UNIT - II

Object Oriented Programming

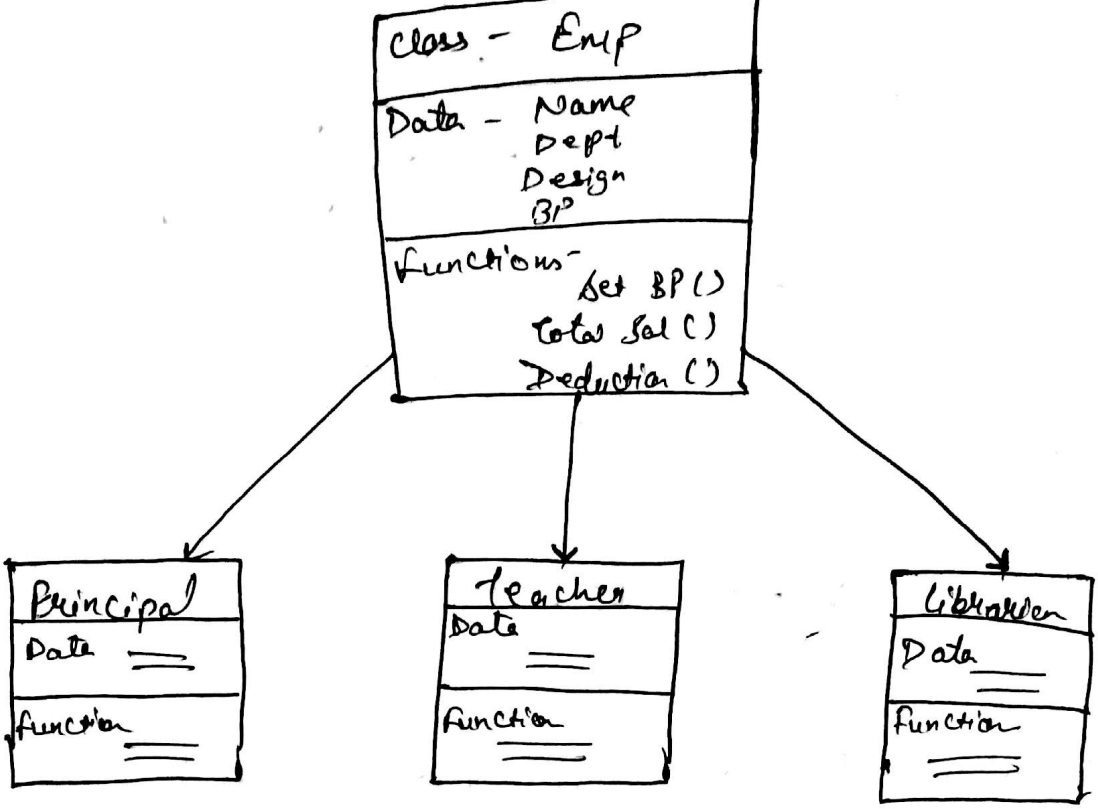
Object: Object can be any item, place, person or any other entity or activity. All objects has following characteristics.—

1) **Identity**: It is the name associated with an object which helps in identifying the object

2) **Attribute or Data**: An object can be comprise of certain characteristics or properties.
eg) height, length, colour, volume, weight, etc.

3) **Functions**: what object does or what it is capable of doing.
eg) a person can sit, stand, walk, talk, etc.

Class: Object belonging to same group is called a class. Classes represent real world objects which have characteristics & associated behaviour. Thus, it is group of similar objects.



Difference b/w OOPs & procedural programming.

OOPS	Procedural.
• Emphasis on object	• Emphasis on function
• Once class is defined objects can be created.	• Emphasis is on solving problem through function
• It is bottom-up approach.	• It is top-down approach
• Data hiding feature prevents accidental change in data.	• Presence of global variable increase the change of accidental change in data
• Features like data encapsulation, polymorphism, inheritance, are the key features.	• These features are not present.
• It break downs a task into objects where each object encapsulates its data & function	• It break downs a task into collection of variable data-structures & sub-routines.

- Suited for large programs
- eg) C++, Java, Python

- Suited for small programs
- eg) C, Pascal

- **Class declaration**

while declaring a class, four attributes are declared -

1) Data Member
2) Member function
3) Program access level
4) class tag-name.

