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 of 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 were 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 scoreboard.

The War card game project has been a lengthy process. The project was created over a month. Due to extensions, I was able to implement better modularity, check off additional requirements and put the final additions to the project. Because I was staying on top of every due date, I was able to improve upon the project. I was not only able to improve the code but also write well-defined documentation which can be seen in the code. Since Wars' creation, I spent over 9 hours a week working on War in C++. Writing this game was a phenomenal learning experience and it will be a great pleasure to look back at it in 10 years.

The project can be found on my GitHub at https://github.com/jasojone/Cis17c2021.

Approach to Development

The development process was derived from a beginner's standpoint. With that said, I started 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 player's 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 off 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 that 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 is written worked and would be usable in the project. Testing frequently enabled for a streamlined program to be written. Consistent functionality allows for less debugging when it is time to test run. When some bugs and or errors 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.

```
void testCounters(int playCount, Player p1, Player cpu)
{
        // print play count
        cout << "Play count " << playCount << endl;
        // print card count held in player's stack in play.
        cout << "Your hand in play " << p1.handInPlay.size() << '\n';
        // print the card count in player's pile of won cards.
        cout << "your winnings " << p1.cardsWon.size() << '\n';
        // print card count held in cpu's stack in play.
        cout << "CPU's hand in play " << cpu.handInPlay.size() << '\n';
        // print the card count in cpu's pile of won cards.
        cout << "CPU's winnings " << cpu.cardsWon.size() << '\n';
        // print the card count in the war queue.
        cout << "WAR queue " << p1.war.size() + cpu.war.size() << endl;
        // add all the cards up and print them.
        cout << "Total cards in play " << p1.cardsWon.size()+p1.handInPlay.size() + cpu.cardsWon.size()+cpu.handInPlay.size() + p1.war.size() + cpu.war.size() << '\n';
        // print the suit, value and ascii art for all card held by player.
        cout << "Your Cards" << endl;
        p1.printHand();
        // print the suit, value and ascii art for all card held by cpu.
        cout << "cpu's cards" << endl;
        cpu.printHand();
}
```

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 were still 52 unique cards.

```
You have 33 cards total
.------.
|4  ♦  |
|  /\  |
|  \/  |
|♦    4|
`------'

4 of DIAMONDS
.------.
|9  ♦  |
|  /\  |
|  \/  |
|♦    9|
`------'

9 of DIAMONDS
.------.
|10 ♣  |
|  ()  |
| ()() |
|♣   10|
`------'
```

This print would print the player's card in its 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 versions if necessary and keep a close 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.

