- Midterm review
    - Tagged union
    - Inheritance
    - Virtual functions
        - What doe and how used
    - Public private protected inheritance
    - Static and dynamic types
    - Dis of virtual functions and pure virtual functions
        - Virtual may be overwritten but pure virtual must be
        - Polymorphic has at least one virtual function
        - Abstract class has at least one pure virtual function
        - Every polymorphic class has a pointer to the virtual table
        - Can’t create an object of an abstract type
        - Write abstract class to organize an abstract set of values that otherwise would not be an object to be used
            - Parts of multiple classes
                - Card has values but is not the class
    - Tagged union
        - Playing cards is either joker of sc but not both
            - Provide method of representing alternatives
    - Inheritance
        - Effectively same problem solved by tagged
        - Representing a set of alternatives
    - Functional languages only have variance and unions to inheritance
    - Why choose between the two
        - If set of variance won't grow then variant is good but inheritance is growing it is better
        - If object is returned by value then variance is better
            - But if never passes then inheritance is better
        - If data is recursive then you want inheritance hierarchy and not variance
    - Static dynamic types
        - Static type is never changing or fixed types
        - Dynamic type whatever is referred to when called. The runtime type
            - f(value & v)
                - V is static value
            - If called with f*new number(...))
                - Then the dynamic type of v is number
            - Behavior of function depends on dynamic type of V
                - Cout << V; with depend on if it value or number to see what it does
            - This is a polymorphic behavior
                - Its behavior is defined by the type provided
        - Polymorphic type has polymorphic functions

    - Interface
        - C# and have
            - Abstract class with pure virtual functions
            - Allows the change of implementation without change of usability
                - User -> interface<- implementation
- Not test
- Json storage
    - Struct value { ~value(){} };
    - Struct array : value, vector<value *> { ~Array()}
        - Array(int n) : value(), vector<value *> (n)
        - {}
        - //this allows the vector to set size to n
        - //vec<int>(100) will create 100 elements
        - //left shift by n is 1 + 2^20 for new array(16620)
    - };
    - kill(new Array(16620))
        - Void kill(value * p) {
            - Delete P}
        - What destructor is called the array of the value? The destructor is not virtual so it calls the base case and it deallocated pointer and not leaked that memory
        - If virtual function in base class it needs virtual ~value() {} in the base class
            - You better have a virtual destructor
            - Don't have virtual destructor if not polymorphic class
    - Make copy of value class we give pointer to a value value * copy (value * p) will return a copy of P that is distinct so we can modify on without affecting the other
    - Value * copy (value * P) {
        - //return new? (*P); // we don't know what type the ? is
        - return p->clone();
            - //is a virtual function that returns a copy
    - Struct value {
        - Virtual value * clone () const = 0;
    - Struct Array : value, vector<value *>
    - {
        - Array * clone() const override
        - {
            - Return new Array(*this);
        - }
    - };
        - This is a virtual constructor pattern
            - Constructors can’t be virtual so need this pattern