Syntax:

```
class <name>
{
    public: variable declaration;
            function declaration;
};
```

- **Function declaration:**

Syntax:

```
type class_name::function_name(parameter list)
{
    function body
}
```

where

type: data type of value to be returned.
Class_name: name of class to which fun belongs
function_name = name of f^n to be declared.
Parameter list = list of formal arguments.

eg)

```
int Maths::Sum(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

- **Object Creation**

Object is an instance of Class and it can be created of its class type by using the Class-name.

Syntax: Class-name Object-name;

eg)

```
void main ()
{
    math temp;
    int a,b,c,r;
    clrscr();
    cout << "Enter a and b";
    cin >> a >> b;
    r = temp.sum(a,b);
    cout << "Sum=" << r << "\n";
    getch();
}
```

For the following problem:

SWAP in C++, define a class teacher with following specifications, name :- 20 char, subject 10 char, basic DA, HRA, float type, Salary- float type. Calculate f:: Computes the salary and returns it. Salary is sum of basic DA & HRA. Read Data function accepts data value and invokes Calculate function. Display() function prints data on screen.

Sol

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
class teacher
{
```

```
public : float calculat();
         void readdata();
         void display();
         float basic, da, hra, Salary;
         char name[20], subject[10]; };
float teacher::Calculate()
{
    Salary = basic + da + hra;
    return Salary;
}
void teacher::readdata()
{
    cout<<"\n Enter name of teacher:";
    gets(name);
    cout<<"\n Enter subject:";
    gets(subject);
    cout<<" Enter basic, da, hra:";
    cin>>basic>>da>>hra;
Salary=calculate();
}
void teacher::display()
{
    cout<<" Name:";
    puts(name);
    cout<<" Subject:";
    puts(subject);
    cout<<"Salary:"<<Salary<<"Basic:"<<basic
        <<"DA"<<da<<"HRA:"<<hra;
}
void main()
{
    clrscr();
```

```
    teacher t;
    t.readdata();
    t.display();
    getch();
}
```

Inline function:

Member function can be defined inside the class & outside the class also. If function is small then it can be defined inside the class and is called as inline function.

eg)

```
class x
{
    public: void read()
            { cin>>a; }
            void print()
            { cout<<a; }
};
```

Encapsulation:

It is a way of combining data and function into an independent ~~identity~~ entity ~~else~~ called class.

Wrapping up of data & function into a single unit called class is known as encapsulation.

eg) class teacher
{
public : char name[20], sub[10];
char desig[10], dept[10];
float basic, DA, HRA;
float sal();
float total();
float deduction();
};

Abstraction :

Variables & functions declared under keyword "public" are accessible outside the class in the program. Since, public fns of a class can be invoked from anywhere in the program, we can say that various objects in a program can talk to each other through these functions. Thus activity of seperating essential function of an object in the form of public interface is known as abstraction.

Data hiding : Information hiding

Access to private part of an object is restricted in the sense that functions of the object can access this data. Thus private parts of object are not available and can not be altered by external changes. This property of object is called as information hiding or data hiding

Data abstraction is representing essential features without including background details. It determines what is & what not important.
Goals of abstraction is

* User do not know and should not know how object is implement.
* User should not know how complex and simple the program is.
* User can only use the method and function which provide necessary interface to perform.
* Thus, activity of seperating essential function of an object in the form of public interface is called abstraction.

eg) while driving a car we know the essential features like Gear handling, Steering, Clutch and brakes but while driving we get into internal details like wiring, motor etc. what is happening inside is hidden from us. This is abstraction where we know the essential thing to drive a Car without including background details or explaination.

Public Access Specifier:

Public data can be accessed outside the class, through non-member using function

Private Access Specifier:

Private data of a class can be accessed only through member function of class in which it is declared. Any member declared in a class before an access specifier are private be default.

```
class exarray
{
    private: int arr[10];
    public : int largest();
             int sum();
};
```

arr[10] is private data member which can be accessed only through member functions. largest() and sum();

Protected Access Specifier:

Protected members are the members that can be used only by member functions and friends of the class in which it is declared.