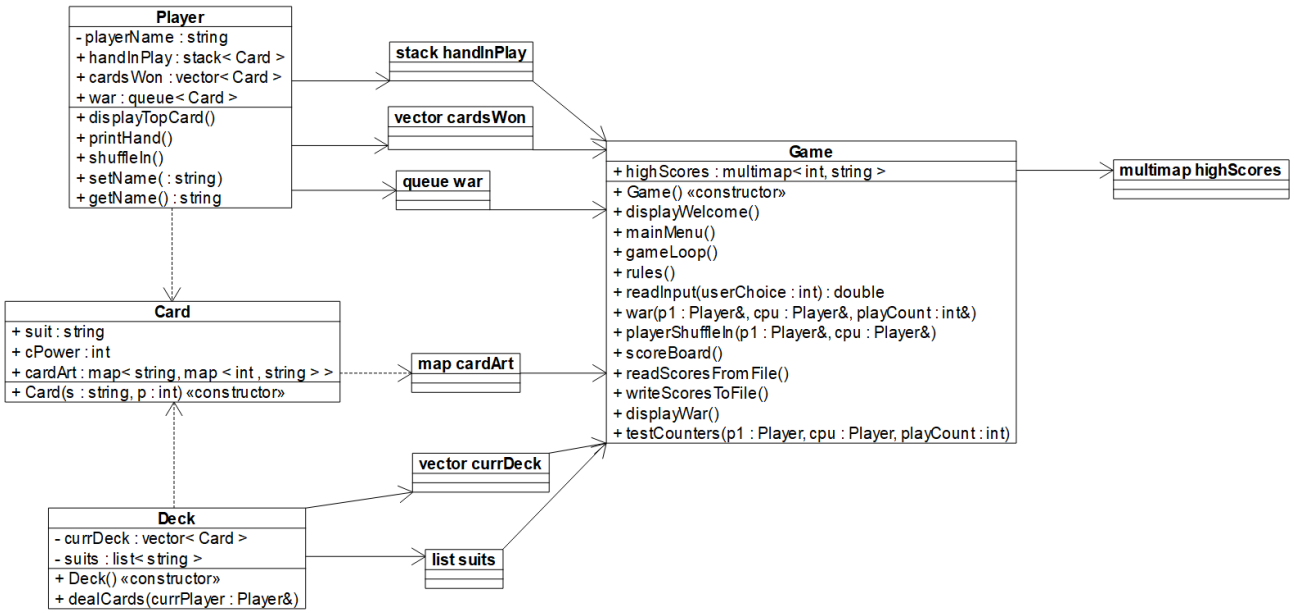
Game Rules

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 cards 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

The organization of the project was based on object-orientated programming. Although there was no inheritance or polymorphism there was still a need for classes and modularity. Dependencies will be demonstrated in the UML. The Card struct will contain the attributes of the Card. Its attributes include the card's suit, the power of the card, and the card's 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 designed to hold all the attributes of the game. The Game class will allow for the main function to maintain a minimalistic appearance. attributes of the game. The Game class will allow for the main function to maintain a minimalistic appearance.

UML Diagram



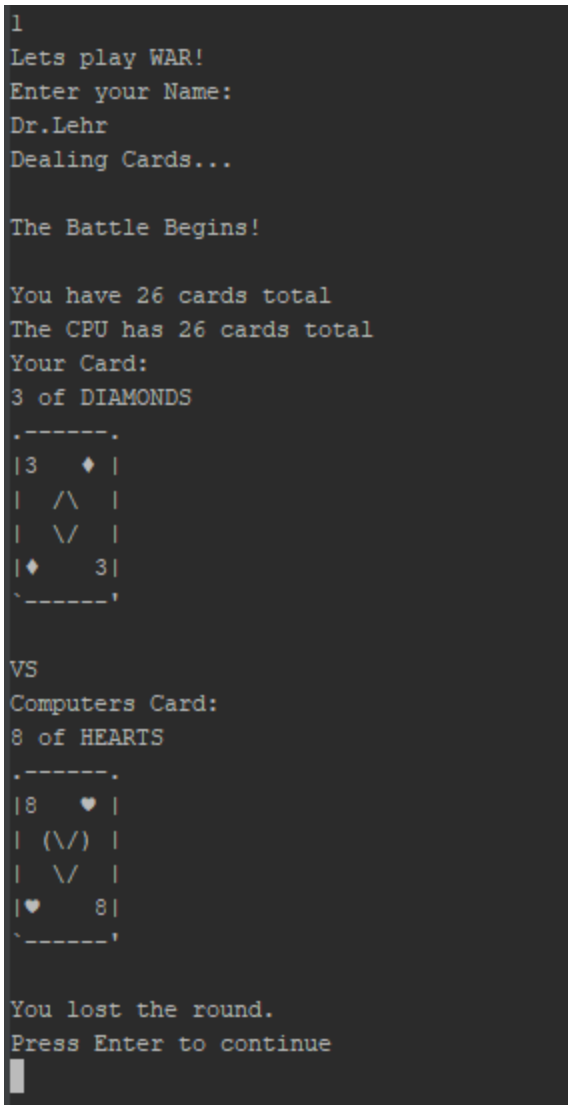
Sample Input/Output

The Main Menu.

```
-------------------------
|        Main Menu      |
-------------------------
1) Play WAR
2) Rules
3) Score Board
4) Exit Game


Select your option and press enter.
```

