- Suit and Rank / set theory
    - And is an operator
    - Card = S and R
        - S X R
            - S cross R
                - Cartesian products
                - Combines every element in one set with every element in another set creating pairs
                    - Get the set of ordered pairs (ace, spades)
        - S X R = {(S,A),(S,2)...}
    - C++ does not but other languages have Tuple
        - Tuple will pair
            - Tuple is the generic pair of pairs, triples, quadoples
    - Look at haskcal
        - Functional language
        - Based on type theory
            - Set is collection of elements / values
            - Type is a set of values
                - Only different is in definition
    - Reread discrete book
        - To learn set theory
    - When a value is a type then it generally does not belong to any other type
    - The x is not cartiaions product here by the product type
    - For tuple in c++ use standard
        - std::Get<0>(x);
    - Tuple is like a class
        - Contain the different value types
- Now we want to add a joker to our deck
    - Can make 2 sets on standard card
        - SC=S X R
    - We also can have the red black for colors
        - Color = {R,B}
        - JokerCard = Color
            - But it is still different than that
                - Since it does not equal same card that is Red
        - Also color is not the card so it needs to encapsulate the color
            - JC={(B),(R),}
                - struct JC{
                  Joker c;};
    - So a card is either standard card or joker cards
        - Definitely not both (not and)
        - Playing card = standard rd U joker card
        - Basically just append the (R),(B) on the ned of the card set

        - Card = {(S,A),(S,2),...(R)(B)}
    - The enum for joker will have same bit pattern than suit and rank so how do we represent with and's and or's
        - Without a hack/ optimization to start it differently
    - We have
        - struct PC {
            SC s;
            JC s; };
        - Not good since this is and but we don't have an ace and a joker so use union
            - Union PC {
                SC s;
                JC s; };
                - It is either one or
                    - So when it is initialized you can only access the one you assigned first
                        - It is very unsafe and crates bugs
                        - But is useful
                - Need to know how you initialized it before you use it
        - PC = SC U PC (should be half a box instead of U)
            - Disjoint union
                - Forces the 2 sets to be disjoint so the two sets are disjoint
- Disjoint union set
    - Create a label
        - L={0,1}
        - Unique value to be combined with this operator to represent
            - ${}^{o}_{S}U\ {}^{1}_{T}$(idk what that is anymore)
    - SC' = {0} X SC = {(0,(A,S)),...}
        - Add 0 to every element to the set
        - And joker do that same with 1 to get the cross
            - JC' = {1} X JC = {(1,R),(1,B)}
    - Now we can look at the first value and see if it is SC or JC
        - PC = SC' U JC'
        - Do this by
            - struct PC {
                union Data {
                    SC s;
                    JC j;
                };
                Int T; };
                - Tagged union / piscriniate
                    - Lets use know what we are looking at

        - This is called a sum type
            - PC = SC + JC
        - Or variant which is the data structure that we made
- If you want or this is what your design space will be
    - If you have and it will be classes and structs
- Variant is new in c++ and is in c++17
- Program example of Card creation
    - Use standard variant to create a variant
        - It is different to do and will lend to bugs

```
enum Suit { …
enum Rank { …
enum Color {
        Red,
        Black,
}
class JokerCard{
private:
        Color color;
public:
        JokerCard(Color c)
                : Color(c)
        Color get_color…
//represents a playing card which is either a standard card or a joker card
//this is a tagged or disjoint union
union PlayingCard data {
        StandardCard sc;
        JokerCard jc;
};
//if you create it as standard card it can't access it as if it was a joker
class PlayingCard {
private:
        Int kind;
        //this is the tag or discriminator (at least our first pass on it)
        PlayingCard data;
        //this holds underline combined
public:
        PlayingCard(StandardCard sc);
        PlayingCard(JokerCard sc);
                //will create standard card
                //then joker around it
```

- Or

```
1. PlayingCard(Suit s, Rank r)
2.         : kind(0), data(s,r)
```

3. {}
4. PlayingCard(Color c)
5. : kind(1), data(c)
6. {}

- We need constructors in the union as well for this
    - 0 is normal card and calls normal constructor
    - Union is public by default

1. union PlayingCardData{
2. PlayingCardData(Suit s, Rank r)
3. : SC(c)
4. …
5. : JC(c)

- Kind should have labels instead of 0 and 1
    - Make a enum than we can use the name
- Construct deck will call
    - Card{Queen, Hearts}
- To print it out you now need to know if it's SC or JC
    - operator<<
        - If (c.is_standard() return OS;
    - For get rank it now needs
        - return data.SC.get_rank()
            - Since its now nested
        - Or can return standard card
            - StandardCard get_standard() const {return data.SC}

- This joker has 63 bits of wasted data with this implementation
- To prevent using joker as standard card
    - Use assertion
        - Get_rank
        - assert(is_standard());
    - Remember asserts go away when you build for release
    - To put guard rails on your own thing so you don't use it wrong (or other people using your shit)
- To use variant
    - Struct PlayingCard
      : std::variant<StandardCard, JokerCard>
      {};