|  |
| --- |
| war the card game in c++ |
|  |

|  |  |
| --- | --- |
| April 2021 | By Jason R. Jones |

The objective is simple, win all the cards in the deck at all costs.

war the card game in c++

By Jason R. Jones

# Introduction

War the card game has a special place in my heart. I have so many fond memories playing War with my older sister Krista. Although many of the rules when we played were sometimes made up on the fly. We would commonly alter the number of cards that where laid face down in the event of a war. As it turns out this was for the better. Now that I have created War in C++ and used the traditional rules, I see that a game of War can take quite a long time to complete.

One cannot simply ignore the finitude of War; in fact, one may argue that the probability of an extraordinarily long game is the biggest deterrent from playing. This notion ties in perfectly with the C++ data structures class as one can use the idea of mathematical analysis to determine the length of each game of War. I will do this by simply counting the number of hands played per game. I will then store the highest and lowest hand counts for a score board.

The War card game project has been a lengthy process. The project was created over the span of a month. Thankfully due to extensions I was able to implement better modularity, check off more requirements and perfect the project. Because I was staying on top of every due date, I was able to improve upon the project was not only able to improve the code but also write well-defined documentation. Since its creation I spent over 9 hours a week working on it. Some days I invested more time and other days due to other responsibilities little to no time at all.

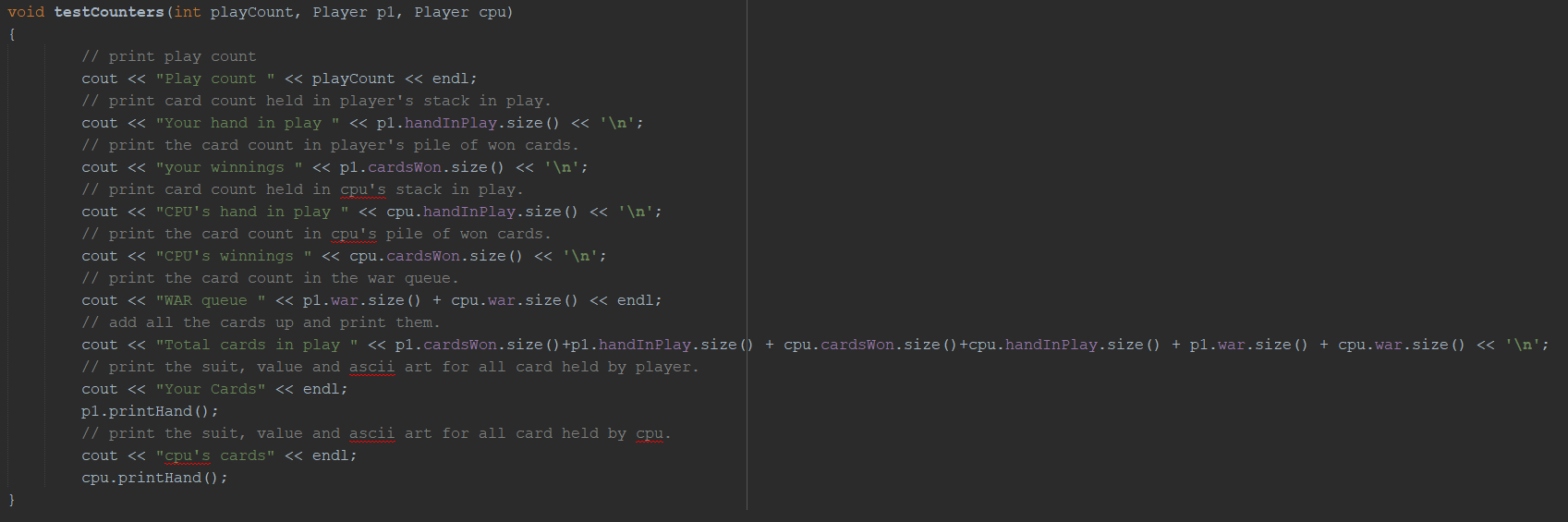
The project can be found on my GitHub at [https://github.com/jasojone/Cis17c2021](https://github.com/jasojone/Cis17c2021%20).

