Nathan Roe
OOP with C++
EN.605.604 Section 81
2/27/2022

Module 4 Assignment: Poker Hands

Design

For this assignment, the project requirements state that the program should compare Poker Hands to determine the ranking and display the results. Based on the project description, I am designing the program to work with standard Poker hands available with a basic 52-card deck (no jokers) for High hand poker games (winner has highest hand ranking).

To do this, I will break the code into three objects: A Card, a Hand, and a Scorer. In this case, the card will be a very basic object similar to a node with required information for acting as a playing card. The hand will deal with functionality required for maintaining a hand of cards, and the Scorer will do comparisons between hands to evaluate scoring. I will then make a main program that reads Config files using Libconfig to set up cards and hands, and use the scorer to evaluate the round. The main program will then print the results to the console.

Card needs to:

- Track Suit
- Track Value
- Enumerate all Suits/Values
- Operators so that cards can be compared using >,<,==

Hand needs to:

- Store cards
- Receive dealt cards
- Discard unwanted cards
- Track Hand Type
- Enumerate all Hand Types

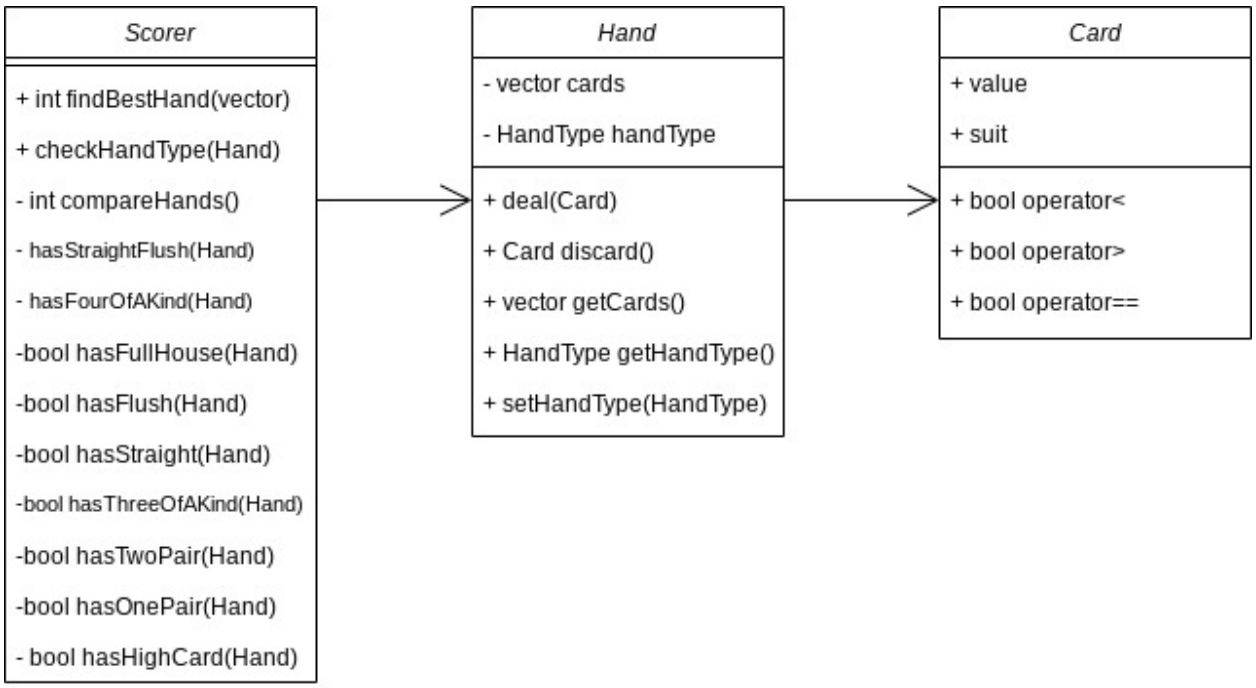
Scorer needs to:

- Evaluate sets of Hands to determine wins/ties
    - I would like to set this functionality up in a way that can evaluate more than 2 hands at a time so that it could be used for a full poker game. To do this, I will accept a vector of Hands to the function:
        - Iterate through hands, comparing each to the previous best hand
        - Move best hands to the front of the vector
        - Return the number of tied hands so that the caller can see which hands were the best

Nathan Roe
OOP with C++
EN.605.604 Section 81
2/27/2022

Module 4 Assignment: Poker Hands

- Check hands for each hand type
    - Check for Hand Types from best to worst to avoid needing to deal with determining whether a hand has a a higher type, and can stop searching since best hand found would be the correct type
- Compare hands of the same type to determine higher hand
    - Can sort cards for each hand such that they can be compared iteratively (ex: put the pairs at the front so that pair cards are compared, followed by remaining high cards)



Inputs

Because of the number of input files generated for the project, the exact contents of each file is not included in this document. The files used are visible at the following GitHub repository: https://github.com/nattyroe/EN604-cpp-projects/tree/main/PokerHands/input
Input files were generated for test scenarios based on the project assignment sheet. Each test scenario has a matching input file, as well as some additional test files for testing more than two hands.

```
#pragma once

// @file
// @author Nathan Roe
// Card object for standard Ace-high playing cards
//
// Each card has a Value and Suit, and operator
// overloads so that cards can be compared against
// each other.
class Card
{
public:
    // Enum of posible card values
    enum class Value
    {
        Invalid = 0,
        Two = 2,
        Three = 3,
        Four = 4,
        Five = 5,
        Six = 6,
        Seven = 7,
        Eight = 8,
        Nine = 9,
        Ten = 10,
        Jack = 11,
        Queen = 12,
        King = 13,
        Ace = 14
    };

    // Enum of posible card suits
    enum class Suit
    {
        Invalid = ' ',
        Clubs = 'C',
        Diamonds = 'D',
        Hearts = 'H',
        Spades = 'S'
    };

    // Constructs empty, invalid card
    Card();

    // Constructs card using Suit/Value enum parameters
    Card(Value value, Suit suit);

    // Check whether Card has valid suit and value
    //
    // @return True if both suit and value are valid, otherwise false
    bool isValid();

    // Override comparitor operators to compare cards
    bool operator<(const Card &card);
    bool operator>(const Card &card);
    bool operator==(const Card &card);

