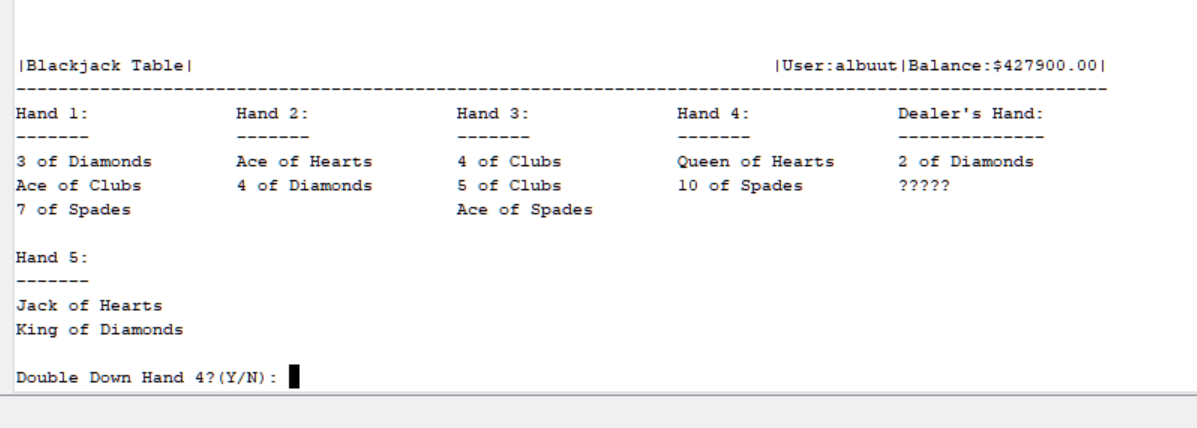
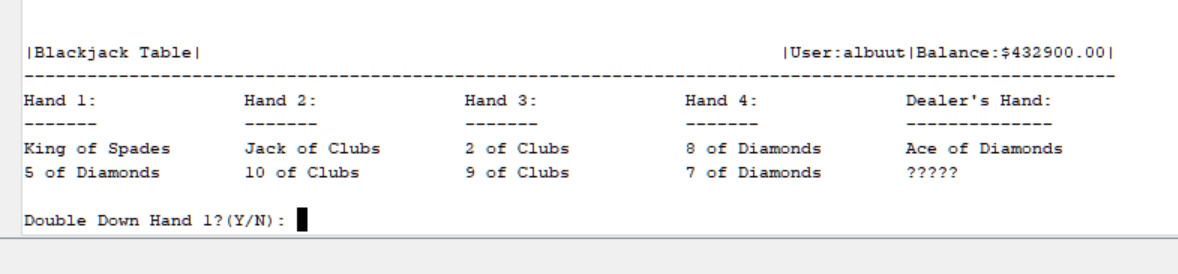






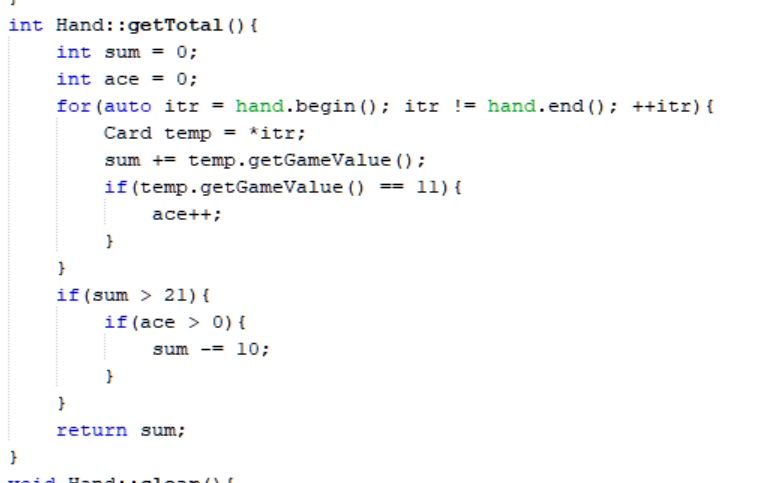






Checklist

**Sequences:List**

****

I use list to form my hands so I had had a scalable way to connect my data/hands due to the restriction on using vectors. By using a list I can have it scale up and allow me to iterate through the values quite easily. As seen above, it allows me to go through each hand value to find the game value to sum. I’m using a Forward iterator due to I’m assigning the \*itr to a temporary card and extract it’s game value from it. I also want to retain my hand values and not delete them each time I want to iterate through my list. Hence, why I didn’t want to use a queue for my hands.