NOTE: Protected members are inheritable but private members are not inheritable.

```
class ABC
{
    int x, y;
    int a;
    int sqr(int a)
    { return a*a;
    }
```

```
public: int z;
        int add(int a, int b)
        {
            int c = a+b;
            return c;
        }
};
void main()
{
    ABC temp;
    temp.z = 10;        //valid.
    temp.x = 5;         //invalid (private)
    temp.sqr(10);       //invalid(private)
```

Polymorphism :

Polymorphism has been derived from a greek word poly and morpha. Poly means many and morpha means shape. In object oriented programming Polymorphism means one name with multiple behaviour or different behaviours. function

eg for a add function.

```
float area(float a)
{
    return (a*a);
}
```

Types of polymorphism

1) Compile time — function overloading, operator overloading
2) Run time — overriding by virtual function.

1) Function Overloading:

When several function declarations of same name are defined with different type of parameters or different no. of parameters then function is said to be overloaded. This feature is called function overloading.

ways to overload a function

1) **By changing no. of arguments:**

In this type of overloading, we define two function with same name but different no. of arguments.

```
int sum(int x, int y)
{
    cout << x+y;
}
int sum (int x, int y, int z)
{
    cout << x+y+z;
}
int main ()
{
    sum(10,20);
    sum(10,20,30);
}
```

2) **By using different types of arguments:**

In this type of overloading, we define two or more functions with same name and same no. of parameters but type of parameters is different.

```
int sum(int x, int y)
{
    cout << x+y;
}
double sum(float x, float y)
{
    cout << x+y;
}
int main()
{
    sum(10,20);
    sum(10.5,20.6);
}
```

(3) By using default arguments

In this, we make a call to the function without passing any value for that parameter, the function will take the default value specif

```
sum(int x, int y=0)
{
    cout << x+y;
}
int main()
{
    sum(10);
}
```

Rules for overloading :

(1) Each overloaded function ~~for~~ must differ either by no. of its formal parameters or their data

② Return type of overloaded function may or may not be same.

③ Default arguments of overloaded function are not considered by C++ compiler as part of parameter list.

④ Do not use same function name for two unrelated function.

Advantages of overloading

① Program becomes easier to read.

② Programmer does not waste time in searching new name for similar function.

③ Programmer can devote more time in logic development and need not remember different function names.

9) Operator Overloading

Step-1: Create a class

Step-2: declare its operator function in public part of class.

Step-3: Define operator function to implement the required operation through objects.

```
Class math
{
    public: int x, y;
            int sum();
};
math :: sum(int x, int y)
```

```
{
    cout<<x+y;
}
int main()
{
    math temp;
    temp.sum(2,3);
}
```

~~Operator overloading~~

Run-time polymorphism :

(overriding through virtual functions)

It is achieved via pointers and virtual fu.

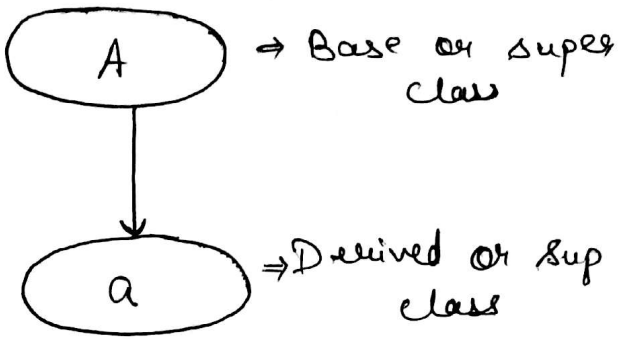
Base class pointer is used to point to an object of any class derived from that base. When a base pointer points to a derived objects that contains a virtual function, C++ determines which version of that function to call based upon type of object pointed to by the pointer. And this determination is made at run time

```
Class base
{
    public: virtual void disp();
            {
                cout<<" Good Day";
            }
};
```

```
class derived : public base
{
    public : void disp()
             {
                 cout<<"Hello";
             }
};
int main()
{
    base *p, b;
    derived d1;
    p = &b;
    p->disp();
    p = &d1;
    p->disp();
    return 0;
}
```

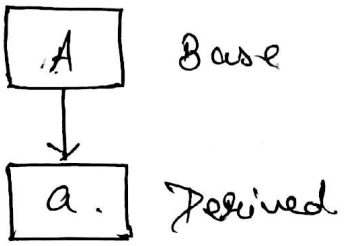
Inheritance:

Mechanism of deriving a new class from an old one is called inheritance. It helps in reusibility of the code. Old class is called base class and new one is called derived class or sub class. Derived class inherits some or all traits from base class. A class can also inherit properties from more than one class or from more than one level.