```

```
58      // Card Suit
59      Suit suit = Suit::Invalid;
60      // Card Value
61      Value value = Value::Invalid;
62  };
```

```cpp
#include "card.h"

// Constructs empty, invalid card
Card::Card() {}

// Constructs card using Suit/Value enum parameters
Card::Card(Value value, Suit suit)
{
    this->value = value;
    this->suit = suit;
} // End Card constructor

// Check whether Card has valid suit and value
bool Card::isValid()
{
    return (!(this->suit == Card::Suit::Invalid) &&
            !(this->value == Card::Value::Invalid));
} // End function isValid

// Less-than operator for comparing Cards
bool Card::operator<(const Card &card)
{
    return this->value < card.value;
} // End < operator override

// Greater-than operator for comparing Cards
bool Card::operator>(const Card &card)
{
    return this->value > card.value;
} // End > operator override

// Equality operator for comparing Cards
bool Card::operator==(const Card &card)
{
    return this->value == card.value;
} // End == operator override
```

```
#pragma once
#include <string>
#include <vector>

class Card;

using namespace std;

// @file
// @author Nathan Roe
// Hand object for holding Cards for Poker
//
// Tracks set of cards using deal and discard functions.
// Also contains a Poker Hand Type.
class Hand
{
public:
    // Enum of posible card values
    enum class HandType
    {
        None,
        HighCard,
        OnePair,
        TwoPair,
        ThreeOfAKind,
        Straight,
        Flush,
        FullHouse,
        FourOfAKind,
        StraightFlush
    };

    // Adds Card to hand
    //
    // @param *card - pointer to card being dealt
    void deal(const Card *card);

    // Removes Card from hand if match exists
    //
    // @param *card - pointer to match of card being discarded
    // @return the Card removed from the hand
    Card discard(const Card *card);

    // Get formatted string of cards in the hand
    //
    // @return std::string of cards giving value and suit
    string printCards();

    // Get string of hand type
    //
    // @return std::string of cards giving value and suit
    string printHandType();

    // Get the current number of cards in the hand
    //
    // @return an int containing the hand size
    int size();
```

```
58
59      // Setter for Poker Hand Type, unvalidated
60      //
61      // @param handType - the HandType of the current hand
62      void setHandType(HandType handType);
63
64      // Getter for Poker Hand Type, unvalidated
65      //
66      // @return the HandType of the current hand
67      HandType getHandType();
68
69      // Getter for a std::vector of current Cards
70      //
71      // @return the current hand of Cards in a vector
72      vector<Card> getCards();
73
74  private:
75      // Vector for storing current hand
76      vector<Card> cards;
77      // Variable for storing HandType
78      HandType handType = HandType::None;
79  };
```

```cpp
#include "hand.h"
#include "card.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Adds Card to hand
void Hand::deal(const Card *card)
{
    this->cards.push_back(*card);
} // End function deal

// Removes Card from hand if match exists
Card Hand::discard(const Card *card)
{
    Card discard;
    // Iterate through hand to find matching card
    for (unsigned int idx = 0; idx < this->cards.size(); ++idx)
    {
        Card currCard = this->cards[idx];
        // If card value and suit match, store card, and remove from hand
        if (currCard.suit == card->suit &&
            currCard.value == card->value)
        {
            discard = this->cards[idx];
            this->cards.erase(this->cards.begin() + idx);
        }
    }
    return discard;
} // End function discard

// Get the current number of cards in the hand
int Hand::size()
{
    return static_cast<int>(this->cards.size());
} // End function size

// Get formatted string of cards in the hand
std::string Hand::printCards()
{
    std::string result = "";
    // Print Value
    for (Card card : this->cards)
    {
        switch (card.value)
        {
        case Card::Value::Two:
            result += "2 of ";
            break;
        case Card::Value::Three:
            result += "3 of ";
            break;
        case Card::Value::Four:
            result += "4 of ";
            break;
```

```
58          case Card::Value::Five:
59              result += "5 of ";
60              break;
61          case Card::Value::Six:
62              result += "6 of ";
63              break;
64          case Card::Value::Seven:
65              result += "7 of ";
66              break;
67          case Card::Value::Eight:
68              result += "8 of ";
69              break;
70          case Card::Value::Nine:
71              result += "9 of ";
72              break;
73          case Card::Value::Ten:
74              result += "10 of ";
75              break;
76          case Card::Value::Jack:
77              result += "J of ";
78              break;
79          case Card::Value::Queen:
80              result += "Q of ";
81              break;
82          case Card::Value::King:
83              result += "K of ";
84              break;
85          case Card::Value::Ace:
86              result += "A of ";
87              break;
88          default:
89              result += "ERROR of ";
90              break;
91          };
92
93          // Print suit
94          switch (card.suit)
95          {
96          case Card::Suit::Clubs:
97              result += "Clubs, ";
98              break;
99          case Card::Suit::Diamonds:
100             result += "Diamonds, ";
101             break;
102         case Card::Suit::Hearts:
103             result += "Hearts, ";
104             break;
105         case Card::Suit::Spades:
106             result += "Spades, ";
107             break;
108         default:
109             result += "ERROR, ";
110             break;
111         };
112     }
113
114     // Remove the extra ", " from end of string
```

```
115     return result.substr(0, result.size() - 2);
116 } // End function printCards
117 
118 // Get string of hand type
119 string Hand::printHandType()
120 {
121     switch (this->handType)
122     {
123     case HandType::HighCard:
124         return "High Card";
125     case HandType::OnePair:
126         return "One Pair";
127     case HandType::TwoPair:
128         return "Two Pair";
129     case HandType::ThreeOfAKind:
130         return "Three of a Kind";
131     case HandType::Straight:
132         return "Straight";
133     case HandType::Flush:
134         return "Flush";
135     case HandType::FullHouse:
136         return "Full House";
137     case HandType::FourOfAKind:
138         return "Four of a Kind";
139     case HandType::StraightFlush:
140         return "Straight Flush";
141     default:
142         return "Unknown; use Scorer::checkHandType";
143     };
144 } // End function printHandType
145 
146 void Hand::setHandType(HandType handType)
147 {
148     this->handType = handType;
149 }
150 
151 // Getter for Poker Hand Type, unvalidated
152 Hand::HandType Hand::getHandType()
153 {
154     return this->handType;
155 } // End function getHandType
156 
157 // Getter for a std::vector of current Cards
158 vector<Card> Hand::getCards()
159 {
160     return this->cards;
161 } // End function getHandType
```

```
#pragma once
#include <vector>

class Card;
class Hand;

using namespace std;

// @file
// @author Nathan Roe
// Class to assess and find winner for set of Poker Hands.
//
// Given a set of hands, will evaluate the Poker Hand Type
// and determine the winner or winners.
class Scorer
{
public:
    // Evaluates a set of hands to determine winner(s)
    //
    // Accepts a vector of Hands and moves winner or
    //    winners to the front of the vector. The return
    //    value indicates the last index of a winner.
    //    One winner returns 0, two-way tie returns 1, etc.
    //    Returns -1 for error
    // @param *hands - pointer to vector of Hands to evaluate and rank
    // @return the index of the final winner in the partially
    //     sorted vector
    int findBestHand(vector<Hand> *hands);

    // Determines and sets the HandType for a given poker hand
    //
    // @param *hand - Hand for which to set HandType
    // @return vector of Cards sorted for comparison based on HandType
    vector<Card> checkHandType(Hand *hand);

private:
    const int HAND_SIZE = 5;

    // Compare two Hands to determine which is better
    //
    // @param hand1 - First Hand for comparing
    // @param hand2 - Second Hand for comparing
