Whole structure can either completely be mutable or non-mutable
Not that some part is mutable and other is non-mutable

```
let j = &mut start.y. ;        (Reference of a specific element)
println! ("element = {}", j) ;
```

Tuple is effectively "sequence of Types" ⇒ Type of Tuple.

```
let start = Point3d { x: 4, a: true, y: -2, z: 9} ;
```

It doesn't matter where you put variable until you're accessing it by name.
RUST also allows Unions.
You're not allowed to initialize all variables of union at same time.

ARRAY. doesn't have support in lower level unless chip is built that way.

- Implementation differs from lang. to lang.

Effectively sequence of multiple "same type of elements".
Index is used to access ARRAY. based on Expressions.

t is Tuple (.i=0, 1, 2.....) ⇒ t.i is not allowed in RUST

If you know size of array, do "Static Allocation" - consecutive
For Dynamic Arrays, do "Dynamic Allocation". with some exceptions.

```
Mult (int n, int mat[n][n]) {  }
```

On Stack Frame, we can allocate when this function is called ⇒ Effectively Static Allocation.

arr[i] gives reference to that element ⇒ have both L-value & r-value.

```
arr[i] = k        j = arr[i].
```

Two different things.

arr(i,j) → access (i,j)th element in square space

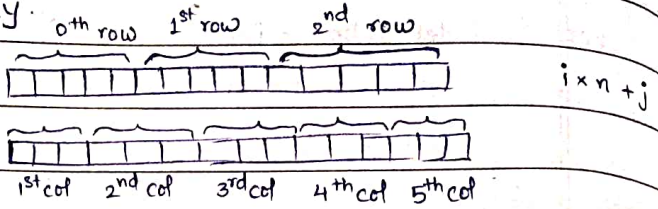
arr[i][j] ⇒ jth element of ith array in Array of arrays.

Implementation of arr(i,j):

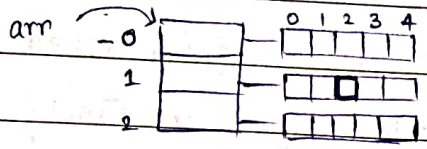
- Multidim. array as Singledim. array.

arr[3][5]

arr(i,j) < Row Major / Colum Major



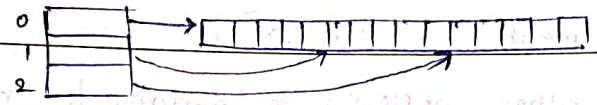
Implementation of arr[i][j] (Array of arrays)



Can be stored in Non-consecutive location.

If you specify size, most probably, compiler will allocate in consecutive location

C does like this (below)



arr[i][j] ✓

((arr+i)+j) ✓

* is NOT commutative

i[arr][j] ✓

j[i][arr] ✗

You can extend this idea to any dimensional array.

```
arr = malloc(3 * sizeof(int*));
for(int i=0; i<3; i++){
    arr[i] = malloc(5 * sizeof(int));
}
```

The above array is allocated in Heap memory can be anywhere

We are asking malloc to give 12B (not 3 elements)

arr[i][j] simplifies the way you are handling memories. (single.dim. arrays)

C doesnt store Metadata. - can't tell whether it is out of bound (range).

RUST handles in Type-theoritic way - can detect out of bound.

Array allows indices - You've check if it is within bound - run time check needed.

Structure & Tuple are safe in this context. (member variables not indices)

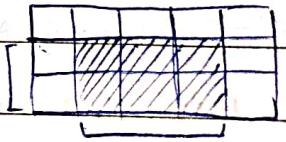
Some languages allow Slicing. - Python, RUST.

arr[3:5]

In RUST, slices are of different types.