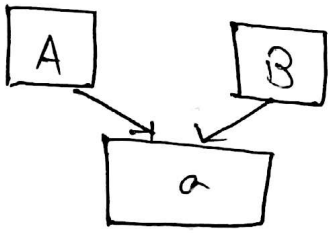


Types of Inheritance :

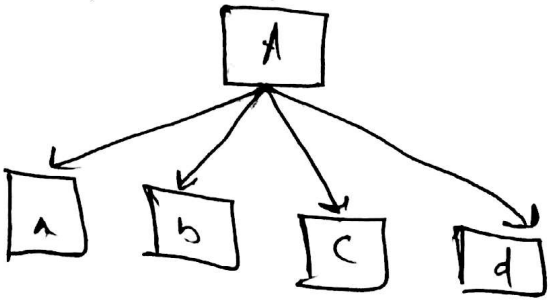
1) Single (1 level) inheritance : when a derived class inherits from only one base class, it is known as single level inheritance.



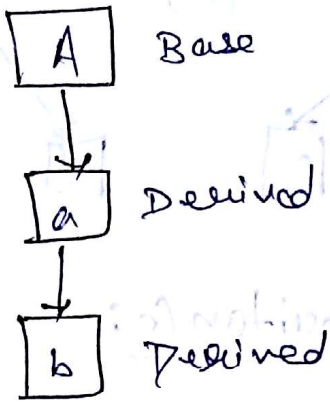
2) Multiple inheritance : when derived class inherits from multiple base classes, it is known as multiple inheritance.



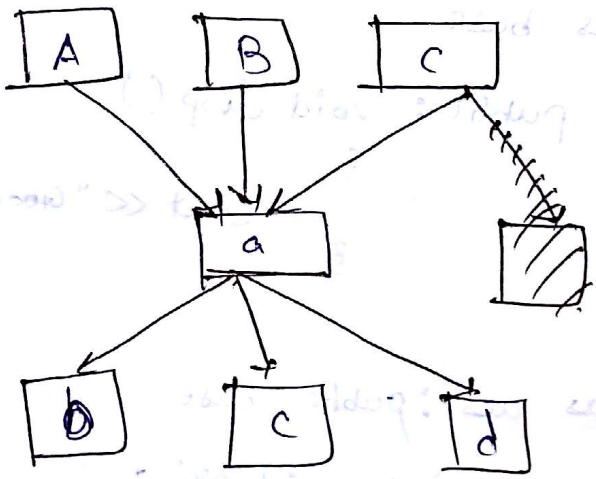
3) Hierarchial Inheritance : when many derived class inherits from single base class, it is known as hierarchial inheritance



4) Multilevel inheritance: It represents a transitive name. Mechanism of deriving a class from another derived class is known as multilevel inheritance.

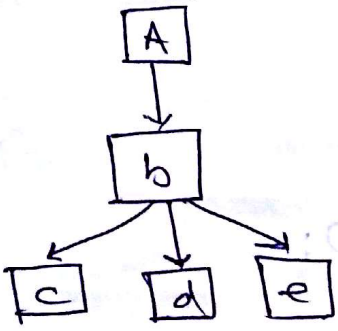


5) Hybrid Inheritance: It combines two or more forms of inheritance. When a derived class inherits from multiple base classes and all of its base classes inherits from a single base class. It forms hybrid inheritance.

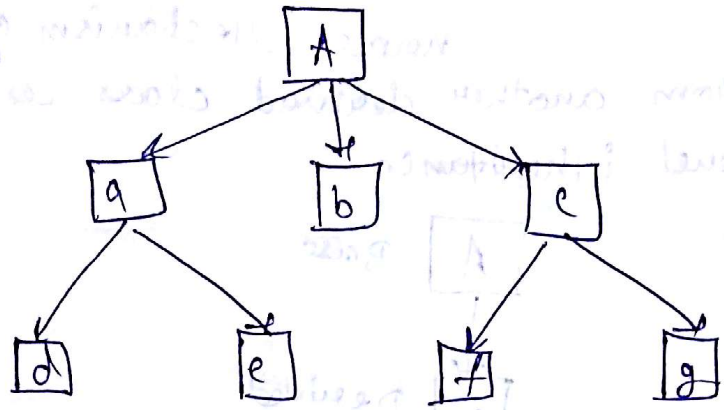


Q. Example of Multiple or Multilevel.

Sol



Hierarchical & Multilevel



Syntax of inheritance:

```
class derived class name : visibility_mode baseclassname
{
    ---------
    ---------
    ---------
};
```