    // @return 1 for hand1, 2 for hand2, 0 for tie,
    //    or -1 for error
    int compareHands(Hand *hand1, Hand *hand2);

    // Check hand for presence of Straight Flush
    //
    // If type is present, sort cards for comparison to
    // similar sets of cards
    // @param *cards - pointer to vector of cards to check
    // @return true if hand type is present, false otherwise
    bool hasStraightFlush(vector<Card> *cards);

    // Check hand for presence of Four of a Kind
    //
    // If type is present, sort cards for comparison
```

```
58      // @param *cards - pointer to vector of cards to check
59      // @return true if hand type is present, false otherwise
60      bool hasFourOfAKind(vector<Card> *cards);
61
62      // Check hand for presence of Full House
63      //
64      // If type is present, sort cards for comparison
65      // @param *cards - pointer to vector of cards to check
66      // @return true if hand type is present, false otherwise
67      bool hasFullHouse(vector<Card> *cards);
68
69      // Check hand for presence of Flush
70      //
71      // If type is present, sort cards for comparison
72      // @param *cards - pointer to vector of cards to check
73      // @return true if hand type is present, false otherwise
74      bool hasFlush(vector<Card> *cards);
75
76      // Check hand for presence of Straight
77      //
78      // If type is present, sort cards for comparison
79      // @param *cards - pointer to vector of cards to check
80      // @return true if hand type is present, false otherwise
81      bool hasStraight(vector<Card> *cards);
82
83      // Check hand for presence of Three of a Kind
84      //
85      // If type is present, sort cards for comparison
86      // @param *cards - pointer to vector of cards to check
87      // @return true if hand type is present, false otherwise
88      bool hasThreeOfAKind(vector<Card> *cards);
89
90      // Check hand for presence of Two Pair
91      //
92      // If type is present, sort cards for comparison
93      // @param *cards - pointer to vector of cards to check
94      // @return true if hand type is present, false otherwise
95      bool hasTwoPair(vector<Card> *cards);
96
97      // Check hand for presence of One Pair
98      //
99      // If type is present, sort cards for comparison
100     // @param *cards - pointer to vector of cards to check
101     // @return true if hand type is present, false otherwise
102     bool hasOnePair(vector<Card> *cards);
103
104     // Check hand for presence of High Card
105     //
106     // If type is present, sort cards for comparison
107     // @param *cards - pointer to vector of cards to check
108     // @return true if hand type is present, false otherwise
109     bool hasHighCard(vector<Card> *cards);
110 };
```

```cpp
#include "scorer.h"
#include "hand.h"
#include "card.h"
#include <vector>
#include <algorithm>

using namespace std;

// Evaluates a set of hands to determine winner(s)
int Scorer::findBestHand(vector<Hand> *hands)
{
    // Return error if not enough hands to score
    if (hands->size() < 2)
    {
        return -1;
    }

    int numTied = 0;
    // Begin with first hand as current best
    Hand *bestHand = &hands->at(0);
    // Compare remaining hands to the current best hand
    for (int idx = 1; idx < hands->size(); ++idx)
    {
        int result = compareHands(bestHand, &hands->at(idx));
        // If hands tie, move new hand to front section of vector
        if (result == 0)
        {
            ++numTied;
            Hand hand = hands->at(idx);
            hands->erase(hands->begin() + idx);
            hands->insert(hands->begin() + numTied, hand);
        }
        // If current best hand is better, leave vector unchanged
        else if (result == 1)
        {
            continue;
        }
        // If new hand is better, move to front and reset number of ties
        else if (result == 2)
        {
            numTied = 0;
            Hand hand = hands->at(idx);
            hands->erase(hands->begin() + idx);
            hands->insert(hands->begin(), hand);
        }
        // Return error for unexpected value
        else
        {
            return -1;
        }
    }
    return numTied;
} // End function findBestHand

// Compare two Hands to determine which is better
int Scorer::compareHands(Hand *hand1, Hand *hand2)
{
```

```
58      // If hands are not sets of 5 cards, return error
59      if (!(hand1->size() == HAND_SIZE) ||
60          !(hand2->size() == HAND_SIZE))
61      {
62          return -1;
63      }
64
65      // Check type of each hand, and prep cards for comparison
66      vector<Card> cards1 = checkHandType(hand1);
67      vector<Card> cards2 = checkHandType(hand2);
68
69      // Verify that each hand is a valid Poker Hand Type
70      if (hand1->getHandType() == Hand::HandType::None ||
71          hand2->getHandType() == Hand::HandType::None)
72      {
73          return -1;
74      }
75
76      int result = -1;
77
78      // Check to see if one Hand is a higher rank than the other
79      if (hand1->getHandType() > hand2->getHandType())
80      {
81          result = 1;
82      }
83      else if (hand1->getHandType() < hand2->getHandType())
84      {
85          result = 2;
86      }
87      // If hands are of same rank, compare cards to find best hand
88      else
89      {
90          // Since hands are pre-sorted, iterate through cards until
91          // a higher card is found
92          for (int idx = 0; idx < HAND_SIZE; ++idx)
93          {
94              if (cards1.at(idx) > cards2.at(idx))
95              {
96                  result = 1;
97                  break;
98              }
99              else if (cards1.at(idx) < cards2.at(idx))
100             {
101                 result = 2;
102                 break;
103             }
104             // If all cards are of same value, return tie
105             else
106             {
107                 result = 0;
108             }
109         }
110     }
111     return result;
112 } // End function compareHands
113
114 // Determines and sets the HandType for a given poker hand
```

```cpp
vector<Card> Scorer::checkHandType(Hand *hand)
{
    vector<Card> cards = hand->getCards();
    // Verify that all cards in the hand are valid
    for (Card card : cards)
    {
        if (!card.isValid())
        {
            return cards;
        }
    }

    // Test hand against Poker Hand Types from Best to Worst
    if (hasStraightFlush(&cards))
    {
        hand->setHandType(Hand::HandType::StraightFlush);
    }
    else if (hasFourOfAKind(&cards))
    {
        hand->setHandType(Hand::HandType::FourOfAKind);
    }
    else if (hasFullHouse(&cards))
    {
        hand->setHandType(Hand::HandType::FullHouse);
    }
    else if (hasFlush(&cards))
    {
        hand->setHandType(Hand::HandType::Flush);
    }
    else if (hasStraight(&cards))
    {
        hand->setHandType(Hand::HandType::Straight);
    }
    else if (hasThreeOfAKind(&cards))
    {
        hand->setHandType(Hand::HandType::ThreeOfAKind);
    }
    else if (hasTwoPair(&cards))
    {
        hand->setHandType(Hand::HandType::TwoPair);
    }
    else if (hasOnePair(&cards))
    {
        hand->setHandType(Hand::HandType::OnePair);
    }
    else if (hasHighCard(&cards))
    {
        hand->setHandType(Hand::HandType::HighCard);
    }
    // If no valid hand type is found, assign None
    else
    {
        hand->setHandType(Hand::HandType::None);
    }

    // Return vector of cards sorted for comparison based on HandType
    return cards;
```

```
172 } // End function checkHandType
173
174 // Check hand for presence of Straight Flush
175 bool Scorer::hasStraightFlush(vector<Card> *cards)
176 {
177     // Return false if vector is not 5 cards
178     if (!(cards->size() == HAND_SIZE))
179     {
180         return false;
181     }
182
183     // Sort cards from high to low
184     sort(cards->begin(), cards->end());
185     reverse(cards->begin(), cards->end());
186
187     // Special case for Ace-Low
188     if (cards->at(0).value == Card::Value::Ace &&
189         cards->at(1).value == Card::Value::Five)
190     {
191         // Set starting card value/suit to the 5
192         int startVal = static_cast<typename std::underlying_type<Card::Value>::type>
(cards->at(1).value);
193         Card::Suit startSuit = cards->at(0).suit;
194         // Iterate through remaining cards, and exit if not in
195         // descending order and matching suit
196         for (int idx = 2; idx < cards->size(); ++idx)
197         {
198             int cardVal = static_cast<typename
