- Defensive programing
  - Adding guards to prevent user for misusing it at least while you are testing it
    - Shipping usually does not ship with guards turned on
- Design by contract
  - People providing implementation are the reader
  - And users are clients
  - This is a contract between the two parties
  - We are not implementations of the standard library we are the consumers of that library
  - If I call a function (correctly) what guarantee do I get (how it will / should function)
  - 2 function expectations
    - Preconditions, postcondition
    - Preconditions
      - What needs to be true before calling this function
    - Postcondition
      - What will outcome of the function be
  - If you satisfy the precondition I promise this postcondition will happen
    - A trabut on function is something to look at it with the functions
      - Double sqrt(double N)
        [[pre : N >= 0]]
      - So now sqrt(-1)
        - Will not satisfy the precondition sqrt(-1) normal returns Nan so it does not break just sends Man back
          - Leads to unpredictable behavior
      - Compiler can now act on the pre: so if it is false it can handle it
      - Can do same as pre with post
        - Post takes identifier of what result is (R)
        - [[post R: R*R==N]]
          - If R*R != N abort / terminate
          - Can provide a way to handle it but default is terminate
          - Calls
            - if(!(N>=0)
            - {
            -         on_compares();
            -         Terminate();
            - }
        - Double sqrt(double N)
          - [[pre : N > = 0]]
          - [[post R: R*R==N]]
          - [[assert : e]]//c++20
            - The post calls

            - if(!(R*R==N) …
      - Global var gets a bit weird
      - If you have N in pre and post N should not change
        - It can fail if N changed then post will be wrong
          - Undefined behavior
- Ex : sort(vector<double> & v);
  - Sorting a vector of double(v)
  - Precondinot nothing in the vector can be Nan
    - If you compare 2 Nans Nan < Nan
      - It is false same number is not less than itself
      - However,
        - If you compare 2 Nans
          - Nan<= Nan
            - Is false
          - Nan == Nan
            - Is false
          - Nan != Nan
            - Is false
          - It is both equal to itself and not equal to itself
    - Can call none of to prevent that
- sqrt((vec<double> & v)
  - [[pre: std::None_of(v, [](double) {return std::isNan(N);})]]
  - Post condition
    - V needs to be sorted and has all same elements
      - V is the permutation of V
    - Just cuz you can write past does not mean you can in a clean way so it should not always be done
  - Cow data stature
    - Copy and write data struct nic in single thread environment but not nice in multithreaded environment
    - Has pointer that only creates new when changed
      - Look this up
    - Hash tables has a program that only has rate of collision be 1/n n=size of input
      - So the strings does not collide with another often
      - Now do we check this post condition effectively
        - We do not
  - Int * Distance(int * f, int * l)
    - Return l-f;
    - Find distance between 2 elements in a range
      - For finding middle of vectors or containers
    - But this works

- In x, y;
- Distance(&x, &y);
  - But that is garbage data
    - Effective undefined behavior
      - No guarantee where they will ies in memory
      - So there is a precondition
  - [pre: Axiom: is_reachable(f,l)]]
      - //is_reachable makes sure you can
      - //reach f from l
    - Axiom is something we can’t define f can be incremented to find l
      - Way to write a scape hatch to unspillable conditions
  - Pre and post is mostly c++ and adia (another language)
    - To think of this ask what behavior we want to do
      - How costly are the tests to check
      - And should they be there after release or not
    - Topological sort
      - Can find the dependent and cycles to find what needs to happen before something else
        - Used for paralyzation (parallel compiling)
- Audit
  - Audit : is if its expensive
  - Default : is cheap
  - Axion : is infinite expensive
    - [[pre Audit : …]]
  - Can tell the compiler to assume it is true
    - For optimization
      - [[asume: P != 0]]
        - if(P)
        - Sort
        - *p
      - So now if(p) and sort is true it does not do it anymore
      - Time travel optimization
  - Pre and post if they are false it does not automatically break so it becomes undefined

- So it optimizes but is not desired. (cautious when using)

- Contracts come in 2 flavors
  - Wide contracts & narrow contracts -> pertains to pre and post conditions
  - Wide:
    - Wide can handle multiple things so it can be larger range of values as input
  - Narrow:
    - Will terminate if it is outside the narrow scope of inputs
  - If multiplying max x max it is undefined behavior
    - It has no guarantees to wrap that overflow making it negative
      - It is undefined
  - Wide version of sqrt does not have preconditions it accepts everything
    - Double sqrt(double N)
      - [[pre: true]]//does N >= 0 -> T
        - It's always true
        - But true does not imply that N >= 0
        - T is typology (discrete math)
      - [[post R : R*R == N]]
    - More from narrow to wide weakens the precondition and strengthen post condition
      - [[post R : R * R == N]]
        - if(N>=0)
          - R*R==N;
        - Else
          - isNan(R);
  - After release of ver 1 with ver 2 should you widen sqrt?
    - Yes i’m still in contract to say your input still works just more does too
    - Other way does not cuz you break contract to say what used to work now does not work
    - Javascript sqrt
      - Function sqrt(N)
        - Takes and N
          - Including string and arrays and extra
        - sqrt(“π”)
          - Works but does not return the right answer but rather the sqrt of the ascii code of pi symbol
        - If you strengthen the precondition you will break someone else's code
          - If you ship it someone will use it “incorrectly” so if you narrow it that person still will used it wrong but will break their code
  - Adding functionality won't break code but removing functionality will

- Think of software as rings
    - Wide contracts need to exist on the outer riges
        - Has to accept everything first you validate it then check if its what you want
- v[N]
    - Is narrow if n is outside it crashes
    - v.at(N)
        - Widder it throws exception if N is outside of the array
        - Sort using .At then it is a huge number of checks you don't need to do
            - Huge performance hit
- Self documenting code
    - B.s. talk on this later