# Approach to Development

The development process was one which was derived form a beginner standpoint. With that said, I started off with first mapping out how the game would be structured. What elements of the game would require classes, what classes would require structures, and what data structures would be used. The first rule of business was planning out the data structures. The first data structure that I was set out on using was the map. I wanted to use a map for the deck of cards. Since each card not only has a suit and a power, I decided to create a map of maps so that I could implement the ascii card art to improve the visual appearance of the game. The players cards are broken up into two separate containers as it would be in the actual game. A stack was used to hold the cards that are in play. A stack worked perfectly for this purpose because the cards are simply just taken of the top and compared for play. I decided to use vectors for the cards which are won from the stack card comparisons. This was primarily to aid in easily using the shuffle algorithm due to the rule of shuffling the cards won before returning them to your hand in play.

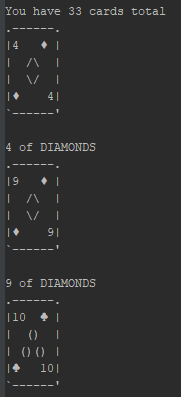
The main concept of development which was implemented was test driven development. Every time I created a new class, function, or when writing even just a few lines of code testing was done. This ensured that the new code written worked and would be usable in the project. By testing frequently this enabled for a streamline program to be written. Consistent functionality allows for less debugging when it is time to test run. When there where bugs and or errors that needed to be addressed the fact that not many lines or logic had been added made it quite easy to debug and test.

Here are some examples of code written for the sheer fact of testing.



These out statements allowed me to keep track of where the cards are at all times of the game.

There was also a print hand function which was just utilized to keep track of the cards in play. This ensured that cards where still 52 unique cards.



This print would print the players card in their entirety.

# Version Control

Version control was utilized throughout the development process of this project. For version control GitHub was employed. Updates were committed and pushed when creating new functions, classes, as well as changing logic within the classes and game loop. This enabled me to easily revert to older version if necessary and keep a clos eye on the development process.

GitHub allows for the developer to keep a consistent and accurate set of project versions. Not only can one see the latest version published, one can see the lines of code changed. This allows for the developer to keep track and have the option to revert to previous code to keep the development process moving toward completion.

# Game Rules

In the basic game there are two players, and you use a standard 52 card pack. Cards rank as usual from high to low: A K Q J T 9 8 7 6 5 4 3 2. Suits are ignored in this game.

Deal out all the cards, so that each player has 26. Players do not look at their cards but keep them in a packet face down. The object of the game is to win all the cards.

Both players now turn their top card face up and put them on the table. Whoever turned the higher card takes both cards and adds them (face down) to the bottom of their packet. Then both players turn up their next card and so on.

If the turned-up cards are equal, there is a **war**. The tied cards stay on the table and both players play the next card of their pile face down and then another card face-up. Whoever has the higher of the new face-up cards wins the war and adds all six cards face-down to the bottom of their packet. If the new face-up cards are equal as well, the war continues: each player puts another card face-down and one face-up. The war goes on like this if the face-up cards continue to be equal. As soon as they are different the player of the higher card wins all the cards in the war.

The game continues until one player has all the cards and wins. This can take a long time.

Most descriptions of War are not clear about what happens if a player runs out of cards during a war.