std::underlying_type<Card::Value>::type>(cards->at(idx).value);
199             Card::Suit cardSuit = cards->at(idx).suit;
200             if (!(cards->at(idx).isValid()) ||
201                 !(cardSuit == startSuit) ||
202                 !(cardVal == startVal-- - 1))
203             {
204                 return false;
205             }
206         }
207         // Move ace to end
208         Card ace = cards->front();
209         cards->erase(cards->begin());
210         cards->push_back(ace);
211     }
212     // Check Cases with no Aces
213     else
214     {
215         // Set starting card value/suit
216         int startVal = static_cast<typename std::underlying_type<Card::Value>::type>
(cards->at(0).value);
217         Card::Suit startSuit = cards->at(0).suit;
218         // Iterate through remaining cards, and exit if not in
219         // descending order and matching suit
220         for (int idx = 1; idx < cards->size(); ++idx)
221         {
222             int cardVal = static_cast<typename
std::underlying_type<Card::Value>::type>(cards->at(idx).value);
223             Card::Suit cardSuit = cards->at(idx).suit;
224             if (!(cards->at(idx).isValid()) ||
225                 !(cardSuit == startSuit) ||
```

```
226                 !(cardVal == startVal-- - 1))
227             {
228                 return false;
229             }
230         }
231     }
232     return true;
233 } // End function hasStraightFlush
234 
235 // Check hand for presence of Four of a Kind
236 bool Scorer::hasFourOfAKind(vector<Card> *cards)
237 {
238     // Return false if vector is not 5 cards
239     if (!(cards->size() == HAND_SIZE))
240     {
241         return false;
242     }
243 
244     // Sort cards from hight to low
245     sort(cards->begin(), cards->end());
246     reverse(cards->begin(), cards->end());
247 
248     int FOAK = 4;
249     int foundFOAK = false;
250     Card::Value startVal = Card::Value::Invalid;
251     // Test cards in sets (1-4) and (2-5) to see if all values match
252     for (int idx = 0; idx <= cards->size() - FOAK; ++idx)
253     {
254         foundFOAK = true;
255         startVal = cards->at(idx).value;
256         // Iterate over 3 following cards and check for match
257         for (int innerIdx = idx + 1; innerIdx < idx + FOAK; ++innerIdx)
258         {
259             Card::Value cardVal = cards->at(innerIdx).value;
260             // If different value is found, no FOAK
261             if (!(cards->at(innerIdx).isValid()) ||
262                 !(startVal == cardVal))
263             {
264                 foundFOAK = false;
265                 break;
266             }
267         }
268         // If first four cards are FOAK, break loop
269         if (foundFOAK)
270         {
271             break;
272         }
273     }
274 
275     // If cards have FOAK, sort matching cards to the front
276     if (foundFOAK)
277     {
278         vector<Card> unusedCards;
279         for (int idx = 0; idx < cards->size(); ++idx)
280         {
281             if (!(cards->at(idx).value == startVal))
282             {
```

```
283                 unusedCards.push_back(cards->at(idx));
284                 cards->erase(cards->begin() + idx);
285             }
286         }
287         // Sort non-FOAK cards low to high
288         sort(unusedCards.begin(), unusedCards.end());
289         // Put unused cards at the end of card set from
290         // hight to low
291         for (int idx = 0; idx < unusedCards.size(); ++idx)
292         {
293             cards->push_back(unusedCards.back());
294             unusedCards.pop_back();
295         }
296     }
297 
298     return foundFOAK;
299 } // End function hasFourOfAKind
300 
301 // Check hand for presence of Full House
302 bool Scorer::hasFullHouse(vector<Card> *cards)
303 {
304     // Return false if vector is not 5 cards
305     if (!(cards->size() == HAND_SIZE))
306     {
307         return false;
308     }
309 
310     sort(cards->begin(), cards->end());
311     reverse(cards->begin(), cards->end());
312 
313     bool foundFF = false;
314     Card::Value pair = Card::Value::Invalid;
315     Card::Value triple = Card::Value::Invalid;
316     // Check wether first two cards match
317     if (cards->at(0).value == cards->at(1).value)
318     {
319         // If first two cards match, check whether
320         // first three cards make a set of 3
321         if (cards->at(1).value == cards->at(2).value)
322         {
323             triple = cards->at(0).value;
324         }
325         // Otherwise, set pair value as value of first card
326         else
327         {
328             pair = cards->at(0).value;
329         }
330 
331         if (!(pair == Card::Value::Invalid))
332         {
333             // If first two cards are a pair, check remaining
334             // cards to see if they make a set of 3
335             if (cards->at(2).value == cards->at(3).value &&
336                 cards->at(3).value == cards->at(4).value)
337             {
338                 foundFF = true;
339                 triple = cards->at(2).value;
```

```
340                 // Move set of 3 to the front
341                 cards->push_back(cards->front());
342                 cards->erase(cards->begin());
343                 cards->push_back(cards->front());
344                 cards->erase(cards->begin());
345             }
346         }
347         else if (!(triple == Card::Value::Invalid))
348         {
349             // If first three cards make a set of 3, check
350             // remaining cards to see if they are a pair
351             if (cards->at(3).value == cards->at(4).value)
352             {
353                 foundFF = true;
354                 pair = cards->at(3).value;
355             }
356         }
357     }
358     return foundFF;
359 } // End function hasFullHouse
360 
361 // Check hand for presence of Flush
362 bool Scorer::hasFlush(vector<Card> *cards)
363 {
364     // Return false if vector is not 5 cards
365     if (!(cards->size() == HAND_SIZE))
366     {
367         return false;
368     }
369 
370     // Sort cards from high to low
371     sort(cards->begin(), cards->end());
372     reverse(cards->begin(), cards->end());
373 
374     Card::Suit startSuit = cards->at(0).suit;
375     // Iterate through cards to check whether suits match
376     for (int idx = 1; idx < cards->size(); ++idx)
377     {
378         Card::Suit cardSuit = cards->at(idx).suit;
379         // If different suit is found, return false
380         if (!(cards->at(idx).isValid()) ||
381             !(cardSuit == startSuit))
382         {
383             return false;
384         }
385     }
386 
387     return true;
388 } // End function hasFlush
389 
390 // Check hand for presence of Straight
391 bool Scorer::hasStraight(vector<Card> *cards)
392 {
393     // Return false if vector is not 5 cards
394     if (!(cards->size() == HAND_SIZE))
395     {
396         return false;
```

```
397     }
398
399     // Sort cards high to low
400     sort(cards->begin(), cards->end());
401     reverse(cards->begin(), cards->end());
402
403     // Special case for Ace-Low
404     if (cards->at(0).value == Card::Value::Ace &&
405         cards->at(1).value == Card::Value::Five)
406     {
407         // Set highest card as the 5
408         int startVal = static_cast<typename std::underlying_type<Card::Value>::type>
(cards->at(1).value);
409         for (int idx = 2; idx < cards->size(); ++idx)
410         {
411             int cardVal = static_cast<typename
std::underlying_type<Card::Value>::type>(cards->at(idx).value);
412             // Iterate through remaining cards to see if they are consecutive
413             if (!(cards->at(idx).isValid()) ||
414                 !(cardVal == startVal-- - 1))
415             {
416                 return false;
417             }
418         }
419         // Move Ace to the back
420         Card ace = cards->front();
421         cards->erase(cards->begin());
422         cards->push_back(ace);
423     }
424     // Check Cases with no Aces
425     else
426     {
427         // Set starting card value
428         int startVal = static_cast<typename std::underlying_type<Card::Value>::type>
(cards->at(0).value);
429         for (int idx = 1; idx < cards->size(); ++idx)
430         {
431             int cardVal = static_cast<typename
std::underlying_type<Card::Value>::type>(cards->at(idx).value);
432             // Iterate through remaining cards to see if they are consecutive
433             if (!(cards->at(idx).isValid()) ||
434                 !(cardVal == startVal-- - 1))