eg)

```
class base
{
    public: void disp()
            {
                cout << "Good";
            }
};
class des :public base
{
    public: void print()
            {
                cout << "Day";
            }
};
void main()
{
    clrscr();
    des d;
```

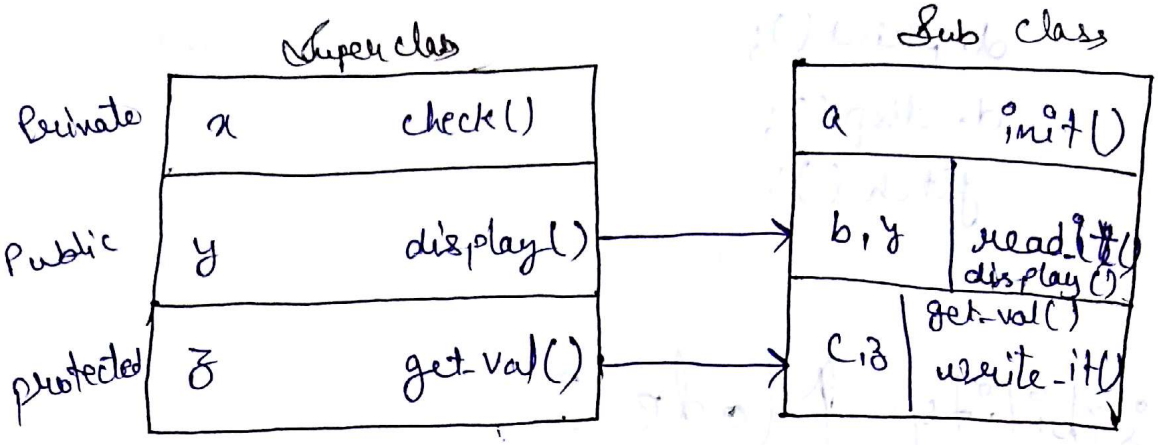
```
    d.print();
    d.disp();
    getch();
}
```

Visibility Mode :

1. **Public** : Public derivation means that the derived class can access the public & protected members of base class but not the private members of base class. Thus, public members of base class become the public member of derived class & protected members of base class become the protected members of derived class.

eg)

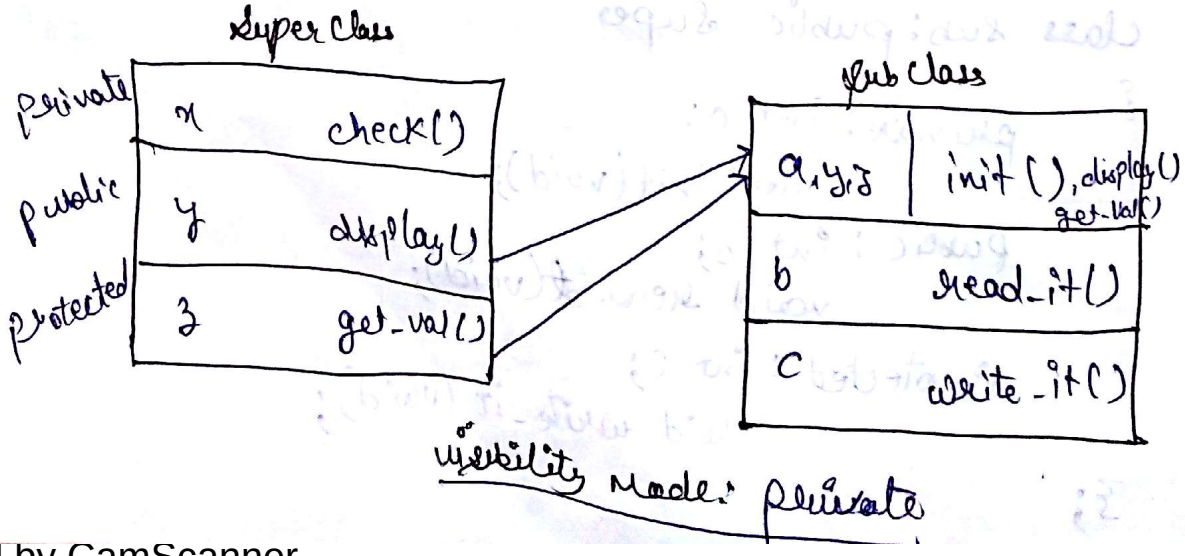
```
class super
{
    private : int x;
              void check (void);
    public : int y;
             void display (void);
    protected : int z;
                void get_val (void);
};
class sub : public super
{
    private : int a;
              void init (void);
    public : int b;
             void read_it (void);
    protected : int c;
                void write_it (void);
};
```