FORTRAN: Slicing in diff. dim.

arr[1:2][1:3]



```
fn main() {
    let arr: [i32; 3] = [1,2,3];        → Statically allocated array.
    println!("{}", mem::size_of_val(&arr));     12
    println!("{}", mem::align_of_val(&arr));    4
    let n = arr[2]      → can be expression (recommended to be constant rather than variable).
    println!("{}", n);
    let arr: [usize 3] = [1,2,3];
    let j: usize = arr[2] + 5
    let n = arr[j]
```

Smart Compiler ⇓ Generic

RUST doesn't do TYPE COERCION by default.

arr[5]

Compiler does 2 things

j could be i32, u32, usize & finds.

j is used as index

(TYPE Inference) ⇒ j assumes the best possible type

At runtime, main thread will panik due to out of bound (similar to Java, ~~Python~~)

```
let arr: [&str; 3] = ["abc", "def", "ghi"];      Size = 16×3 = 48
let n = arr[2];                                  Alignment = 8
let arr: [ [0,0,0], [1,1,1], [2,2,2] ];          S: 36
let n = arr[2][0];                               A: 4
```

2D array is a collection of 1D arrays. [3D → 2D → 1D]

```
let n = arr[2]
println!("{}", n)   × Error in printing.
println!("{}", n[1]); ✓
let str = String::from("Hello");
let slice = &str[0..2];      → ..2      println!("{}", slice); → He
```

↳ rectangular obj using array / string.

Start & end ⇒ (within bounds)

&[T] → T can be of any type

Effectively slice is different type than array type.

let str = [1,2,3,4,5,6]	let mut str = [1,2,3,4,5,6];
let slice = &str[..3]	let slice = &mut str[..3];
println!("{}", slice[2]);	slice[2] = 7; println!("{}", str[2]); → 7

Change reflected in Main Array.

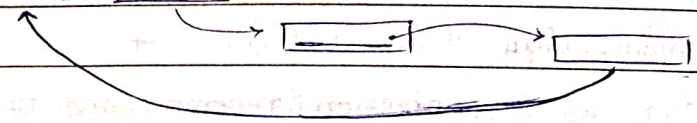
28/2/24

Earlier languages, C, C++, Python (not for users) allow Pointers.

Character pointer, Struct pointer.

Kind of data represented by Pointer Style.

[TYPE] POINTER



You can create programs at System level with pointers.

Most of programs doesn't fit in RAM as it is ⇒ Paging, Segmentation

Maintaining boundary is difficult.

Randamization of memory address.

At same load, same memory location is access

Issues with Pointers:

- Dynamic allocation of Memory in Heap of diff. sizes
- POINTERS as Reference to Memory locations.

```
int *p;
foo(){ int n=3 ; p=&n ;}
main(){
    foo();
    printf("%d", *p);
}
```

→ Variables get destroyed after completion of execution of function foo();

It may give "Segmentation fault" or print 3 (accessing illegal portion of memory which is not supposed to be accessed)

```
main(){
    int *p = malloc (sizeof(int));
    *p = 3;
    free(p);
    printf ("%d", *p);
}
```

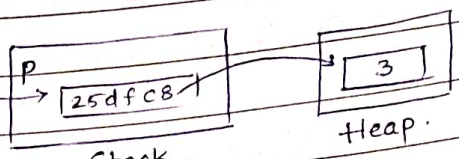
DANGLING REFERENCE

Position to which it is referring is not valid (not there). Address in pointer doesn't point to a valid location.

when you return from a function, Stack Allocated variables are deleted

```
foo(){
    int *p = malloc
    *p = 3;
    printf("%d", *p)
}
main(){
    foo();
}
```

[TYPE] POINTER



Every time you call foo(); stack variables are deleted but not heap variables leading to "MEMORY LEAK".

C has free() to prevent Memory Leaks
C++ has Destructor (destroyer function).
} Left to Programmer (Developer) to find & prevent memory leaks.

Memory leak can also happen in lang. which doesn't allow pointers.

Python & Java deals with Memory leaks using "Garbage Collector"

```
l = [1,2,3]
l = [4,5,6]
l = [1,2,3]
```

When you lose reference in heap memory, delete it.