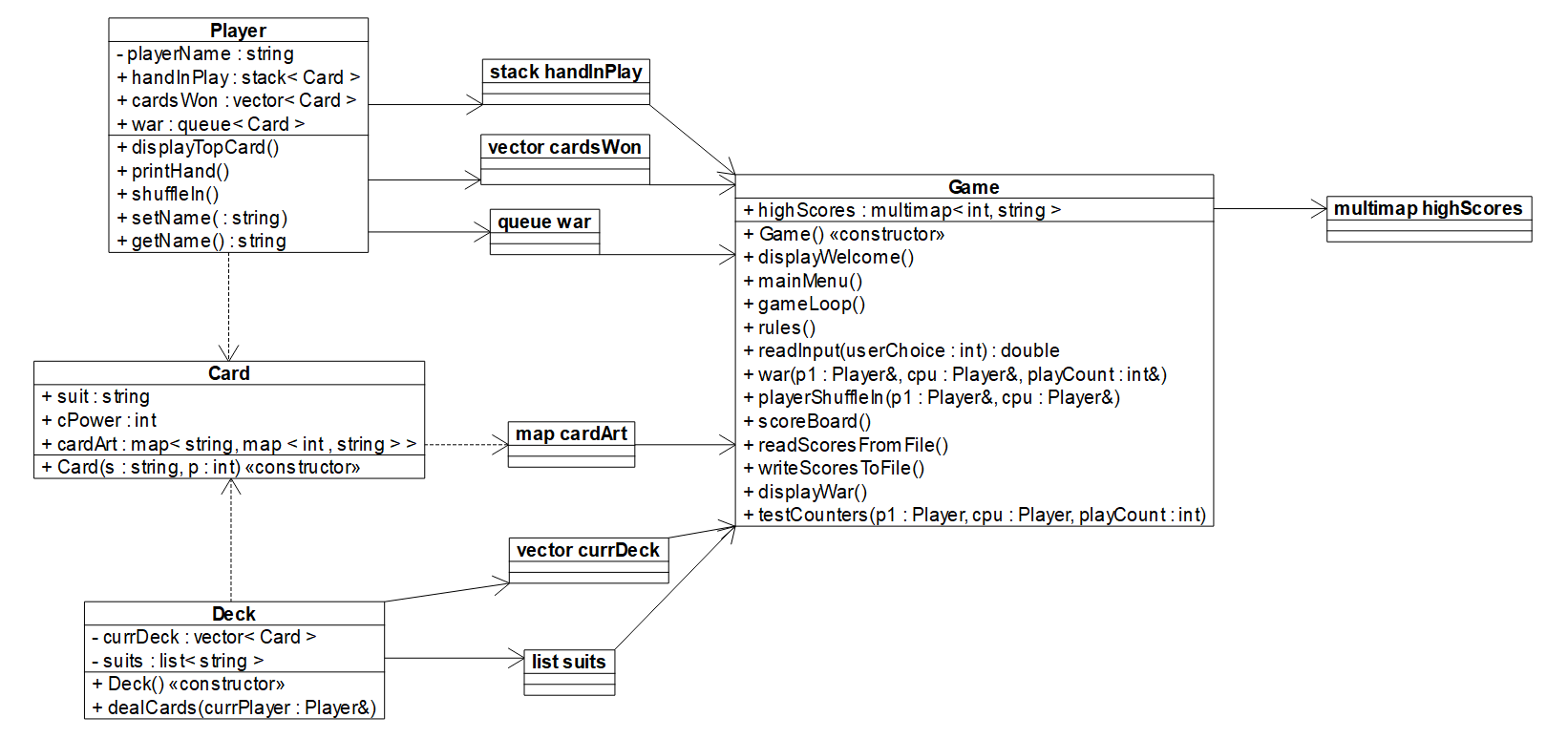
In the event…

If you do not have enough cards to complete the war, you lose. If neither player has enough cards, the one who runs out first loses. If both run out simultaneously, it is a draw. Example: Players A and B both play sevens, so there is a war. Each player plays a card face down, but this is player B's last card. Player A wins since player B does not have enough cards to fight the war.

# Description of Code

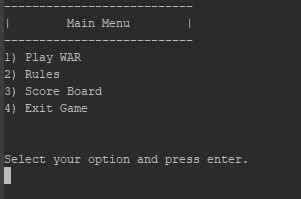
Organization of the project was based in object orientated programming. Although there was no inheritance or polymorphism there was still a need classes and modularity. Dependencies will be demonstrated in the UML. The Card struct will contain the attributes of the Card. Its attributes include the cards suit, the power of the card and the cards ascii art. The Deck class will contain the attributes of the deck of cards. The Deck attributes a vector of cards, a string array with suits for map keys. The Player class will contain the attributes of the player. Its attributes include players name string, hand in play stack, cards won vector, and war cards queue. The Game class is a designed to hold all the attributes of the game. The Game class will allow for the main function to maintain a minimalistic appearance.