Visibility Mode: public

2. Private : Private derivation means derived class access Public and protected members of base class privately. Thus with private derived class the public & protected members of base class become private members of derived class.

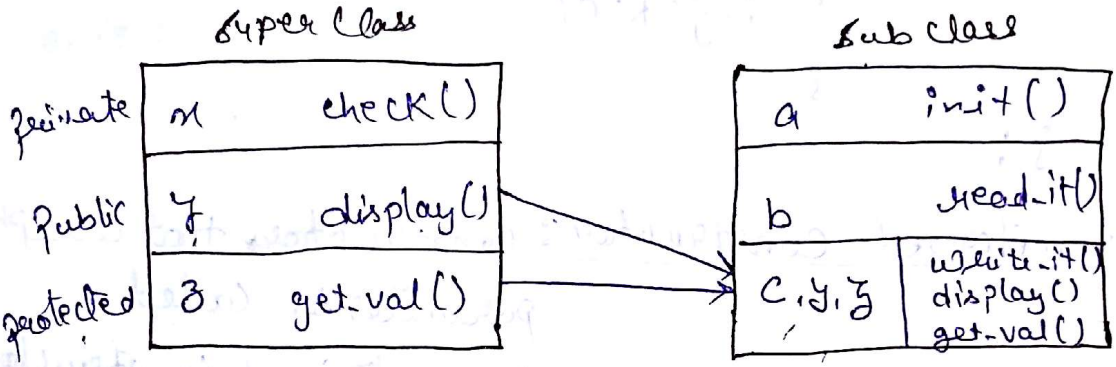
eg)
```
class Super
{
    ———
    ———
    ———
};
Class Sub : private Super
{
    ———
    ———
    ———
};
```



Visibility Mode: private

3. **protected**: the derived class can access the public and protected members of base class protectedly. Thus with protected derived class the public & protected members of base class become protected members of derived class.

eg)
```
class super
{
    ========
};
class sub: protected super
{
    ========
};
```



visibility mode: protected

Constructor:

Constructor is member function of a class that is automatically called when an object is created of that class. It has same name as that of class name and its primary job is to initialize the object to a legal initial value for the class. If a class has a Constructor, each object of that class will be initialized before any use is made of the object. Thus, it is a function that allocates memory space for data items or state variables of each object.

Types of Constructor

1) Default Constructor: It is used in situation where we initialize the data elements before they are used. They can be defined inside & outside the function.

eg)
```
class X
{
    private : int i;
    public : int j; int K;
             X ()
             {
                 i=j=K=0;
             }
};
```

2) Parameterized Constructor: Constructors that accept parameter is called parameterized constructor.

3) Copy Constructor: It is a constructor of the form Class name (class name &).

Compiler will use copy constructor whenever you initialize an instance using value of another instance of same type.

eg)
```
class Sample
{
    private : int i,j;
    public : Sample (int a, int b)
             {
                 i=a, j=b;
             }
```

```
Sample(Sample &)
{
    j = s.j;
    i = s.j;
    Cout << "Copy Constructor";
}
};
```

Destructor:

It removes the memory of an object which was allocated by constructor during creation of an object. It carries the same name as class and has a 'tile' sign with it.

eg)
```
class XYZ
{
    public: XYZ();
            ~XYZ();
};
```

Advantage of OOPS

1) Revisibility of Code (Inheritance)
2) Ease of comprehension (polymorphism)
3) Ease of maintainence (Encapsulation & Abstraction)
4) Redesigning & extension

Functional Programming :

F.P Programming is an expression based programming in which everything is based on mathematical function. It emphasises on what a function should do rather than how it should do.