**Associate Container: Map**

****

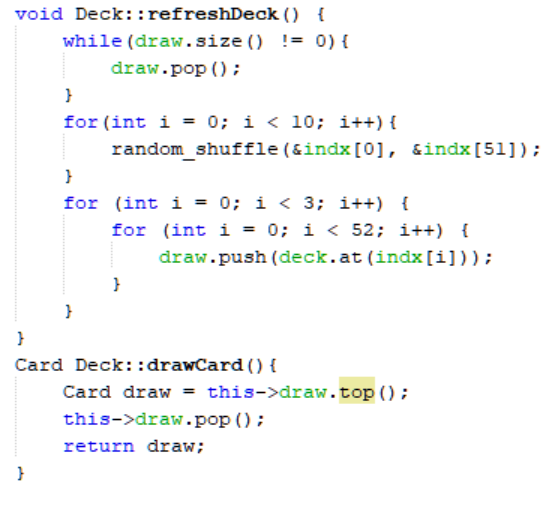
I used a map for my login class. By having all the files read into a map with the email as the key. It allows the program to browse through the map of Players to see if it contains the Username. If not, it means the map does not contain this email. If you observe Validate and Check login, they’re both used in order to check if either there exist an account with that username to allow the user to either create a new user or to login. I also used a map to hold my hands so that I had a scalable location to hold my hands which hold a hand object. I also did the same with deck in order to easily call upon a card. For my write functions, I used an input iterator which allows me to write out the values inside my maps. I also use a forward operator in order to read the values from a file and assign them into my map.

**Associate Container: Set**

****

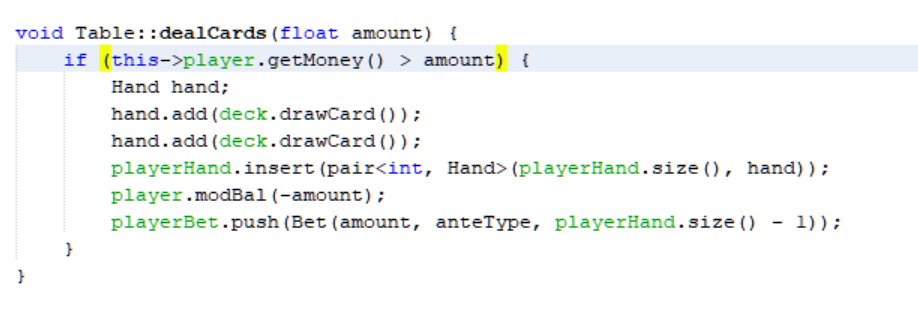
I used a set to hold the index of which hand. I wanted it to be scalable like map for hands, due to the addition of having splits whenever the hand plays. Since the size can increase it’ll allow me to go through my whole hand if I was to add one mid-way. A set also can only contain unique values, by using a set. I can always guarantee that I won’t be interacting with a previous hand. I use an input iterator here due to I just need to output the index for which the current turn corresponds to.

**Container Adaptors: Stack**

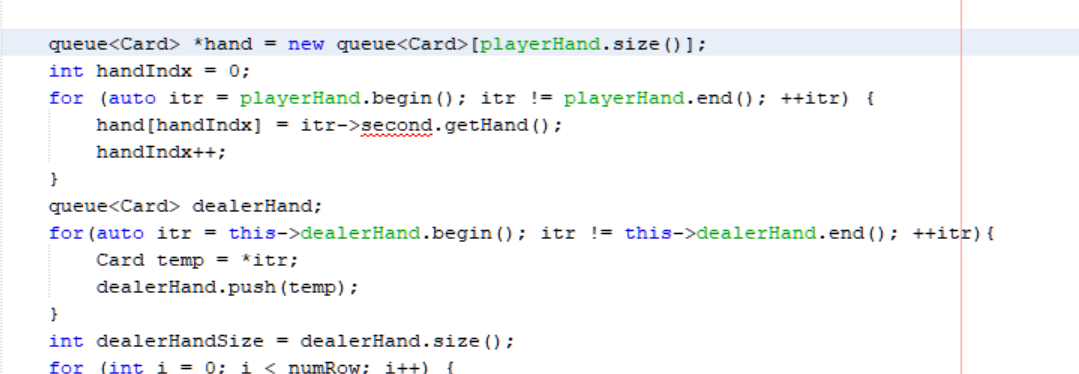


I used a stack to place random cards into it. The order in which the cards come from doesn’t matter much. I just needed a container which would be able to add itself into the container and be removed when I use it. So by putting various cards into the draw stack. I can hold the top value in a temp, remove it, and then return it when using drawCard(). I don’t need to have the knowledge of what the card is. I just need to return it to whoever calls for that value. I didn’t use any iterators for this container due to the values I input in are independent of order in the stack and I just need to grab whatever value is on top.