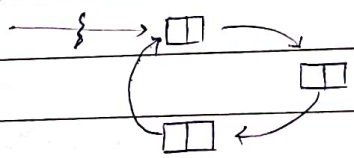
Naive :

Sophisticated: Count no. of references of corr. block that is there.

Any block with count 0 ⇒ It's not being used ⇒ can Delete it.

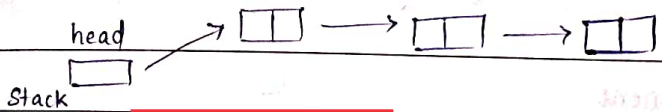
For every stack frame, you have to check

[P] If count is non-zero, does it mean block is being used?



Reference Cycle is issue in this case

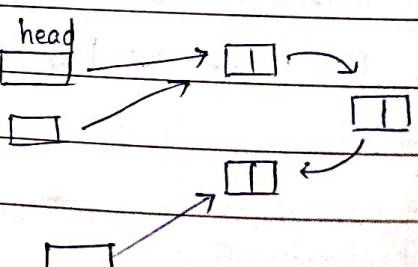
[S] How many ref are there for which lifetime has not ended (on stack & not on heap). whichever are reachable they are useful otherwise useless. (Mark & Sweep)

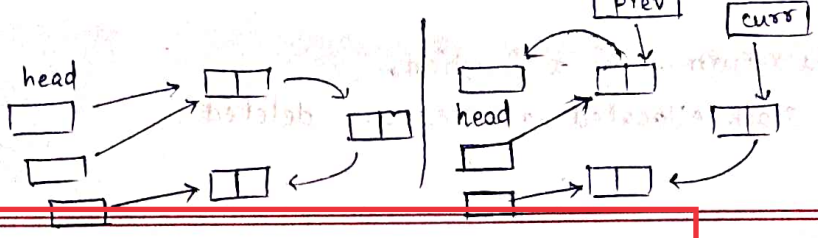


Mark useless blocks as free.

It is Recursive Algorithm. (takes up some stack space)

GC may crash if is of same order.





Reverse arrow helps to remove Recursiveness in Mark & Sweep Algorithm

GC is not single operation. GC have a problem in Concurrent running.

1/3/24

LIST: Object, list = NULL.

recursive definition

* functional programming Languages.

LISP considers program itself as List.

↳ Some lists are evaluatable (executable.) and some are data.

l = [1, 2, 3].

New languages such as Python include List Comprehension.

List Comprehension: Expression evaluates to list. (Map Reduce)

[2 * x for x in l]

→ composite data structure (type)

FILE I/O: Ex: Linux.

Writing to something which is offline.

Text file 1B ⇒ char

Image file 1B ⇒ may not be a char.

open → read mode ⇒ can't write to same file

write mode ⇒ creates if file is not there.

- Read
- Write
- Seek
- Close.

Python – readline ⇒ reads byte by byte characters. (Tape analogy)

If you don't do Seek, default operation is read next file.

Close() ⇒ Everything written to file gets flushed from cache to Hard

Assignment	vs	Equality.
Takes value of RHS & assigns to LHS.		- Boolean check
Shallow Copy and Deep Copy.		

References are effectively Pointers.

You can't create a reference without an object standalone.

References are of 2 types.

```
fn main(){
    let n = 3;
    let r = &n;
    println!("{}", r);
}
```

Type of r ⇒ reference to an integer.

All types are inferred by compiler by saying that which fits better.

If type of n is fixed, then type of r is also fixed

No dereferencing is required.

```
let n=3;
let P=3;
let e=&P;
let r=&n;
    ("{}",r);
if e==r {
    println!("Equal");
}
```

You can use reference as if they are normal variables.

e==r checks values stored in the address rather than address.

n is owner. p is owner.

e is borrower of mem.location from p.

```
let mut n = 3;
    P = 3;
    e = &mut P;
    r = &mut n;
        ⟶ ("{}",r);
r=4
if e==r {
println!("Equal");
}
```

n=4 ⇒ Error

Can't assign to n as it is borrowed.