WAP in C to Calculate Sum of n numbers.

```
Sum ()
{   int n, i, s=0;
    for(int i=0 ; i<=n; i++)
        s+ = i;
}
```

Key features of functional programming

i) Programs are constructed as composition of f^n.
ii) It is highly abstract in nature
iii) Code is highly reliable as it has no side effects.
iv) Imparts simplicity since code is shorter
v) Programs are written quickly & easy to verify.
vi) Can run in multi-processor architecture

eg) +(*WX)(-XZ)

the multiplication & substraction can be evaluated in parallel and then return to "+"

LISP programming:

1) LIST: It is sequence of atoms or other lists seperated by blanks & enclosed in paranthesis.

eg (1 2 3 4) → list.
(1 — atoms)

* Numeric Arithmetic:

Addition	>(+ 5 10)
Subtraction	>(- 10 5)
Multiplication	>(* 5 10)
Division	>(/ 10 5)
Absolute	>abs(-7)

* Transidental function:

power	>(exp 3 5)
Squareroot	>(sqrt 144)
cosine	>(cos(3.14))

* Relational predicate:

equality	>(= 8 9)
less than equal to	>(<= 8 9)
greater than	>(> 8 9)
maximum	>max(8 9 3 4)
minimum	>min(8 9 3 4)

List Manipulation:

1) It is a **CAR** returns first element of a non-null list

eg) car(1 2 3) it returns 1.

2) **CDR:** It returns element of list except the first element

eg) cdr(1 2 3) it returns 2.3.

Q: Calculate length of a list

Q: Search an element in the list.

Q: reverse of a list.

3) **Cons:** It joins the two lists.

eg) cons(a(bcd)).

It returns "a b c d."

4) **Append:** It joins the two strings

eg) append("taj"("is beautiful"))

It returns "taj is beautiful"

5) **Reverse:** >reverse(1 2 3 4)

4 3 2 1

6) **length:** >length(1 2 3 4)

4

7) **Comment:** > ;;

Input - Output Statement

> (+ 3) // Input 3 Values)

10
15
25

* Prints the argument as it is received and then returns the argument.

b. > print (a b c)

a b c

> princ (a b c)

a
b
c

> terpri

new line.

Q) Calculate the hypotenuse of a triangle.

Sol

> print (Sqrt (+ (* a a) (* b b)))

$$s = \frac{a+b+c}{2}, \quad \text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

> / (+ (+ a b) c) 2

* (/ (+ (+ a b) c) 2) (- (/ (+ (+ a b) c) 2) a)) (- (/ (+ (+ a b)

c) 2) b)) (- (/ (+ (+ a b) c) 2) c)))

Difference b/w procedural & functional pr

Procedural	Functional
① has lower order functio	① Has high order func
② has side effects.	② Avoid side effects
③ It is not pure in nature	③ It pure in nature
④ It does not ensure referential transparency	④ Ensure referential transparency.

High order function:

Functions are high order when they take other functions as arguments & return them as results.

High order function enable currying, a technique in which a fn is applied to its argument one at a time with each application returning a new function that accepts the next argument.

$$fn\ x \Rightarrow fn\ y \Rightarrow x$$

iii) Strict & non-strict evaluation

LISP follows strict evaluation where as Haskell, Miranda follows non-strict evaluation.

⇒ In strict evaluation, argument to a function are evaluated before function call

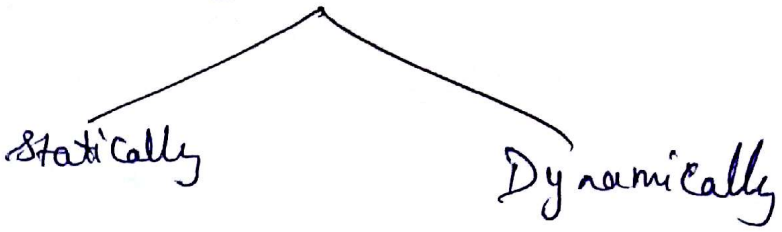
⇒ In lazy evaluation, argument are passed to a function are unevaluated and calling function determines when arguments are to be evaluated.