# UML Diagram

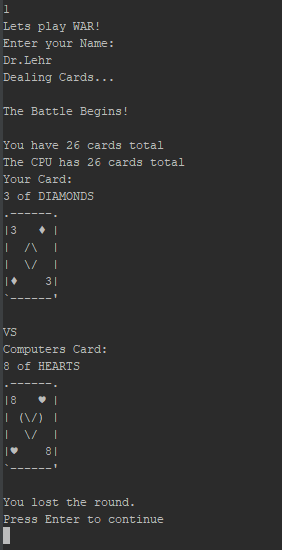


# Sample Input/Output

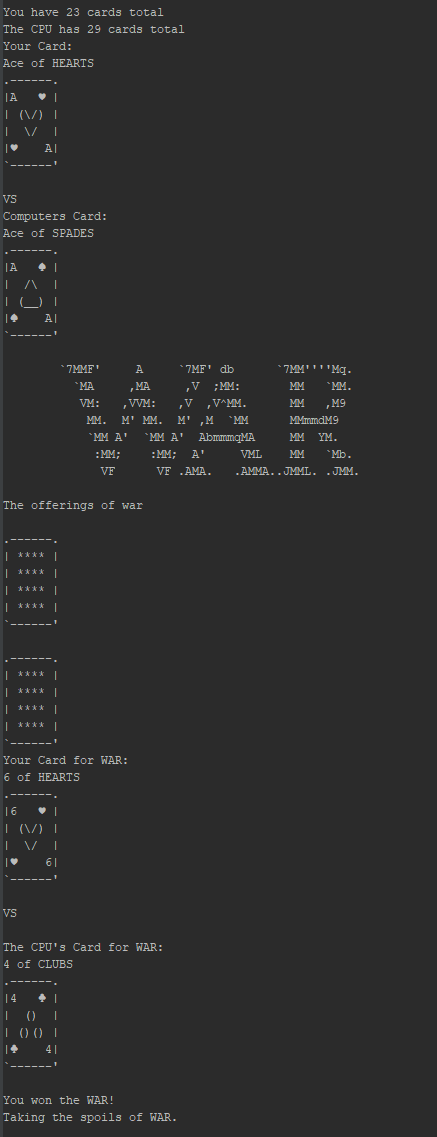
The Main Menu.



