- Contract
    - An agreement of expectations on interface (between developer and user)
    - Wide contract
        - Has a loose preconditions
            - Accepts wide variety of input
            - It accepts everything
    - Narrow contracts
        - Strictor preconditions
- Look up mathematical classifications of functions
    - Functions are in 2 stats
        - If a function takes T inputs and has an output f is U. Is it a function T to U
            - f:T -> U
        - Partial functions
            - Does not map every input to an output
                - f(x)=1/x where x!= 0
            - Every function we make is a partial function
                - If it maps everything it is a total function
                    - Mathematicians hate partial functions
            - Parital is more narrow
- C++ allows for undefined behavior
        - Allowed to dereference pointers no matter what
    - Java needs all behaviors defined a wide contract
- Call out of contract you get undefined behavior
    - Just because precondition is violated it does not neciatilly lead to undefined behavior
- Posix is underlying operating system for most linux
- If you have wide contract should you have output checking?
        - Yes you have the responsibility to check the results since it may have failed and wants to return something incorrectly
    - Program terminates with an uncaught exception

```
1. //hpp examples
2. Extern SqrtFn & sqrt;
3. //Extern is a variable i have dedicated but haven't used it ?? look this up
4.         struct sqrtFn{
5.                 virtual Double Operator() (double N) const = 0;
6.         };
7.         //not practicable way to do this just for this example
8.         //Preconditions for victosl
9.         [[pre: N > o ]]
10.        [[post R : R*R==N]]

1. //cpp examples
2. //define class to override
```

3. Struct Wide_sqrtFn : sqrtFn
4. {
5. Double operator()(double N) const override
6. [[post: N >= 0 ? R*R==N : isNan(R)]]
7. {}
8. };
9. sqrtFn sqrt = * new Wide_sqrtFn();
    - Provided substitution for what -N would do but this still breaks precondition of N >= 0 so
        - sqrt(π); // works though it will get the ascii code and sqrt that not actually pi
        - sqrt(-1); //does not work
    - Would have been fine if only wide existed and was not derived
    - Language requires all preconditions for override to be the same
- Going to derived can weakest preconditions and strengthen post conditions
    - But with language features it says we cannot do either
- Every power of 2 only has 1 in the bit representing so is_power2() would check if bits have more then 1 one in it
- Derived class cannot narrow preconditions
    - If derived class has narrower set of requirements then you can't use the derived object everywhere you could use base object vilate substitution principles we were specifically broken Liskov substitution principle
    - Power of 2 (n-1) & n
        - The bitwise &
        - N-1 bitwise and n will get all zeroes if its a power of 2
        - Power of 2 is 1 and rest zeroes so 10 or 100 or 10000000 all power of 2
            - Or easiest to left shift is and check if its equal to zero
- Substitution / polymorphic
    - Voo f (Value **)
        - Don't know what object are given
    - If struct B{ };
        - Dtruct D : B {int N; };
        - D is declared to be a subtype of B
            - B is a superclass of D
        - If looked at with set of values
            - D's set of values is a subset of B's set of Values
    - Subtyping isn't restricted to classes we can have the breakdown of numbers
    - Conversion / coercion / cast
        - Cast are specific other two are not
        - Changing type to convert to another to work
        - sqrt(double); sqrt(int);
            - Converts int to double to do it
        - This is not polymorphic it is just changing it

- Liskov substitution principle
    - She made language called clu where this comes from
        - Had interesting idea about subtyping / polymorphism form that language
        - Let p(x) Be a property(s) probable about an object x of type T. the p(y) should be provable for an object y of type S. if S is a subtype of T.
    - Can call operators explicitly
        - The () operator
        - f.operator()(N)
        - f()(f,n)
        - f(n)
            - All calls something //for sqrt Fn
    - Liskov is a design principle if it does not hold the hierarchy is a bit skewed (design is off)
    - For versions compatibility you can add more but never take any any
        - Binary compatibility all binary and size need to always be the same source code compatibility can change layout and size just as long as it is still there