**Container Adaptor: Queue**

****



I used a queue to store my bets. The only time I’ll use them is when I store them into an array or when I go through a check to see if the user has won. The data is not read to often, so having it constantly being held somewhere is not that useful. I also use the queue for my print functions. Since I don’t need to really store them and hold them for a long time. I just used a queue due to it follows the structure of adding to the back, and popping off the front. This allows me to keep my data ordered and be scalable depending on what size I want it without having to due constant checks on sizes. It also allows to remember innately the order in which these values have been inputted or pushed in. Queue also doesn’t require me to use the different iterators due to everything is in order and I’m just reading whatever is the top value and popping it off.

**Iterators: See the section for the containers for each iterator.**

**Trivial Iterator:** None, this is already handled by the STL

**Input Iterator:** Write for logins and calculating the total of hand, values from the hand order

**Output Iterator:** I didn’t have to do any blanket masking of values or changing values to be all the same only. There wasn’t any reason where I needed to go through a container and just set it equal to some value.

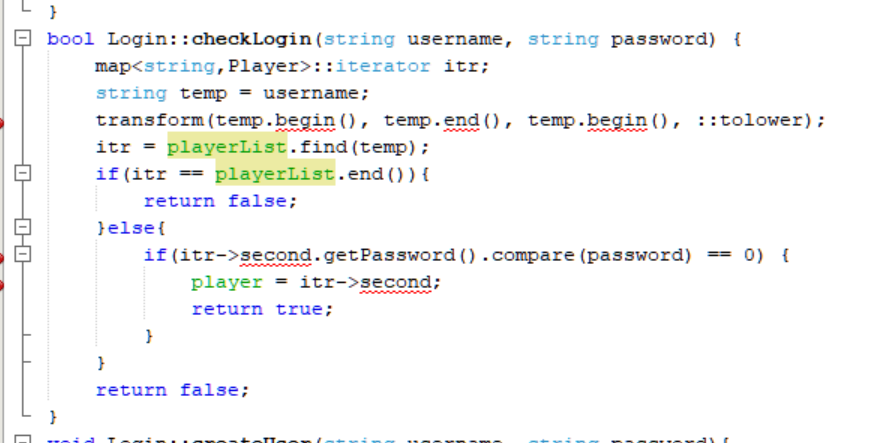
**Forward Iterator**: I used to read from logins and to write into a map because I needed to manipulate specific objects or values held in the container.

**Bi-Directional Iterator**: I had no reason to go back and forth between my containers due to sorts were already used in the STL and I didn’t have to search for any values. I just needed the ability to iterate in one direction in order to calculate sums or determine what value gets manipulated.

**Random Access Iterator:** Since I already shuffled an index array and place into a stack, using random\_shuffle. There isn’t a real reason for me to be randomly accessing the array. This would of essentially been contained in there.

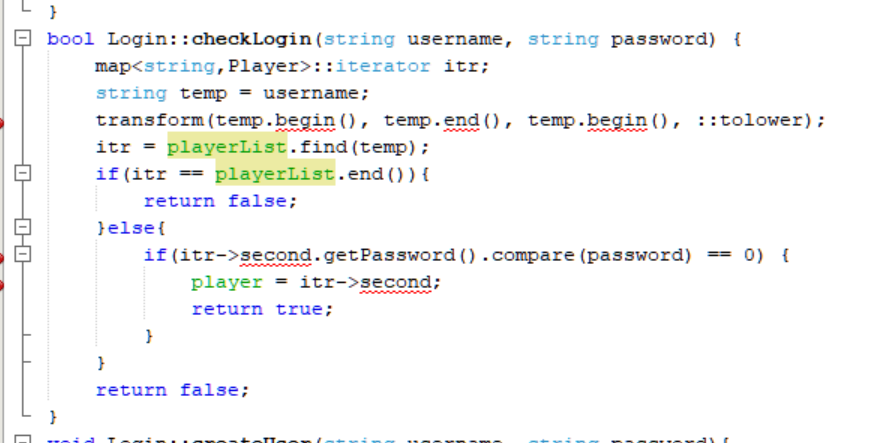
**Algorithms**

**Non-Mutating Algorithms: Find**

****

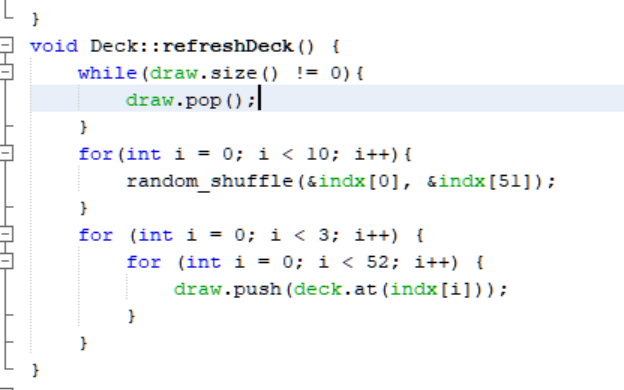
I used the find algorithm to search through the keys of my map. I used a string to construct my keys which hold the username. Therefore, whenever I look for player I can search through my map by looking at the username as the key when I input it. This is fairly useful due to it tidy’s up a large chunk of code for data validation and finding specific values. This allows me to search through the map in order to confirm if the username is taken or if there is someone with this username without having to access the Player object.