There'll be no error if there is no equality check.

You can also create pointers.

```
let e: *const i32;
let n = 3; let P=3;
e=&P;
let r=&n;
```

(Raw Pointer) ⇒ *[Type]

Here e and r are not equal as e is pointer & r is reference.

Reference ⇒ Compare values present in address.

Pointers ⇒ Compare address

4/2/24

Function Calling:

Parameters

Return value ⇒ (C, C++ only 1 value) (Python, Java allows to return mul. values)

FUNCTIONS PROCEDURES → something that doesn't return a value

↳ something that returns a value which can / can't be used.

→ Block of codes which perform well-defined work.

→ has a NAME, PARAMETERS.

datatype of return value ← int sum (int a, int b) { ← FORMAL PARAMETERS.

⇒ Def. of function

}

sum([2,3,4,5],i)

sum(2,3); is an Expression. (Function Calling)

ACTUAL PARAMETERS

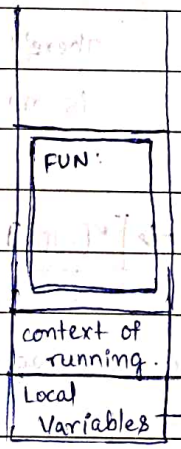
Functions are also used in Assembly Language.

```
FUN:
    ===
    JMP x++      RET
    | | | |
    | | | |
main: ===
    ===
    JMP FUN      CALL FUN
```

Inlining: Instead of calling the function, you are writing it

BookKeeping: To know where to come back to (from where it was called).

↳ is also done by Compiler itself



→ That are in Scope & within Lifetime

→ Has offset from Top of the stack

→ Instruction execution done at "Compile Time".

processes the parameters. (Type Checking of Parameters is done at Compile Time)
In C, there is no specific order in which parameters are evaluated & left to Compiler. Eg: sum(i++, ++i) gives a problem [Bad Code you should not write]. Then keeps it in variables needed there. At this point, Context of program starts changing.

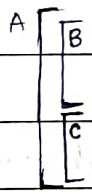
Before the function is executed, it creates its own SF and statically allocate variables. After executing, it stores in specific Register & comes to current context by popping out the function.

j = sum(i++, ++i) ⇒ Return value is stored in j.

Calling function is separate Instruction. (Everything else steps down).

Assigning value to j is a separate Instruction

→ Nested functions are complex to handle.



Pascal and Ada requires Declaration of function before Definition of function.

RUST can do Type Inference.

Header file ⇒ Declare the functions written in C file

C file ⇒ Definition of the functions.

Function Library | Shared Object.

C, C++, Java ⇒ Declaration is NOT mandatory

You can give hint by saying inline for function.

Inlining is done for "Simple functions" / small block of code

Generally Compiler decides whether to inline a function / not based on no: of variables, amount of code

```
int sum(int a, int b);

sum(1,5);
```

5/2/24

In passing Parameters, every language has default "PreOrder way", except for overloading normal operators (+) in C++.

```
int + (int a, int b)
```

5 + 6 → invoking in In-order

There are 2 ways to pass Parameters:

1) Copy the value to parameter [call by value]

2) Don't copy. Take reference & pass [call by Reference].

```
int x;
fun(int y){
    y = 3;
    print("%d", x);
}
main(){
    x = 2;
    fun(x);
    print("%d", x);
}
```

Output: 2 } Call by Value
2 (Actual parameters are not affected)

Read Onlyness of actual parameters

In C, No Reference, uses pointers.

```
fun(int *y){ *y = 3; print("%d", x);}
main(){ x = 2; fun(&x); print("%d", x);}
```

(Actual values to be changed then we use)

Call by Reference:

Output: 3
3

- Call by value/return. (copies back corr. value to Actual parameter on return)

Output: 2
3

In C, if you pass array by value, it still passes by Reference.

Call by value is a bit costly operation (copying takes time). as compared to call by Reference (copying reference (fixed size) takes less time).