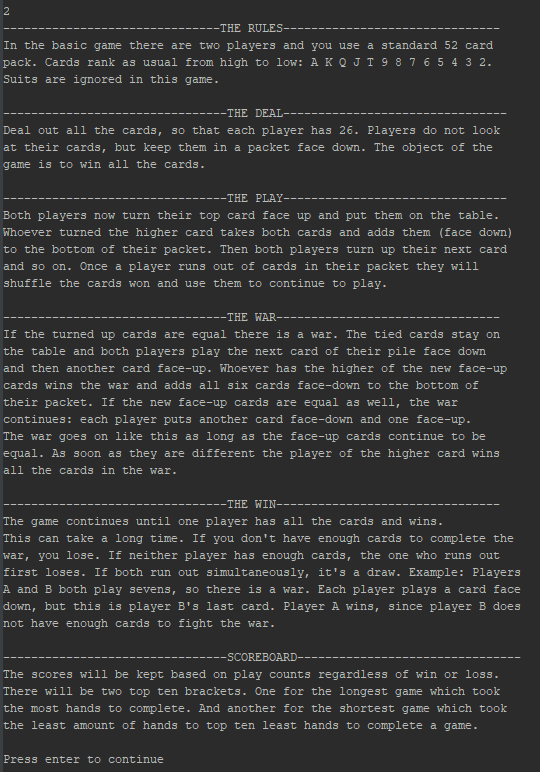
After User selection



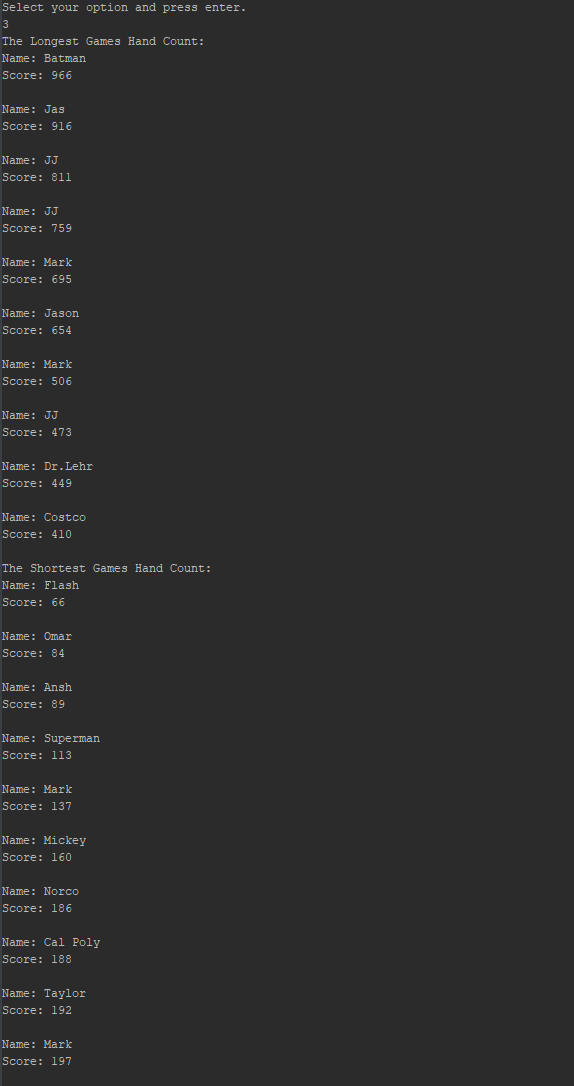
This shows the User selected option 1 and starts the game.



This is what an instance of war looks like.



The rules menu option. This will display the rules of the game.



And finally, the scoreboard option. This will display the most hands played, and the least hands played.

# Check Off Sheet

1. Container classes
   1. Sequences
      1. [list](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/List.html) (private member variable of Deck that hold the suits)
      2. [slist](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/Slist.html)
      3. [bit\_vector](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/bit_vector.html)
   2. Associative Containers (At least 2)
      1. [set](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/set.html)
      2. [map](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/Map.html) (multimap for high scores, map of a map for cards)
      3. [hash](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/hash.html)
   3. Container adaptors (At least 2)
      1. [stack](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/stack.html) (the hand in play)
      2. [queue](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/queue.html) (war queue to hold the war offerings)
      3. [priority\_queue](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/priority_queue.html)
2. Iterators
   1. Concepts (Describe the iterators utilized for each Container)
      1. [Trivial Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/trivial.html)
      2. [Input Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/InputIterator.html)
      3. [Output Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/OutputIterator.html) (scoreBoard to write scores to file)
      4. [Forward Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/ForwardIterator.html) (scoreBoard shortest game count print)
      5. [Bidirectional Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/BidirectionalIterator.html) (scoreboard longest game count print)
      6. [Random Access Iterator](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/RandomAccessIterator.html)
3. Algorithms (Choose at least 1 from each category)
   1. Non-mutating algorithms
      1. [for\_each](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/for_each.html)
      2. [find](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/find.html) (for readScores loop)
      3. [count](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/count.html)
      4. [equal](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/equal.html)
      5. [search](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/search.html)
   2. Mutating algorithms
      1. [copy](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/copy.html)
      2. Swap
      3. Transform
      4. Replace
      5. [fill](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/fill.html)
      6. Remove
      7. Random\_Shuffle (shuffle the deck when dealing and when shuffling in the won cards)
   3. Organization
      1. Sort
      2. Binary search
      3. [merge](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/merge.html)
      4. [inplace\_merge](http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/inplace_merge.html)
      5. Minimum and maxim

# Works Cited

Rules of card games: War. (2021). Retrieved 26 April 2021, from https://www.pagat.com/war/war.html