435             {
436                 return false;
437             }
438         }
439     }
440     return true;
441 } // End function hasStraight
442
443 // Check hand for presence of Three of a Kind
444 bool Scorer::hasThreeOfAKind(vector<Card> *cards)
445 {
446     // Return false if vector is not 5 cards
447     if (!(cards->size() == HAND_SIZE))
448     {
449         return false;
450     }
```

```
451
452     // Sort cards high to low
453     sort(cards->begin(), cards->end());
454     reverse(cards->begin(), cards->end());
455
456     int TOAK = 3;
457     int foundTOAK = false;
458     Card::Value startVal = Card::Value::Invalid;
459     // Check (1-3), (2-4), (3-5) for three of a kind
460     for (int idx = 0; idx <= cards->size() - TOAK; ++idx)
461     {
462         foundTOAK = true;
463         startVal = cards->at(idx).value;
464         // Compare three consecutive cards for matching values
465         for (int innerIdx = idx + 1; innerIdx < idx + TOAK; ++innerIdx)
466         {
467             Card::Value cardVal = cards->at(innerIdx).value;
468             // If different value is found, move to next set of 3
469             if (!(cards->at(innerIdx).isValid()) ||
470                 !(startVal == cardVal))
471             {
472                 foundTOAK = false;
473                 break;
474             }
475         }
476         // End search if three of a kind is found
477         if (foundTOAK)
478         {
479             break;
480         }
481     }
482
483     // If three of a kind is found, sort cards for comparison
484     if (foundTOAK)
485     {
486         vector<Card> unusedCards;
487         int cardsMoved = 0;
488         // Remove cards that are not part of TOAK
489         for (int idx = 0; idx < HAND_SIZE; ++idx)
490         {
491             int cardIdx = idx - cardsMoved;
492             if (!(cards->at(cardIdx).value == startVal))
493             {
494                 unusedCards.push_back(cards->at(cardIdx));
495                 cards->erase(cards->begin() + cardIdx);
496                 ++cardsMoved;
497             }
498         }
499         // Place non-TOAK cards from high-low and end of cards
500         sort(unusedCards.begin(), unusedCards.end());
501         size_t startSize = unusedCards.size();
502         for (int idx = 0; idx < startSize; ++idx)
503         {
504             cards->push_back(unusedCards.back());
505             unusedCards.pop_back();
506         }
507     }
```

```
508
509     return foundTOAK;
510 } // End function hasThreeOfAKind
511
512 // Check hand for presence of Two Pair
513 bool Scorer::hasTwoPair(vector<Card> *cards)
514 {
515     // Return false if vector is not 5 cards
516     if (!(cards->size() == HAND_SIZE))
517     {
518         return false;
519     }
520
521     // Sort cards from high to low
522     sort(cards->begin(), cards->end());
523     reverse(cards->begin(), cards->end());
524
525     bool foundTP = false;
526     vector<Card> pair1;
527     vector<Card> pair2;
528
529     // Search cards for a pair
530     for (int idx = 1; idx < cards->size(); ++idx)
531     {
532         int prevIdx = idx - 1;
533         Card *card = &cards->at(idx);
534         Card *prevCard = &cards->at(prevIdx);
535         // If pair is found, store cards
536         if (card->value == prevCard->value)
537         {
538             foundTP = true;
539             pair1.push_back(*prevCard);
540             pair1.push_back(*card);
541             cards->erase(cards->begin() + prevIdx);
542             cards->erase(cards->begin() + prevIdx);
543             break;
544         }
545     }
546
547     // If pair is found, search remaining cards for second pair
548     if (foundTP)
549     {
550         foundTP = false;
551         for (int idx = 1; idx < cards->size(); ++idx)
552         {
553             int prevIdx = idx - 1;
554             Card *card = &cards->at(idx);
555             Card *prevCard = &cards->at(prevIdx);
556             // If pair is found, store cards
557             if (card->value == prevCard->value)
558             {
559                 foundTP = true;
560                 pair2.push_back(*prevCard);
561                 pair2.push_back(*card);
562                 cards->erase(cards->begin() + prevIdx);
563                 cards->erase(cards->begin() + prevIdx);
564             }
```

```
565         }
566     }
567 
568     // If two-pair found, sort cards highPair-lowPair-spareCard
569     if (foundTP)
570     {
571         // Place pairs in set highest to lowest
572         if (pair1.front().value > pair2.front().value)
573         {
574             cards->push_back(pair1.front());
575             cards->push_back(pair1.back());
576             cards->push_back(pair2.front());
577             cards->push_back(pair2.back());
578         }
579         else
580         {
581             cards->push_back(pair2.front());
582             cards->push_back(pair2.back());
583             cards->push_back(pair1.front());
584             cards->push_back(pair1.back());
585         }
586         // Move spare card to the end
587         cards->push_back(cards->front());
588         cards->erase(cards->begin());
589     }
590     // Place unused pair back into card set
591     else if (pair1.size() > 0)
592     {
593         cards->push_back(pair1.front());
594         cards->push_back(pair1.back());
595     }
596 
597     return foundTP;
598 } // End function foundTwoPair
599 
600 // Check hand for presence of One Pair
601 bool Scorer::hasOnePair(vector<Card> *cards)
602 {
603     // Return false if vector is not 5 cards
604     if (!(cards->size() == HAND_SIZE))
605     {
606         return false;
607     }
608 
609     sort(cards->begin(), cards->end());
610     reverse(cards->begin(), cards->end());
611 
612     bool foundPair = false;
613     vector<Card> pair;
614 
615     // Search cards for pair of equal value
616     for (int idx = 1; idx < cards->size(); ++idx)
617     {
618         int prevIdx = idx - 1;
619         Card *card = &cards->at(idx);
620         Card *prevCard = &cards->at(prevIdx);
621         // If pair is found, store pair
```

```
622         if (card->value == prevCard->value)
623         {
624             foundPair = true;
625             pair.push_back(*prevCard);
626             pair.push_back(*card);
627             cards->erase(cards->begin() + prevIdx);
628             cards->erase(cards->begin() + prevIdx);
629             break;
630         }
631     }
632 
633     // If pair is found, place pair at front of cards
634     if (foundPair)
635     {
636         cards->insert(cards->begin(), pair.front());
637         cards->insert(cards->begin(), pair.back());
638     }
639 
640     return foundPair;
641 } // End function hasOnePair
642 
643 // Check hand for presence of High Card
644 bool Scorer::hasHighCard(vector<Card> *cards)
645 {
646     // Return false if vector is not 5 cards
647     if (!(cards->size() == HAND_SIZE))
648     {
649         return false;
650     }
651 
652     // Sort cards from high to low
653     sort(cards->begin(), cards->end());
654     reverse(cards->begin(), cards->end());
655 
656     return true;
657 } // End function hasHighCard
```

```cpp
#include "card.h"
#include "hand.h"
#include "scorer.h"
#include <iostream>
#include <vector>
#include <libconfig.h++>

using namespace std;
using namespace libconfig;
// @file
// @author Nathan Roe
// Compares Poker Hands provided in file to find winners
//
// Given a set of hands, will evaluate the Poker Hand Type
// and determine the winner or winners.

// Creates Card object using integer value and string suit
Card makeCard(int value, string suit)
{
    Card::Value cardValue = Card::Value::Invalid;
    Card::Suit cardSuit = Card::Suit::Invalid;
    // Set card value
    try
    {
        cardValue = static_cast<Card::Value>(value);
    }
    catch (const exception &e)
    {
        cout << e.what() << endl;
        cout << "Invalid Card Value: " << value << endl;
        cardValue = Card::Value::Invalid;
    }

    // Set card suit
    if (suit.size() > 0)
    {
        char cardChar = suit[0];
        switch (cardChar)
        {
        case 'C':
            cardSuit = Card::Suit::Clubs;
            break;
        case 'D':
            cardSuit = Card::Suit::Diamonds;
            break;
        case 'H':
            cardSuit = Card::Suit::Hearts;
            break;
        case 'S':
            cardSuit = Card::Suit::Spades;
            break;
        default:
            cardSuit = Card::Suit::Invalid;
            break;
        }
    }