**Mutating Algorithms: Transform**

****

I use the transform algorithm to convert my usernames into a lowercase version. By having it as a lowercase version from the input, I can have a non-case sensitive input in which I can compare if they’re a match. This is useful due to users may randomly or arbitrarily capitalize letters. It also is quite useful due to having a username being unique due to case-sensitivity would look unusual. I also only applied this on my username due to having it on password would be absurd, having that case sensitive is quite useful and make it harder for someone to brute force the password by opening it up to more permutations.

**Mutating Algorithms: random\_shuffle**

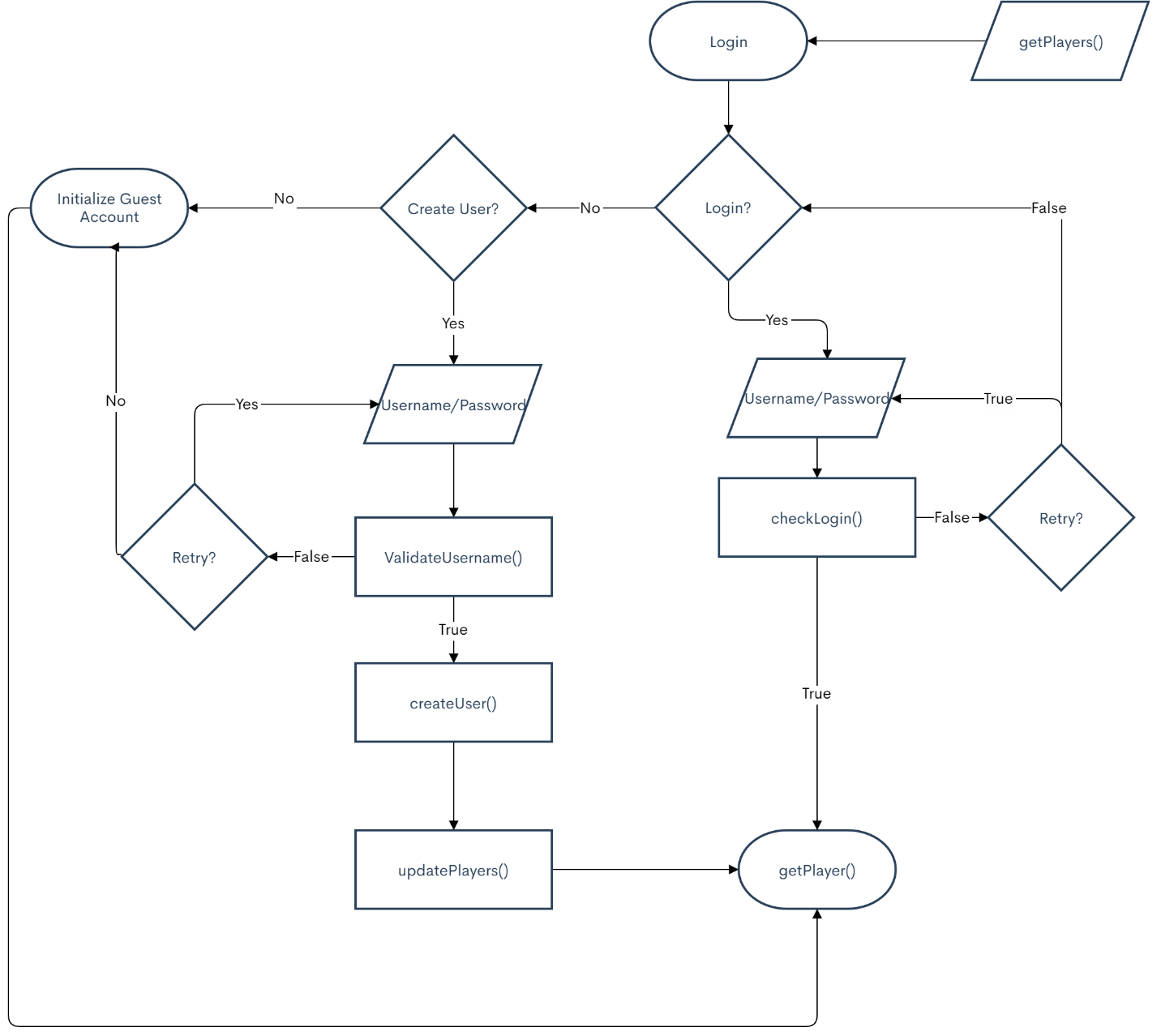
****

I used a random shuffle on an index which contains the keys to the map of my cards. Each key leads to a unique card. So I random\_shuffle the array 10 times in order to get a randomness which would make me happier/content. This allows me to push cards into a stack randomly to be distributed randomly to player and dealer hands. This is a vital part of playing the game blackjack. If cards came out in the same order every time and was predictable. The house would always lose money then due to you could just decide when to bet based on what card is played.

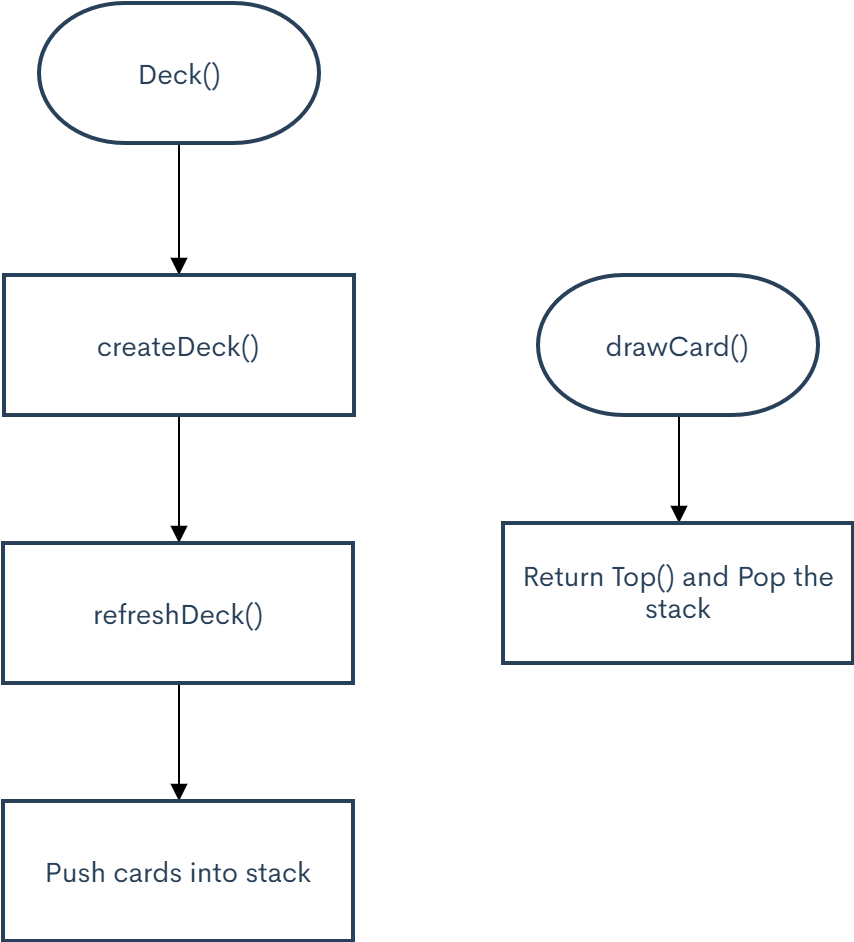
Code Documentation:

Flowcharts

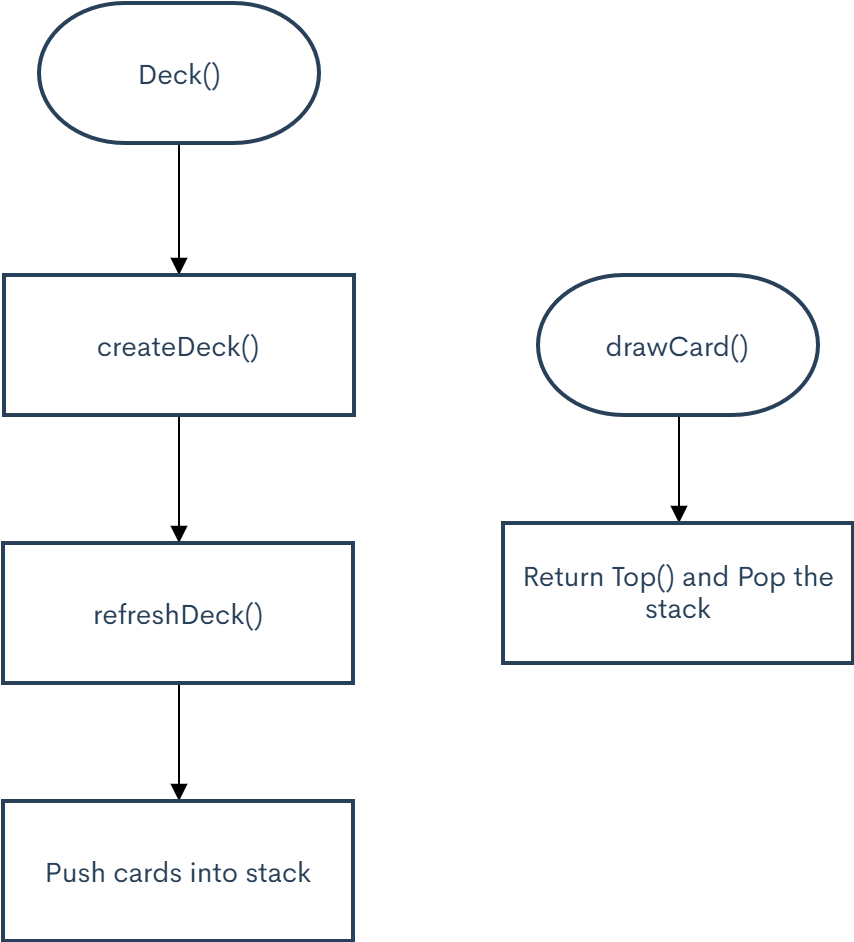
**Login.h**



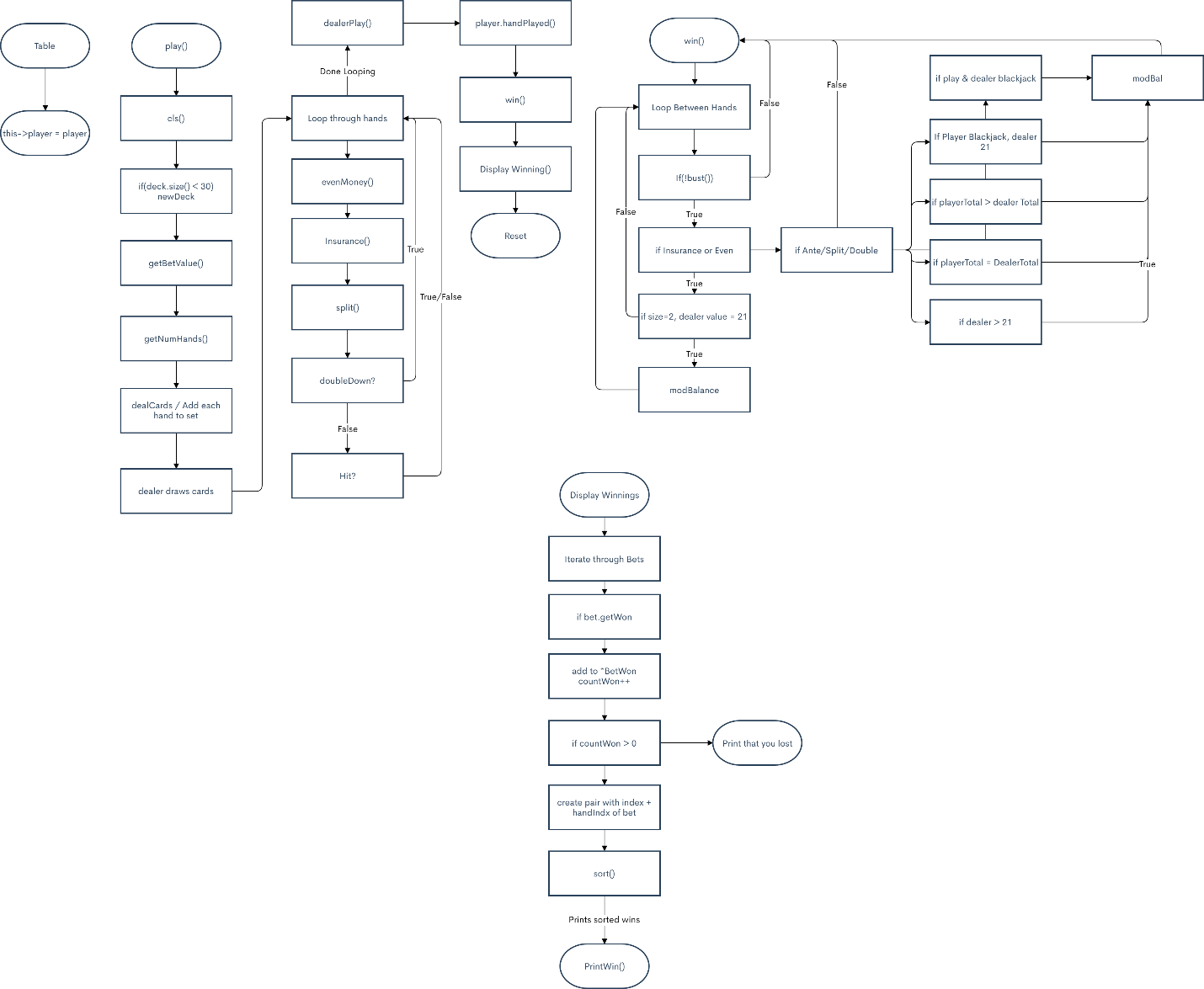
**Deck.h**



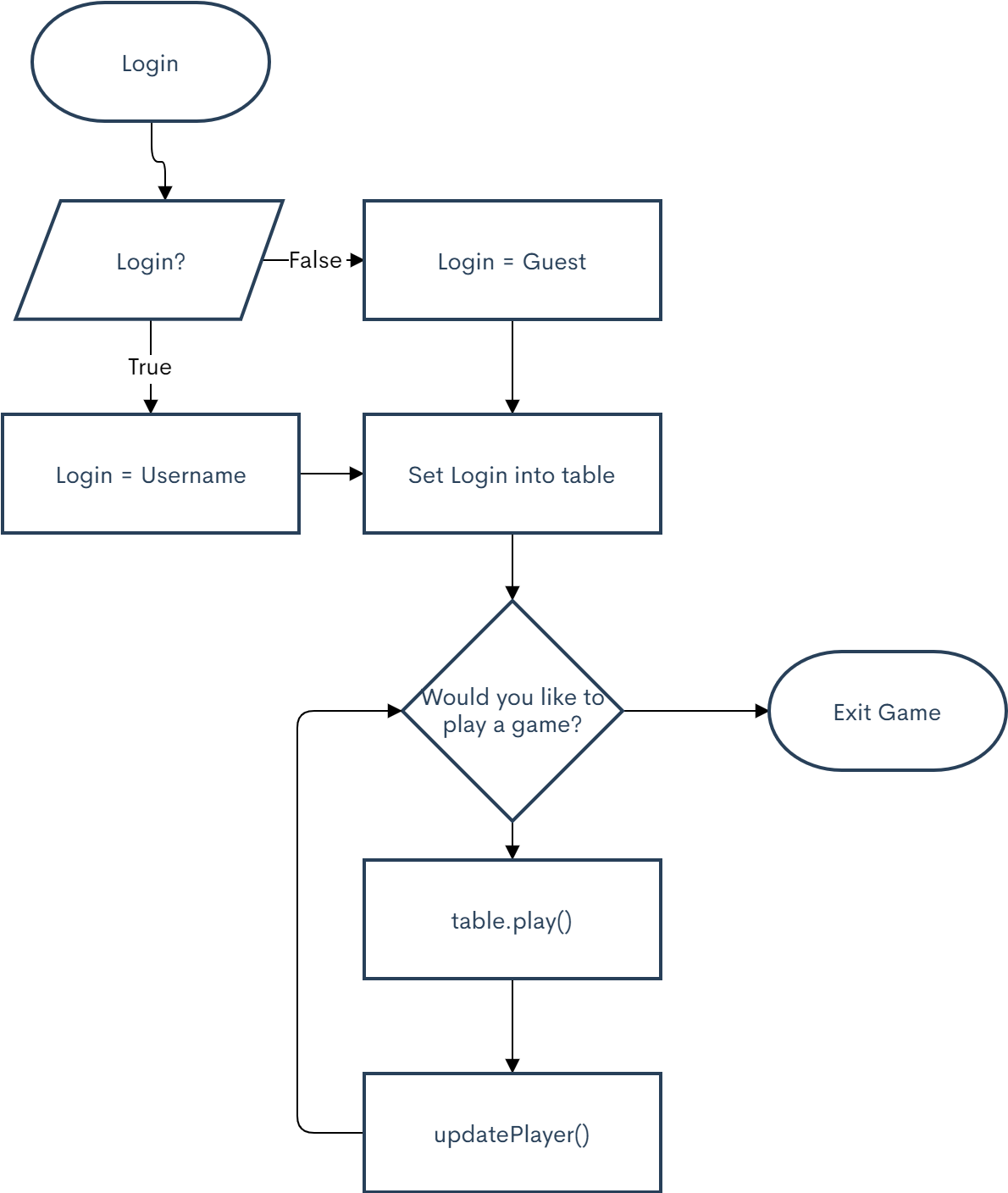
**Hand**



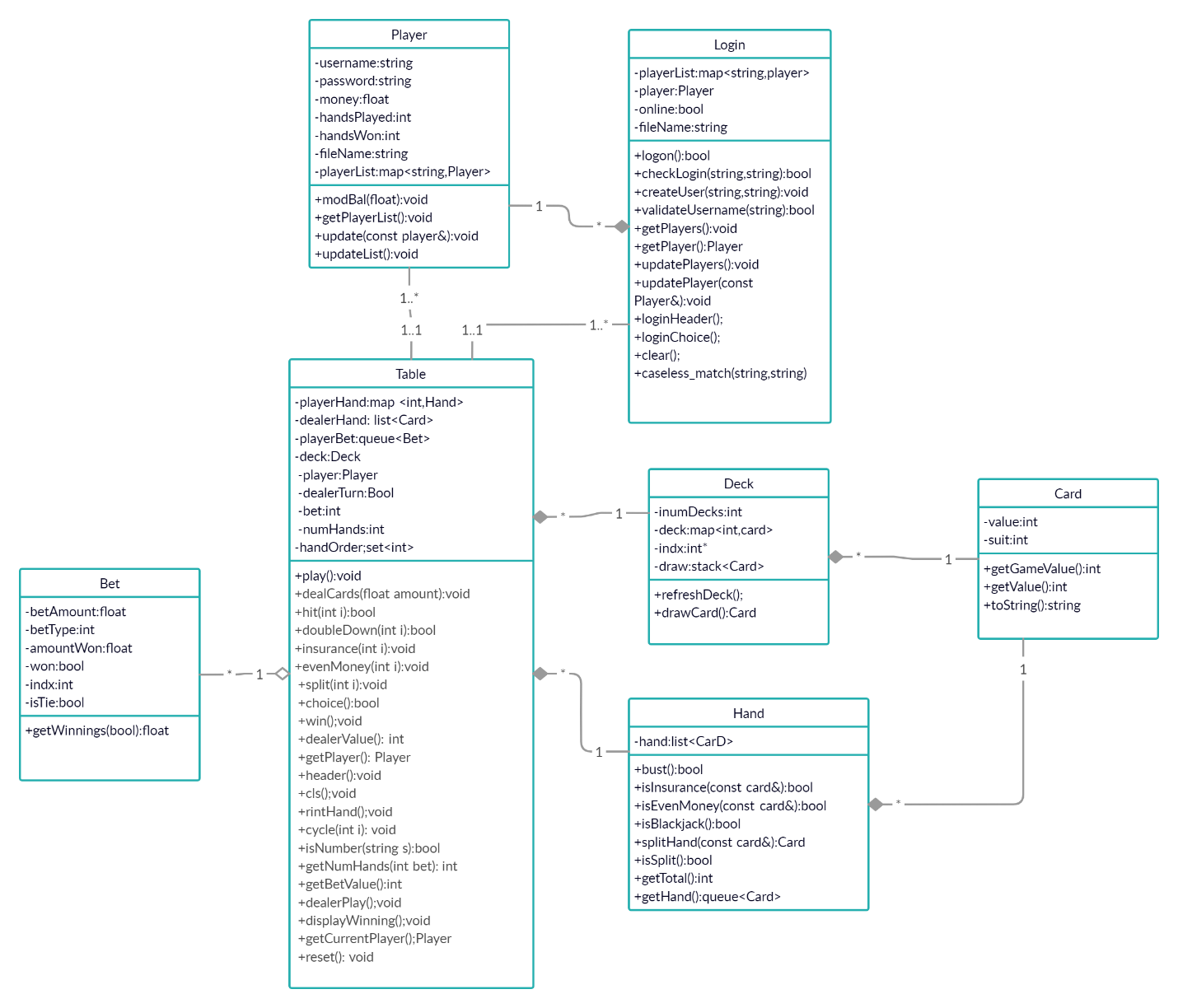
**Table**



**Main**



**UML**



**Psuedo Code:**

**Card->Deck**

**Card->Hand**

**Player->Login-**

**Player->Table**

**Login->Table**

**Bet->Table**

**Deck->Table**

**Hand->table**

**Table**

**Play**

**Create a deck**

**Create hands**

**Put Hand into container**

**Get amount for bet**

**Put bet into container**

**Play through Hands**

**Ask DoubleDown/Hit/Insurance/Even Money**

**Make more bets depending on option**

**Dealer Plays**

**Hit til 17**

**Check Bets**

**If player = win**

**Bet = won**

**Print Winnings**

**Reset Board**

**Play;**

**Card = Value/Suit**

**Deck = container of card,deals cards**

**Hand = container of card with logic**

**Bet = holds bet amount, type, and if won**

**Login = create or login user**

**Player = amount won, hands played, username, password**