After User selection



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

```
You have 23 cards total
The CPU has 29 cards total
Your Card:
Ace of HEARTS
.------.
|A   ♥ |
| (\/) |
|  \/  |
|♥   A|
`------'

VS
Computers Card:
Ace of SPADES
.------.
|A   ♠ |
|  /\  |
| (__) |
|♠   A|
`------'

        `7MMF'     A     `7MF' db      `7MM''''Mq.
          `MA     ,MA     ,V  ;MM:       MM   `MM.
           VM:   ,VVM:   ,V  ,V^MM.      MM   ,M9
            MM.  M' MM.  M' ,M  `MM      MMmmdM9
            `MM A'  `MM A'  AbmmmqMA     MM  YM.
             :MM;    :MM;  A'     VML    MM   `Mb.
              VF      VF .AMA.   .AMMA..JMML. .JMM.

The offerings of war

.------.
| **** |
| **** |
| **** |
| **** |
`------'

.------.
| **** |
| **** |
| **** |
| **** |
`------'
Your Card for WAR:
6 of HEARTS
.------.
|6   ♥ |
| (\/) |
|  \/  |
|♥   6|
`------'

VS

The CPU's Card for WAR:
4 of CLUBS
.------.
|4   ♠ |
|  ()  |
| ()() |
|♠   4|
`------'

You won the WAR!
Taking the spoils of WAR.
```

This is what an instance of war looks like.

```
2
------------------------------THE 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.

------------------------------THE DEAL-------------------------------
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.

------------------------------THE PLAY-------------------------------
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. Once a player runs out of cards in their packet they will
shuffle the cards won and use them to continue to play.

------------------------------THE WAR--------------------------------
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 as long as 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 WIN--------------------------------
The game continues until one player has all the cards and wins.
This can take a long time. If you don't 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's 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.

----------------------------SCOREBOARD-------------------------------
The scores will be kept based on play counts regardless of win or loss.
There will be two top ten brackets. One for the longest game which took
the most hands to complete. And another for the shortest game which took
the least amount of hands to top ten least hands to complete a game.

Press enter to continue
```

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

```
Select your option and press enter.
3
The Longest Games Hand Count:
Name: Batman
Score: 966

Name: Jas
Score: 916

Name: JJ
Score: 811

Name: JJ
Score: 759

Name: Mark
Score: 695

Name: Jason
Score: 654

Name: Mark
Score: 506

Name: JJ
Score: 473

Name: Dr.Lehr
Score: 449

Name: Costco
Score: 410

The Shortest Games Hand Count:
Name: Flash
Score: 66

Name: Omar
Score: 84

Name: Ansh
Score: 89

Name: Superman
Score: 113

Name: Mark
Score: 137

Name: Mickey
Score: 160

Name: Norco
Score: 186

Name: Cal Poly
Score: 188

Name: Taylor
Score: 192

Name: Mark
Score: 197
```

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 (private member variable of Deck that hold the suits)
        2. slist
        3. bit_vector
    2. Associative Containers (At least 2)
        1. set
        2. map (multimap for high scores, map of a map for cards)
        3. hash
    3. Container adaptors (At least 2)
        1. stack (the hand in play)
        2. queue (war queue to hold the war offerings)
        3. priority_queue
2. Iterators
    1. Concepts (Describe the iterators utilized for each Container)
        1. Trivial Iterator
        2. Input Iterator
        3. Output Iterator (scoreBoard to write scores to file)
        4. Forward Iterator (scoreBoard shortest game count print)
        5. Bidirectional Iterator (scoreboard longest game count print)
        6. Random Access Iterator
3. Algorithms (Choose at least 1 from each category)
    1. Non-mutating algorithms
        1. for_each
        2. find (for readScores loop)
        3. count
        4. equal
        5. search
    2. Mutating algorithms
        1. copy
        2. Swap
        3. Transform
        4. Replace
        5. fill
        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
        4. inplace_merge
        5. Minimum and maximum

Works Cited

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