Design:

- *class Card;*

The most basic unit of this design is the instance of the class Card. The Card class contains the value that each Card would hold. The value to each Card is assigned to it by the Deck class.

- *class Deck;*

The Deck class contains an array of Card objects.
The Deck class contains a value to indicate the number of suits in the deck.
The Deck class contains a value to indicate the number of cards in each suit.
The Deck class contains an unordered_map to hold the mapping of consecutive cards in a shuffle.

The Deck constructor is designed to accept a set number of Suits and also the number cards each Suit would contain. This ensures that the Suits and the number of cards in a Deck could be varied according to the user's preference. The constructor then creates a set of Suits*NumberOfCardsPerSuit Cards and places them in it's array of Cards.

- The Deck then assigns a card value to each Card in it's array. This value is calculated using the below formula

cardValue = currentSuitNumber*MaxCardsPerSuit + currentCardNumberInSuit

So, the 4th card in the 2nd suit would have a value of

cardValue = 1*13 + 4 = 17. (Assuming that Suit numbers and card numbers start from 0)

- Shuffle algorithm:

The shuffle algorithm implemented here uses the popular Fisher-Yates algorithm. The algorithm goes through the array starting from the last element in the array. A random number within the un-shuffled part of the array
determines which element would be swapped with the last element in the un-shuffled part and the last element is moved inwards by 1.

This method is repeated until the iteration reaches the first element.

- Main Loop:

The problem statement is to run the shuffle algorithm twice and find a sequence of two consecutive Cards that are common between the two shuffles.
Implementation:

1. Shuffle the card deck
2. Store this shuffled card deck in an unordered_map called shuffleMap(hashtable) in the format <cardValue : nextCardValue>.The idea behind storing these values in the shuffleMap is to have quick access (O(1)) to each Card. This is needed in the step 3.
3. Shuffle the card deck again.

4. Now, Iterate over the reshuffled card deck, Check if the value for ith card in the reshuffled array from the shuffleMap (which gives the next card in sequence in the previous shuffle) is the same as the (i+1)th card in the current shuffled array.

5. If the values in step 4 are not the same for any ith value, then we have found a distinct shuffle. We break from the main loop and exit.

6. If any set of values are the same in Step 4, we store all the values of <card[i] : card[i+1]> in the shuffleMap. This is needed for the next iteration of the check between shuffles. Go back to Step 3.

Design Considerations:

1. The Card class could have been contained a Suit and Value set of variables which would identify each Card with respect to the Suit and Value it had. This design was not implemented since the problem statement deals only with shuffling the card deck which only needs the overall cardValue itself and has no dependency on the Suit and individual Value of the card.

2. The main problem statement requires us to hold the order in which each card was positioned with respect to the previous card in the previous shuffle. One way is to use a hash table to save this information.
The other way to do this is to save this information in the Card class itself like below:

```
class Card{
        int cardValue;

        int nextCardInPrevShuffle;
};
```

This would remove the need for the shuffleMap in the class Deck.

This design was not used to simplify the design of the class Card itself.