```

```
58      return Card(cardValue, cardSuit);
59  } // End function makeCard
60
61  // Creates Card object using string value and suit
62  Card makeCard(string value, string suit)
63  {
64      Card::Value cardValue = Card::Value::Invalid;
65      Card::Suit cardSuit = Card::Suit::Invalid;
66
67      // Set value for Ten through Ace
68      if (value == "Ten" || value == "T")
69      {
70          cardValue = Card::Value::Ten;
71      }
72      else if (value == "Jack" || value == "J")
73      {
74          cardValue = Card::Value::Jack;
75      }
76      else if (value == "Queen" || value == "Q")
77      {
78          cardValue = Card::Value::Queen;
79      }
80      else if (value == "King" || value == "K")
81      {
82          cardValue = Card::Value::King;
83      }
84      else if (value == "Ace" || value == "A")
85      {
86          cardValue = Card::Value::Ace;
87      }
88      // If card is not 10-Ace, try to set based on numerical value
89      else
90      {
91          try
92          {
93              cardValue = static_cast<Card::Value>(stoi(value));
94          }
95          catch (const invalid_argument &e)
96          {
97              cout << e.what() << ": Invalid Card Value: " << value << endl;
98              cardValue = Card::Value::Invalid;
99          }
100         catch (const exception &e)
101         {
102             cout << e.what() << ": Invalid Card Value: " << value << endl;
103             cardValue = Card::Value::Invalid;
104         }
105     }
106
107     // Set suit value
108     if (suit.size() > 0)
109     {
110         char cardChar = suit[0];
111         switch (cardChar)
112         {
113         case 'C':
114             cardSuit = Card::Suit::Clubs;
```

```
115             break;
116         case 'D':
117             cardSuit = Card::Suit::Diamonds;
118             break;
119         case 'H':
120             cardSuit = Card::Suit::Hearts;
121             break;
122         case 'S':
123             cardSuit = Card::Suit::Spades;
124             break;
125         default:
126             cardSuit = Card::Suit::Invalid;
127             break;
128         }
129     }
130 
131     return Card(cardValue, cardSuit);
132 } // End function makeCard
133 
134 // Format and print test results for input files
135 void printResults(int lastWinIdx, vector<Hand> *players)
136 {
137     // Print out error message if error value returned
138     if (lastWinIdx == -1)
139     {
140         cout << "ERROR IN SCENARIO" << endl;
141         return;
142     }
143 
144     // Print out winning hands
145     cout << "(" << players->at(0).printCards() << ")";
146     for (int idx = 1; idx <= lastWinIdx; ++idx)
147     {
148         cout << ", "
149              << "(" << players->at(idx).printCards() << ")";
150     }
151     // Format string based on whether ties are present
152     if (lastWinIdx > 0)
153     {
154         cout << " Tie for the Win.";
155     }
156     else
157     {
158         cout << " Wins.";
159     }
160 
161     // Print out losing hands
162     bool multiLoser = false;
163     for (int idx = lastWinIdx + 1; idx < players->size(); ++idx)
164     {
165         cout << " "
166              << "(" << players->at(idx).printCards() << ")";
167         if (idx < players->size() - 1)
168         {
169             multiLoser = true;
170             cout << " and";
171         }
```

```
172     }
173     // Format string based on presence of multiple losing hands
174     if (multiLoser)
175     {
176         cout << " Lose.";
177     }
178     else if (lastWinIdx < players->size() - 1)
179     {
180         cout << " Loses.";
181     }
182     cout << endl;
183 } // End function printResults
184 
185 // Main function; runs Poker Hand Scorer for each input file provided
186 int main(int argc, char **argv)
187 {
188     Scorer *scorer = new Scorer();
189 
190     // Iterate through input args, and read as paths
191     for (int idx = 1; idx < argc; ++idx)
192     {
193         Config *cfg = new Config();
194         char *path = argv[idx];
195 
196         // Attempt to parse Config files
197         try
198         {
199             cout << path << endl;
200             cfg->readFile(path);
201         }
202         // Catch file path errors
203         catch (const FileIOException &fioex)
204         {
205             cerr << fioex.what() << " while reading file " << path << endl;
206             return (EXIT_FAILURE);
207         }
208         // Catch config file format errors
209         catch (const ParseException &pex)
210         {
211             cerr << "Parse error at " << pex.getFile() << ":" << pex.getLine()
212                  << " - " << pex.getError() << endl;
213             return (EXIT_FAILURE);
214         }
215 
216         vector<Hand> *players = new vector<Hand>;
217 
218         const Setting &root = cfg->getRoot();
219         // Find "hands" data set in config file
220         try
221         {
222             const Setting &hands = root["hands"];
223             int numHands = hands.getLength();
224 
225             // Create a Hand object for each hand in Config
226             for (int handIdx = 0; handIdx < numHands; ++handIdx)
227             {
228 
```

```
229                 Hand *hand = new Hand();
230 
231                 // Read Cards list in config
232                 const Setting &cards = hands[handIdx];
233                 int numCards = cards.getLength();
234                 for (int cardIdx = 0; cardIdx < numCards; ++cardIdx)
235                 {
236                     const Setting &card = cards[cardIdx];
237 
238                     string suit;
239                     bool suitFound = false;
240                     bool valueFound = false;
241 
242                     // Find required suit and value
243                     suitFound = card.lookupValue("suit", suit);
244                     if (suitFound)
245                     {
246                         int value;
247                         string valueString;
248                         // Search for card values formatted as Integers
249                         valueFound = card.lookupValue("value", value);
250                         if (valueFound)
251                         {
252                             Card card = makeCard(value, suit);
253                             // Add valid card to hand
254                             if (card.isValid())
255                             {
256                                 hand->deal(&card);
257                             }
258                         }
259                         else
260                         {
261                             // Search for card values formatted as Strings
262                             valueFound = card.lookupValue("value", valueString);
263                             if (valueFound)
264                             {
265                                 Card card = makeCard(valueString, suit);
266                                 // Add valid card to hand
267                                 if (card.isValid())
268                                 {
269                                     hand->deal(&card);
270                                 }
271                             }
272                         }
273                     }
274                     // Print error for bad hand
275                     if (!(valueFound && suitFound))
276                     {
277                         cout << "Could not find 'suit' and/or 'value' in Hand " <<
    handIdx + 1 << " Card " << cardIdx + 1 << endl;
278                     }
279                 }
280                 players->push_back(*hand);
281             }
282 
283             // Run scorer if multiple hands present
284             if (players->size() > 1)
```

```
285             {
286                 int winnerIdx = scorer->findBestHand(players);
287                 for (Hand hand : *players)
288                 {
289                     cout << "(" << hand.printCards() << "): " << hand.printHandType()
    << endl;
290                 }
291                 printResults(winnerIdx, players);
292             }
293         }
294         // Print error message for improperly formatted file
295         catch (const SettingNotFoundException &nfex)
296         {
297             cout << nfex.what() << ": Could not find 'hands' in " << path << endl;
298         }
299     }
300     return 0;
301 } // End function main
```

```python
""" Runs Unit Test for PokerHands c++ executable

Compares output to Expected result stored in input files. The expected results are
based on c++ output formatting, and are verified to be the correct hand ranking;
tests should be used to verify that code updates do not result in errors in ranking.
"""
import os
import glob
import subprocess
import libconf


# Relative path to executable from the folder contatining
# this test manager
EXECUTABLE_PATH = '../build/x64/PokerHands.exe'


def main():
    # Setup working directory
    start_path = os.getcwd()
    os.chdir(os.path.dirname(os.path.abspath(__file__)))

    exe_path = os.path.join(os.getcwd(), EXECUTABLE_PATH)
    input_path = os.path.join(os.getcwd(), '../Input/')