Acc. to Solution requirement, we decide ~~to~~ kind (type) of parameter passing.

But dereferencing also takes some time.

```
fun(int &y){
    y = 3;
    ;
}
```

In C, default ⇒ Call by Value

In Fortran, default ⇒ Call by Reference

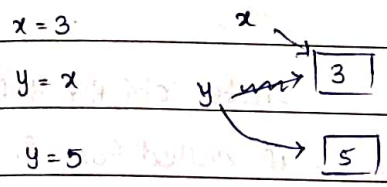
↳ l-value has to be there to store somewhere.

Java uses Hybrid model

↳ Basic datatype (int, bool, float, char) ⇒ Default Call by Value.

↳ User defined datatype (class, obj) ⇒ Def— Call by Reference

Python follows Reference model of variables.



Pass by Reference is Faster

Some lang: "read only" ⇒ Pass by Ref. but can't change value.

C/C++ fun (const int * y) { .... } ⇒ value in address of y is constant.

RUST - both Pointers & Refer

↳ Reference has some metadata associated with it.

↳ At any point of time, you can only have 1 mutable reference.

```
apply (l, f) {
    f(l[i]);
}
```

Function as a parameter

```
apply ( int l[], int (*f)(int)){
    for (i=0; i<n; i++){
        f(l[i]);
    }
}
```

⇓

Lambda function in Python.

Nesting of SubRoutine is not allowed in C/C++.

If Nesting of Subroutine is allowed, function along with surrounding context is req, which are called "closures".

```
fun z() {
    int a,b;
    fun(x){
        y = x + a + b;
        return y;
    }
}
```

6/2/24

```
print(id(j))
j = i + j
return j
print(id(j))
```

```
def sum(i, j=5){
    return i+j;
}
sum(3, 2)  ⟹ 5
sum(4)     ⟹ 9
```

immutable object

j has fixed value when j is not given

Python - All non-default parameters should be at beginning and default parameters should be at the end.

```
def fun(i, l=[]){
    l.append(i);
    return l;
}
```

print(id(l))

Creates obj for default parameter if called for first time. From next time, python will not create it again.

	Expected	Actual
print(fun(1))	[1]	[1]
print(fun(2))	[2]	[1,2]
print(fun(3))	[3]	[1,2,3]
print(fun(4, [1,2]))		[1,2,4]
print(fun(5))		[1,2,3,5]

l → [1]

List is mutable object.

Multiple call for same function using default parameter uses same obj.

id() is library function which returns id.

Here when you call this function, it creates a new object

If you not put any name then position matters.

fun(l=[2,3], i=1).

Some functions allow you to pass multiple arguments (variable # of argumen

How do you write / declare these functions?

Mostly Compiler dependent (sp. Machine Dependent).

printf(" ", ...)

scanf

int printf(char* format, ...) → You don't know type & # of arguments.

va_list params;

va_start(params, format);

i = va_arg(params, int);

f = va_arg(params, double);

You go one after another can't jump.

printf(" ", i, 0.5)

In C/C++, it is not TYPESAFE operation. (C uses Typecasting - worst case At Running time

Overloading is used when # of parameters are within a bound.

Here that bound is also not there.

printf ("%d %f", i, 0.5)

printf ("%f %d", 0.5, i)

Statically Typed ⇒ Difficult to check **TYPE SAFE**.

How do you return a value from function?

return i; ⇒ return the value of the variable i.

Just checks the def. of function & return type.

older lang. ← fun = i; } effectively same → Can't return from Nested Subroutine.

New lang. return i; }

Some lang. like Fortran prefers to return at end of subroutine.

def fun(i, L=[]) ⇒ rtn : int.

New lang. allows to return multiple values.

return 1, 2.

RUST specifies an expression → value of return type.

11/3/24

Keyword function name can specify parameters.

```
fn sum( ){
    println! ("Inside function")
}
fn main(){
    sum();
}
```

```
fn main(){
    println!(). sum();
}
fn sum(){
    println! ("Inside function")
}
```

RUST is not as sequential as C, doesn't differentiate b/w 2 programs above.

