Programming productivity

- Safety to velocity
    - How safe are you being vs how much are you producing more code
        - How many bugs you want vs how much time you want to spend
- Abstraction to performance
    - High level abstraction there is a penalty to the efficiency of your code
        - Abstraction penalty
    - You can use array rather than vector but that needs knowledge of the array before implementation
        - Vector can solve this with push_back and such but this comes with a penalty
    - Indirection
        - Through a pointer
        - Assembly language
            - Load gets the data out of an address to be put into a register
                - Load is keyword
            - Everytime you get a pointer it is a load instruction
                - Cost of loading is heavy since it is not in register
                    - Need to look at the cache
                        - L1 L2 is the cache
                    - If it's not in cache then it looks at ram / memory anc will cycle it to cache to go to the register
                        - If not in memory it needs to go to disk
                - Every step up it is more expensive to run
                    - Everytime you go to main memory it is a huge hit on performance
            - Data oriented design
                - Instead of objects
                - Don throw everything into an object but look how your data will work so it will not hurt your performance by putting all the data in one place
                - Getting data where you need it at as little time as possible
            - Ruby does not scale
        - If you use a lot of pointers you will incur a lot of penalties
            - Have to keep going to main memory
        - If a language has no pointers everything is a pointer
            - C# or Java for example
            - These will have a cost to them
            - The array will be an array of pointers
                - Will have 2 level of indirection
                - This will take a log of performance cost to run
                - Don't write python for scale
                    - Good for prototyping

- Type as sets of value
    - color={red,blue,green}
    - Cross product is the product of every element in 2 sets
        - Cards = Suit X Rank
            - Suit = { C,S,H,D}
            - Rank = {A,2,3,4,...,10,J,Q,K}
    - To make a set like this in code

```
#pragma once
//suit…
std::string clubs = “clubs”;
//find multiple selection to write better in your programming software
/*
Don’t do this it is worst way to make it a string since we don’t need to
know all the spelling or hole words of the suits or ranks
*/
struct Card
{
        std::string suit;
};
```

        - Terrible abstraction
            - We want to be able to compare suits for equality or order them
            - We don’t need the exact spelling of the words
            - This does work but has penalties for pointers and such
        - Can make it char which is a bit in length
            - Make them uppercase

```
char club = ‘C’;
//Ranks
char ace = ‘A’;
struct Card
{
        char suit;
        char rank;
};
```

            - This is no safe because it can be coded to be suit=’Z’;
                - Everything is public
    - Actually use enum to do this
        - Enum!!

```
//constructs a type that defines an enumerator set of values
enum Suit
{
        Club,
        Spade,
        Hearts,
```

```
7.          Diamonds,
8.  };
9.  enum Rank
10. {
11.         Ace,
12. };
13. Suit suit = Hearts;
14. struct Card
15. {
16.         Suit suit;
17.         Rank rank;
18. };
```

- The underlying value is int starting with int 0 but we use them by their name usually
- Now it is safer since it can't be outside of this set so it can't be anything but the four suits'

- enums have no abstraction penalties