    files = os.listdir(input_path)
    for file_name in files:
        test_path = os.path.join(input_path, file_name)
        try:
            # Run Executable and Print Results
            output_bits = subprocess.check_output([exe_path, test_path])
            output_strings = [line.rstrip() for line in output_bits.decode(
                'UTF-8').split('\n') if line]
            print('Output:')
            for line in output_strings:
                print('    ', line)

            # Compare Executable Output with expected results
            with open(test_path) as f:
                config = libconf.load(f)
                try:
                    expected_output = config['expectedResult']
                    if expected_output == output_strings[-1]:
                        print('Test Result: PASS\n')
                    else:
                        print('Test Result: FAIL\n')
                except KeyError:
                    print('No Expected Result Found.\n')
                    continue
        except subprocess.CalledProcessError:
            print(f'Could not run file {test_path}\n')

    # Return working dir to starting location
    os.chdir(start_path)


if __name__ == "__main__":
    main()
```

```
C:\Users\nater\source\repos\PokerHands>python test\PokerHandTest.py > doc\output.txt

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush1.cfg
     (K of Diamonds, J of Diamonds, 9 of Diamonds, 6 of Diamonds, 4 of Diamonds): Flush
     (Q of Clubs, J of Clubs, 7 of Clubs, 6 of Clubs, 5 of Clubs): Flush
     (K of Diamonds, J of Diamonds, 9 of Diamonds, 6 of Diamonds, 4 of Diamonds) Wins. (Q of Clubs, J of Clubs, 7 of Clubs, 6 of Clubs, 5 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush2.cfg
     (Q of Clubs, J of Clubs, 7 of Clubs, 6 of Clubs, 5 of Clubs): Flush
     (J of Hearts, 10 of Hearts, 9 of Hearts, 4 of Hearts, 2 of Hearts): Flush
     (Q of Clubs, J of Clubs, 7 of Clubs, 6 of Clubs, 5 of Clubs) Wins. (J of Hearts, 10 of Hearts, 9 of Hearts, 4 of Hearts, 2 of Hearts) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush3.cfg
     (J of Hearts, 10 of Hearts, 9 of Hearts, 4 of Hearts, 2 of Hearts): Flush
     (J of Spades, 10 of Spades, 8 of Spades, 6 of Spades, 3 of Spades): Flush
     (J of Hearts, 10 of Hearts, 9 of Hearts, 4 of Hearts, 2 of Hearts) Wins. (J of Spades, 10 of Spades, 8 of Spades, 6 of Spades, 3 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush4.cfg
     (J of Spades, 10 of Spades, 8 of Spades, 6 of Spades, 3 of Spades): Flush
     (J of Hearts, 10 of Hearts, 8 of Hearts, 4 of Hearts, 3 of Hearts): Flush
     (J of Spades, 10 of Spades, 8 of Spades, 6 of Spades, 3 of Spades) Wins. (J of Hearts, 10 of Hearts, 8 of Hearts, 4 of Hearts, 3 of Hearts) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush5.cfg
     (J of Hearts, 10 of Hearts, 8 of Hearts, 4 of Hearts, 3 of Hearts): Flush
     (J of Clubs, 10 of Clubs, 8 of Clubs, 4 of Clubs, 2 of Clubs): Flush
     (J of Hearts, 10 of Hearts, 8 of Hearts, 4 of Hearts, 3 of Hearts) Wins. (J of Clubs, 10 of Clubs, 8 of Clubs, 4 of Clubs, 2 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Flush6.cfg
     (10 of Diamonds, 8 of Diamonds, 7 of Diamonds, 6 of Diamonds, 5 of Diamonds): Flush
     (10 of Spades, 8 of Spades, 7 of Spades, 6 of Spades, 5 of Spades): Flush
     (10 of Diamonds, 8 of Diamonds, 7 of Diamonds, 6 of Diamonds, 5 of Diamonds), (10 of Spades, 8 of Spades, 7 of Spades, 6 of Spades, 5 of Spades) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FourOfAKind1.cfg
     (K of Spades, K of Hearts, K of Clubs, K of Diamonds, 3 of Hearts): Four of a Kind
     (7 of Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, Q of Hearts): Four of a Kind
     (K of Spades, K of Hearts, K of Clubs, K of Diamonds, 3 of Hearts) Wins. (7 of
```

```
Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, Q of Hearts) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FourOfAKind2.cfg
     (7 of Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, Q of Hearts): Four of a Kind
     (7 of Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, 10 of Spades): Four of a Kind
     (7 of Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, Q of Hearts) Wins. (7 of Hearts, 7 of Diamonds, 7 of Spades, 7 of Clubs, 10 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FourOfAKind3.cfg
     (4 of Clubs, 4 of Spades, 4 of Diamonds, 4 of Hearts, 9 of Clubs): Four of a Kind
     (4 of Clubs, 4 of Spades, 4 of Diamonds, 4 of Hearts, 9 of Diamonds): Four of a Kind
     (4 of Clubs, 4 of Spades, 4 of Diamonds, 4 of Hearts, 9 of Clubs), (4 of Clubs, 4 of Spades, 4 of Diamonds, 4 of Hearts, 9 of Diamonds) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FullHouse1.cfg
     (8 of Spades, 8 of Diamonds, 8 of Hearts, 7 of Diamonds, 7 of Clubs): Full House
     (4 of Diamonds, 4 of Spades, 4 of Clubs, 9 of Diamonds, 9 of Clubs): Full House
     (8 of Spades, 8 of Diamonds, 8 of Hearts, 7 of Diamonds, 7 of Clubs) Wins. (4 of Diamonds, 4 of Spades, 4 of Clubs, 9 of Diamonds, 9 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FullHouse2.cfg
     (4 of Diamonds, 4 of Spades, 4 of Clubs, 9 of Diamonds, 9 of Clubs): Full House
     (4 of Diamonds, 4 of Spades, 4 of Clubs, 5 of Clubs, 5 of Diamonds): Full House
     (4 of Diamonds, 4 of Spades, 4 of Clubs, 9 of Diamonds, 9 of Clubs) Wins. (4 of Diamonds, 4 of Spades, 4 of Clubs, 5 of Clubs, 5 of Diamonds) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/FullHouse3.cfg
     (K of Clubs, K of Spades, K of Diamonds, J of Clubs, J of Spades): Full House
     (K of Clubs, K of Hearts, K of Diamonds, J of Clubs, J of Hearts): Full House
     (K of Clubs, K of Spades, K of Diamonds, J of Clubs, J of Spades), (K of Clubs, K of Hearts, K of Diamonds, J of Clubs, J of Hearts) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard1.cfg
     (K of Spades, 6 of Clubs, 5 of Hearts, 3 of Diamonds, 2 of Clubs): High Card
     (Q of Spades, J of Diamonds, 6 of Clubs, 5 of Hearts, 3 of Clubs): High Card
     (K of Spades, 6 of Clubs, 5 of Hearts, 3 of Diamonds, 2 of Clubs) Wins. (Q of Spades, J of Diamonds, 6 of Clubs, 5 of Hearts, 3 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard2.cfg
     (Q of Spades, J of Diamonds, 6 of Clubs, 5 of Hearts, 3 of Clubs): High Card
     (Q of Spades, 10 of Diamonds, 8 of Clubs, 7 of Diamonds, 4 of Spades): High Card
     (Q of Spades, J of Diamonds, 6 of Clubs, 5 of Hearts, 3 of Clubs) Wins. (Q of
```

```
Spades, 10 of Diamonds, 8 of Clubs, 7 of Diamonds, 4 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard3.cfg
     (Q of Spades, 10 of Diamonds, 8 of Clubs, 7 of Diamonds, 4 of Spades): High Card
     (Q of Hearts, 10 of Hearts, 7 of Clubs, 6 of Hearts, 4 of Spades): High Card
     (Q of Spades, 10 of Diamonds, 8 of Clubs, 7 of Diamonds, 4 of Spades) Wins. (Q of Hearts, 10 of Hearts, 7 of Clubs, 6 of Hearts, 4 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard4.cfg
     (Q of Hearts, 10 of Hearts, 7 of Clubs, 6 of Hearts, 4 of Spades): High Card
     (Q of Clubs, 10 of Clubs, 7 of Diamonds, 5 of Clubs, 4 of Diamonds): High Card
     (Q of Hearts, 10 of Hearts, 7 of Clubs, 6 of Hearts, 4 of Spades) Wins. (Q of Clubs, 10 of Clubs, 7 of Diamonds, 5 of Clubs, 4 of Diamonds) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard5.cfg
     (Q of Clubs, 10 of Clubs, 7 of Diamonds, 5 of Clubs, 4 of Diamonds): High Card
     (Q of Hearts, 10 of Diamonds, 7 of Spades, 5 of Spades, 2 of Hearts): High Card
     (Q of Clubs, 10 of Clubs, 7 of Diamonds, 5 of Clubs, 4 of Diamonds) Wins. (Q of Hearts, 10 of Diamonds, 7 of Spades, 5 of Spades, 2 of Hearts) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/HighCard6.cfg
     (10 of Clubs, 8 of Spades, 7 of Spades, 6 of Hearts, 4 of Diamonds): High Card
     (10 of Diamonds, 8 of Diamonds, 7 of Spades, 6 of Clubs, 4 of Clubs): High Card
     (10 of Clubs, 8 of Spades, 7 of Spades, 6 of Hearts, 4 of Diamonds), (10 of Diamonds, 8 of Diamonds, 7 of Spades, 6 of Clubs, 4 of Clubs) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair1.cfg
     (9 of Clubs, 9 of Diamonds, Q of Spades, J of Hearts, 5 of Hearts): One Pair
     (6 of Diamonds, 6 of Hearts, K of Spades, 7 of Hearts, 4 of Clubs): One Pair
     (9 of Clubs, 9 of Diamonds, Q of Spades, J of Hearts, 5 of Hearts) Wins. (6 of Diamonds, 6 of Hearts, K of Spades, 7 of Hearts, 4 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair2.cfg
     (6 of Diamonds, 6 of Hearts, K of Spades, 7 of Hearts, 4 of Clubs): One Pair
     (6 of Diamonds, 6 of Hearts, Q of Hearts, J of Spades, 2 of Clubs): One Pair
     (6 of Diamonds, 6 of Hearts, K of Spades, 7 of Hearts, 4 of Clubs) Wins. (6 of Diamonds, 6 of Hearts, Q of Hearts, J of Spades, 2 of Clubs) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair3 (2 shuffle).cfg
     (4 of Clubs, 6 of Diamonds, K of Spades, 7 of Hearts, 6 of Hearts): One Pair
     (J of Spades, 6 of Diamonds, 6 of Hearts, 2 of Clubs, Q of Hearts): One Pair
     (4 of Clubs, 6 of Diamonds, K of Spades, 7 of Hearts, 6 of Hearts) Wins. (J of Spades, 6 of Diamonds, 6 of Hearts, 2 of Clubs, Q of Hearts) Loses.
Test Result: PASS