If you give a parameter, we should specify type.

```
fn main(){
    sum(5);
    sum(5+3x2);
}
fn sum (x : i32) {
    println! ("Inside function {x}")
}
```

```
fn main(){
    let i = sum(5+3x2);
    println! ("Returned {i}"); }
for sum (x: i32) → i32 {
    println! ("Inside function {x}");
    2 x x
}
```

Return Statement will be a "value" (last statement). ⇒ No semicolon.

Statement ⇒ sentence in syntax of lang. complete in itself & doesn't return value

Expression ⇒ & evaluates to a value

```
2 x x    → Expression
2 x x ;  → Statement.
```

Can we return Multiple values ? Yes

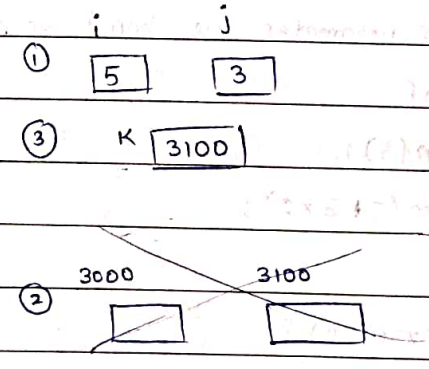
```
fn main(){
    let i = sum(5,3);
    println!("Returned {i}");
}
fn sum(x: i32, y: i32) -> i32 {
    println!("          {x}")
    x+y
}
```

```
fn main(){
    let (i,j) = sum(5,3)
    println!("Returned {i}");
}
fn sum(x: i32, y: i32) -> (i32, i32) {
    println!("Inside function {x}");
    (y,x)
}
```

```
fn main() {
    let i = 5;
    j = 3;
    p = &i;
    let (k, _) = sum(&i, &j);
    println!("Returned {k}");
}
fn sum(x: &i32, y: &i32) -> (i32, i32){
    println!("Inside function {x}");
    (*y, *x)
}
```

```
let mut j = 3;
let (k, _) = sum(&i, &mut j);
println!("Returned {j}");
sum(x: &mut i32, y: &mut i32) -> (i32,
{ println!("Inside function {x}");
  *y = 4;
  (*y, *x)
```

(*y, *x) → y = &(*y+1); ⇒ Error (by default immutable)



If you send reference back, you need to specify "Lifetime"

13/3/24

GENERICS :

reduces repetition that developer has to do.

"Queue" used in OS - diff. implementation for diff. cases

↳ Though they perform same operation but operate on different kind of data

Generics can be applied on classes/ subroutine

Generic Subroutine - doesn't depend on type of arguments in Subroutine

(OOPS - Polymorphism ⇒ Diff. function with same name)

Generics - Only one function but mul. diff types of data going into function.

```
sort (int arr[], int len){
    ==========
    ==========
}
```

```
sort (T arr[], int len){
    ==========
    ==========
}
```

T ← Type not specifying what type. when we call we specify.

Necessity of Subroutine - Same code being used at diff. places

Generics ⇒ C++, Java, Cshark

```
int num[100];
sort (num, 100);
char st[100];
sort (st, 100);
```

template <T>;

Creates 2 copies.

Specific copy is returned when it is called

Type Checking could be done. ⇐ All done by Compiler in C++.

Java doesn't create Copies (instances) of the same function for different datatypes.

```
public static <T extends comparable<T>> void Sort (T arr[]);
```

↳ Elements are comparable to each other.

All calls of subroutine uses same code.

C++ is the ~~most Generic one.....~~ [Fearful Errors - may not give error but do wrong thing]

```
char lib[2][3] = {"ABC", "DEF"};
sort (lib, 2)
```

Pointers are also Comparable ⇒ Compares pointers, gives no error but does wrong Computation (Logical Error)

RUST also allows you to have "Generics".

```
fn largest <T : std :: cmp :: PartialOrd> (list: &[T]) → &T {
```

EXCEPTIONS:

- call special functions. - Stop call stack, return directly to ma
Runtime environment handles exceptions function where it is handled.
Java – long list : depth at which exception has occured.
Python ⇒ try:
≡
except e :

20/3/24

Global variables, Global Constant.

