- Random numbers
    - Don't do rand srand its the old way
    - Class inheritance
        - struct Deck: std::deque<Cards>
            - Make it have all the elements of a standard deque class
        - Inheritance all operations and everything
            - Everything but constructors and destructors
            - using std::deque<Card>
                - Will bring over the constructor and destructor
            - Remember deque is a doubly linked list of arrays
            - Make useful functions for classes is not a bad idea even if you don't use it
                - Keep the print next time
        - For (Card c:d)
            - Is equal to
                - for(auto iter = d.begin(); iter != d.end(); ++iter)
                    - C and c=*iter;
            - Called ranged based loop
- Rand num
    - Need random bit generator / random number generator
        - Std::minstd_rand;
            - Is a pseudo random num always generating the same sequence with a starting point
            - Useful for testing so it's always the same initial state
                - To see the same replay to find bug
        - Std::random::device rng;
            - May or may not have on computer
            - Draws on entropy pool on machine
                - This candrain so it can run out of the random
    - Need distribution system for random
        - Probability distribution
        - Shape binary data to distribution
    - Std::shuffle
        - Has 3rd argument for random distribution
        - Can user minstd_rand prg;
        - std::shuffle(d.begin(), d.end(), prg);
    - If you want to get a password then use random device since it will be random dont use minstd_rand since it won't be truly random and can be tracked
        - #include <random>
    - Std::minstd_rand prg(std::time(nullptr);
        - Will initialize with time
        - But will be same if run in same time so don’t use it
    - To sort the deck

        - std::sort(d.begin(), d.end());
            - Need to compare values to sort through
                - Cards can’t be evaluated
- Encapsulation
    - Don't want to change a card value after creating it
    - Could try to make it const
        - Const Rank
        - Const Suit
        - But shuffle needs to modify the objects
    - Can also make it private
        - Make it a class
        - But makes the constructor not work since its trying to initialize a private in ember
    - As you start to look things down the more code you will need to write

```
1. class Card{
2. public:
3.         Card(Rank r, Suit s)
4.                 : rank(r), suit(s)
5.         {
6.         //can't do this
7.                 //rank = r;
8.                 //suit = s;
9.         //that's the java way
```

        - It is reinitializing them again after initializing them so it should be initialized from the start
            - This is not initialization it is an assignment. Will make them twice so learn good way
                - It is performance issues so learn good way

```
1. {
2.         //need to make an accessor for the print out to read it
3.         Rank get_rank() const{ return rank; }
```

        - It won't modify the things
        - These are “const” guarantees
        - Also can not call any non const member function
    - Suit get_suit() const { return Suit; }
    - Or you could have declared it as a friend
        - Only do it with caution
        - Mostly just leave it to the class to handle
    - If you put in a setter then why make it private to begin with
    - void set_rank(Rank r) { rank = r; }
    - But if you want to put a guard on the function to control range of values
        - So it can be private with a function to mod it
        - But if you don’t ever change it then don't add the setter

            - Ace of spades will always be the ace of spades
    - Think before you create mutators
        - Observers or getters are always a good idea to have
- Sorting
    - Objects have to be mutable so it can be changed
    - It has to be compared so it can sort the value of the object
        - But what values is important to sort by
    - Need to be able to tell one is less than another
        - To sort by ascending order
    - Need to satisfy the tricadomy law
        - a<b
        - a>b
        - a==b
        - One of these have to hold without the other two being true
    - For Card define the overloaded operations
        - bool operator==(Card a, Card b);
            - If you have one the other should be the opposite

```
1. bool
2. operator==(Card a, Card b)
3. {
4.         //for entire value of card not just for game of war so suit matters
5.         return a.get_rank() == b.get_rank() &&
6.                 a.get_suit == b.get_suit();
7. bool
8. operator!=(Card a, Card b)
9. {
10.        return !(a==b);
11.        //when is a!=b is when a does not equal b
12. }
```

        - Don't care where they live in memory just the value
            - Value oriented program
            - Worry about what they do not how they work
            - Java compares addresses first to see if they are from the same place
                - Bad on performance
                - We don't care where they live just what their value is
            - Can reduce things mathematical so we can maximize the complexity from the value
                - $\sum_{i}^{n} i = \frac{n(n+1)}{2}$
                - Which is a lot faster bitwise shift

        - For getting the computer need to give a value more weight to have the first order happen then the second comparison
            - I.e. suits more important than rank
            - This is a lexicographical order
                - Like string comparison
                - Need to keep going until it is finished
- Product order is not total order
    - return a.fet_rank() < b.get_rank()
    - && a.get_Suit() < a.get_suit()
- Instead do if statement

```
1. if(a.get_suit() < b.get_suit())
2. {
3.         return true;
4. }
5. else if ….
```