```

```
Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair4.cfg
     (6 of Diamonds, 6 of Hearts, Q of Hearts, J of Spades, 2 of Clubs): One Pair
     (6 of Diamonds, 6 of Hearts, Q of Spades, 8 of Clubs, 7 of Diamonds): One Pair
     (6 of Diamonds, 6 of Hearts, Q of Hearts, J of Spades, 2 of Clubs) Wins. (6 of Diamonds, 6 of Hearts, Q of Spades, 8 of Clubs, 7 of Diamonds) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair5.cfg
     (6 of Diamonds, 6 of Hearts, Q of Spades, 8 of Clubs, 7 of Diamonds): One Pair
     (6 of Diamonds, 6 of Hearts, Q of Diamonds, 8 of Hearts, 3 of Spades): One Pair
     (6 of Diamonds, 6 of Hearts, Q of Spades, 8 of Clubs, 7 of Diamonds) Wins. (6 of Diamonds, 6 of Hearts, Q of Diamonds, 8 of Hearts, 3 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/OnePair6.cfg
     (8 of Spades, 8 of Diamonds, 10 of Hearts, 6 of Clubs, 5 of Spades): One Pair
     (8 of Hearts, 8 of Clubs, 10 of Clubs, 6 of Spades, 5 of Clubs): One Pair
     (8 of Spades, 8 of Diamonds, 10 of Hearts, 6 of Clubs, 5 of Spades), (8 of Hearts, 8 of Clubs, 10 of Clubs, 6 of Spades, 5 of Clubs) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Straight1.cfg
     (J of Hearts, 10 of Hearts, 9 of Clubs, 8 of Spades, 7 of Hearts): Straight
     (10 of Spades, 9 of Spades, 8 of Clubs, 7 of Hearts, 6 of Spades): Straight
     (J of Hearts, 10 of Hearts, 9 of Clubs, 8 of Spades, 7 of Hearts) Wins. (10 of Spades, 9 of Spades, 8 of Clubs, 7 of Hearts, 6 of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Straight2.cfg
     (10 of Spades, 9 of Spades, 8 of Clubs, 7 of Hearts, 6 of Spades): Straight
     (6 of Clubs, 5 of Spades, 4 of Hearts, 3 of Spades, 2 of Diamonds): Straight
     (10 of Spades, 9 of Spades, 8 of Clubs, 7 of Hearts, 6 of Spades) Wins. (6 of Clubs, 5 of Spades, 4 of Hearts, 3 of Spades, 2 of Diamonds) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/Straight3.cfg
     (9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Diamonds, 5 of Diamonds): Straight
     (9 of Spades, 8 of Spades, 7 of Spades, 6 of Hearts, 5 of Hearts): Straight
     (9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Diamonds, 5 of Diamonds), (9 of Spades, 8 of Spades, 7 of Spades, 6 of Hearts, 5 of Hearts) Tie for the Win.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/StraightFlush1.cfg
     (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs): Straight Flush
     (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts): Straight Flush
     (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs) Wins. (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/StraightFlush2.cfg
```

```
    (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts): Straight Flush
    (6 of Spades, 5 of Spades, 4 of Spades, 3 of Spades, 2 of Spades): Straight Flush
    (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts) Wins. (6 of Spades, 5 of Spades, 4 of Spades, 3 of Spades, 2 of Spades) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/StraightFlush3.cfg
    (7 of Diamonds, 6 of Diamonds, 5 of Diamonds, 4 of Diamonds, 3 of Diamonds): Straight Flush
    (7 of Spades, 6 of Spades, 5 of Spades, 4 of Spades, 3 of Spades): Straight Flush
    (7 of Diamonds, 6 of Diamonds, 5 of Diamonds, 4 of Diamonds, 3 of Diamonds), (7 of Spades, 6 of Spades, 5 of Spades, 4 of Spades, 3 of Spades) Tie for the Win.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/ThreeOfAKind1.cfg
    (6 of Hearts, 6 of Diamonds, 6 of Spades, Q of Clubs, 4 of Spades): Three of a Kind
    (3 of Diamonds, 3 of Spades, 3 of Clubs, K of Spades, 2 of Spades): Three of a Kind
    (6 of Hearts, 6 of Diamonds, 6 of Spades, Q of Clubs, 4 of Spades) Wins. (3 of Diamonds, 3 of Spades, 3 of Clubs, K of Spades, 2 of Spades) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/ThreeOfAKind2.cfg
    (3 of Diamonds, 3 of Spades, 3 of Clubs, K of Spades, 2 of Spades): Three of a Kind
    (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Clubs, 7 of Hearts): Three of a Kind
    (3 of Diamonds, 3 of Spades, 3 of Clubs, K of Spades, 2 of Spades) Wins. (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Clubs, 7 of Hearts) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/ThreeOfAKind3.cfg
    (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Clubs, 7 of Hearts): Three of a Kind
    (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Spades, 5 of Diamonds): Three of a Kind
    (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Clubs, 7 of Hearts) Wins. (3 of Diamonds, 3 of Spades, 3 of Clubs, J of Spades, 5 of Diamonds) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/ThreeOfAKind4.cfg
    (9 of Spades, 9 of Hearts, 9 of Diamonds, 10 of Diamonds, 8 of Hearts): Three of a Kind
    (9 of Clubs, 9 of Spades, 9 of Hearts, 10 of Diamonds, 8 of Diamonds): Three of a Kind
    (9 of Spades, 9 of Hearts, 9 of Diamonds, 10 of Diamonds, 8 of Hearts), (9 of Clubs, 9 of Spades, 9 of Hearts, 10 of Diamonds, 8 of Diamonds) Tie for the Win.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/ThreeWayTie.cfg
    (10 of Diamonds, 9 of Diamonds, 8 of Diamonds, 6 of Diamonds, 7 of Diamonds): Straight Flush
    (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs): Straight Flush
```

```
    (10 of Hearts, 9 of Hearts, 8 of Hearts, 6 of Hearts, 7 of Hearts): Straight Flush
    (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts): Straight Flush
    (10 of Diamonds, 9 of Diamonds, 8 of Diamonds, 6 of Diamonds, 7 of Diamonds), (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs), (10 of Hearts, 9 of Hearts, 8 of Hearts, 6 of Hearts, 7 of Hearts) Tie for the Win. (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TripleTest1.cfg
    (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs): Straight Flush
    (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts): Straight Flush
    (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts): Straight Flush
    (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs) Wins. (8 of Hearts, 7 of Hearts, 6 of Hearts, 5 of Hearts, 4 of Hearts) and (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts) Lose.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TripleTest2.cfg
    (10 of Diamonds, 9 of Diamonds, 8 of Diamonds, 6 of Diamonds, 7 of Diamonds): Straight Flush
    (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs): Straight Flush
    (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts): Straight Flush
    (10 of Diamonds, 9 of Diamonds, 8 of Diamonds, 6 of Diamonds, 7 of Diamonds), (10 of Clubs, 9 of Clubs, 8 of Clubs, 7 of Clubs, 6 of Clubs) Tie for the Win. (3 of Hearts, 5 of Hearts, 2 of Hearts, A of Hearts, 4 of Hearts) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TwoPair1.cfg
    (10 of Diamonds, 10 of Spades, 2 of Spades, 2 of Clubs, K of Clubs): Two Pair
    (5 of Clubs, 5 of Spades, 4 of Diamonds, 4 of Hearts, 10 of Hearts): Two Pair
    (10 of Diamonds, 10 of Spades, 2 of Spades, 2 of Clubs, K of Clubs) Wins. (5 of Clubs, 5 of Spades, 4 of Diamonds, 4 of Hearts, 10 of Hearts) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TwoPair2.cfg
    (5 of Clubs, 5 of Spades, 4 of Diamonds, 4 of Hearts, 10 of Hearts): Two Pair
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades): Two Pair
    (5 of Clubs, 5 of Spades, 4 of Diamonds, 4 of Hearts, 10 of Hearts) Wins. (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TwoPair3.cfg
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades): Two Pair
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, J of Spades): Two Pair
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades) Wins. (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, J of Spades) Loses.
Test Result: PASS

Output:
    C:\Users\nater\source\repos\PokerHands\test\../Input/TwoPair3Flipped.cfg
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades): Two Pair
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, J of Spades): Two Pair
    (5 of Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, Q of Spades) Wins. (5 of
```

```
Clubs, 5 of Spades, 3 of Clubs, 3 of Diamonds, J of Spades) Loses.
Test Result: PASS

Output:
     C:\Users\nater\source\repos\PokerHands\test\../Input/TwoPair4.cfg
     (K of Diamonds, K of Spades, 7 of Diamonds, 7 of Hearts, 8 of Hearts): Two Pair
     (K of Clubs, K of Spades, 7 of Clubs, 7 of Hearts, 8 of Clubs): Two Pair
     (K of Diamonds, K of Spades, 7 of Diamonds, 7 of Hearts, 8 of Hearts), (K of
Clubs, K of Spades, 7 of Clubs, 7 of Hearts, 8 of Clubs) Tie for the Win.
Test Result: PASS
```