Every function returns something called "Error Value".

If Error value = 0 then operation done successfully.

If Error Value ≠ 0 then operation is not done & returned prematurely.

Nesting – Lowest function detects error & gives back error.

↳ At some point, you can't track Nesting depth

PL1 – First prog. lang. to come up with Exceptions.

ON OVERFLOW (Exception)
BEGIN
≡
END

This function will be called whenever overflow occurs in any other function.
If exception happens, OVERFLOW is executed.

C:

```
try{
≡
}catch{
≡
}
```

Some exceptions are already defined in the language.
(End of file exception in Java).

For Multiple Exceptions in a try block, we can write "Multiple catch" ie corresponding to a try block.

```
try{
≡
} catch(io_error e){

} catch (end_of_file e){

}
```

Unhandled exception gets propagated to Calling f
Runtime Environment can handle any exception which are not handled.

```
main(){
try{
fun();
}
catch(div by zero e){

}
```

C++ catch(...){ }

Python finally :

} Defects any exception if not provided specifically.

C++ : You have to handle every exception o/w Prog. will crash (stops)

Exceptions in Java

- Checked : Runtime Env. will handle if you not handle exception (specify)
- Unchecked : Prog. stops if you not handle exception. (specify)

RUST has 2 ways of handling Errors

- Panic
- Result (Enumerated type which has 2 values OK<T> and Error<E>)

```
fn main() {
    panic! ("crashed");
}
```

```
let arr = vec! [1, 2, 3, 4];
let t = arr[30];
```

Called by runtime

```
use std::fs::File;
fn main() {
    let greeting_file_result = File::open("hello.txt");
```

3/4/24

OBJECT-ORIENTED FEATURES

We need to reuse lot of code many times.

Not only functions but data is also reused. (variables...)

In some instances, data operations are also required (not only data).

Data items + Some functions

[Q] How to combine data & functions together in a single entity.?

ans: "Modules"

You may need to restrict access to some function/data.

Some lang. came up with "Object" which provides facilities (above)

Eg: C++, JAVA, C#, Python.

Class - Template which defines things which are in entity.

```
class Student {
        int ID;
        int prog;
        int discipline;
        int dept;
        calc-cgpa();
        calc-sgpa();
}
```

Class itself is just a template.

Student S1; → object

We can access member functions and data by "."

Object is an instance of a class.

Use of class/object is to offer "Abstraction".

C++ - 3 ways of restricting access to member variables

- Public - can be accessed by anyone. (Member variables of object). if S1 is
- Private - Not accessible (function outside is not accessible but inside the class, it
- Protected -

calc-cgpa ( ) {..., ID};

Member variables are private. You can access member variables through Public functions

getter( ) - gets value of member variable.

setter( ) - sets value of member variable

```
get() {
    return ID;
}
```

i = S1.id

Derived Class: Base Class

class ugstudent : student { }

C++ - specify how to derive that class

class ugstudent : public student { }

↳ follows "You can't decrease restriction but only increase restriction".

Protected members can be accessed in Derived Class.

Java - (Protected) - within same module/package can use it.

Copying function is wastage of space.

→ For every object, you have copy of member variables and reference to member functions.

How to initialize a member variable? Using Constructor. with same class name

```
class ugstudent : student {
    ugstudent( ){ }
}
```

Newer lang. uses references as variables.

Student S1 = student(); [Python] newstudent(); [Java]

5/4/24

OOPS provide an abstraction layer of variables, functions, data into object.

Also, abstracts out implementation of functions.

Apart from Inheritance, you also have Sub-Class.

```
class outer{
    class inner{ ⇒ Class inner is accessible only inside Class outer.
    }
    inner i; ⟹ can be accessed in inner class only if variable is public
}
```

Class outer has private variable

↳ In Inner class, → Java can access any variable of OuterClass

(within Scope is accessible)

Constructor is automatically called when you create an object and generally have same name as the class. You can have mul. constructo of same class. (constructors are distinguished by unique signature (# o parameters)).

```
class stack{
    stack(){
    }
    stack(int n){
    }
}
```

Different Signature

In Java/Python, Constructor is to be called explicitly.

stack s = stack();

If only this is written, it just creates reference but not object itself.

Python/Java doesn't have Destructors because there is Garbage Collecter

Destructor ⇒ ~stack(){} ⇒ final release of memory.

First initialize Base class & then Derived class constructor is called by Default

You can change the order in some lang. by using "super();"

~~Polymorphism~~: Compiler chooses one among mul. constructors → Overloading.

```
class student { }
class ugstu : student
class pgstu : student

student s;
ugstu u;
pgstu p;
```

u.calc_cgpa → uses function of that class only.

There are some cases where you need Interchangeability.

```
student *sp;
sp = &u;
sp = &p;
```

(Assign addr. of Der. class to ptr of Base class) valid in C++, Java

sp→calc_sgpa(); ⇒ Uses that function of which obj. is being referenced

Java ⇒ Any class's ref variable can be assigned to "object o"

Compiler - dynamic binding of corresponding call instr.

```
for(.....){
    sp=s[i];
    sp→calc_sgpa();
}
```

Derivation of Class - Inheritence.

Python allows you to do OOPS.

RUST has no object-orientedness.

Abstraction
Inheritance (Derivation)
Overloading
Polymorphism

classmate
Date
Page

```
class Node:
    self.i = 0
    add_node(j):
```

6/4/24

```
class (name):
    /* stmts */
    i=1234
    def print():
```

Some problems arises.

```
class Test:
    i=1234
    def printTest(self):
        print("Hello")
t = Test()
t.printTest()
```

In Python, all members are public by default.

"self" is automatically passed and represents current object's scope. similar to "this" in C++, Java

```
    i="world"
    def printTest(self):
        print("Hello"+i)
```

ERROR ⇒ i is not defined. (even though i is public)

→ i is not in scope of this function.

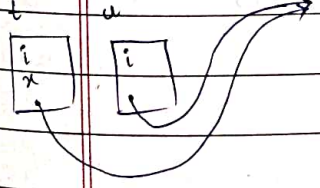
```
i="world"
def printTest(self):
    print("Hello"+i)
i="everyone".
```

Var. of obj. are very specific to that object.

```
def printTest(self):
    print("Hello"+self.i)
i="world"
def printTest(self,j):
    print("Hello"+j)
t=Test();
t.printTest(t.i).
```

~~print(id(j))~~

~~print(id(t.i))~~




```
t=Test()        u=Test()
t.x=1234
t.printTest(t.i)
print(t.x).
print(u.x)
```

```
class Test :
    i = "world".  ——→ Class Variable.
(Constructor) ——→ def __init__ (self) :
No Destructor       i = "initialized"  ~~ variable i is newly created (this is Local Scope)
in Python       def printTest (self, j) :  → Instance Variable
                    print (id(j))
                    print ("Hello" + j).

t = Test ( )
t.x = 1234
t.printTest (t.i).        Helloworld

If you write self.i = "initialized" then o/p will be Helloinitialized.

t.i = "print".
                def printTest ( self, j) :
                        print ("Hello" + self.i)
t.printTest (t.i).

Put "__" to make variable private.

    __i = "world"
    def __init__ (self) :                    [2/more] name [0/1 -]
        self.__i = "initialized"
    def printTest (self, j) :                  __i__ ✓
        print ("Hello" + self.__i)
print (t.__i)  ERROR.

Inheritance :

    class BaseTest :                    Baseclass is derived as "Public".
        testvar = "OK".
    class Test (BaseTest) :             Class Test ( BaseTest, ...., .... ):
        __i = world
```