Type System:

It prevents run time error like exception handling in java. There are two reasons of type system

— ① Program safely
— ② Program efficiency

Type System



Statically: If an error is detected before a program start to execute then symantics of failure need not be tealt with.

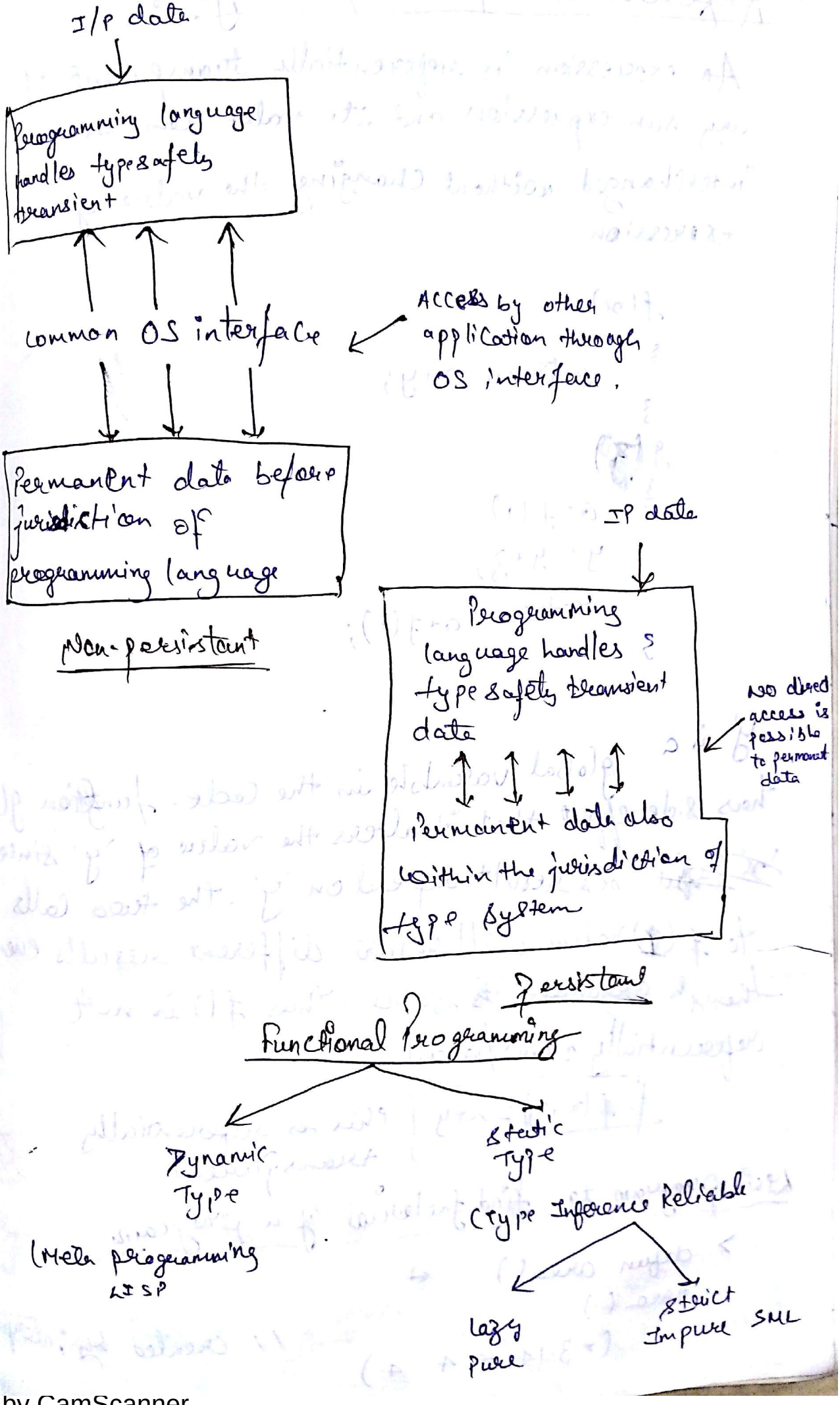
If the absence of typ errors can be guaranteed before execution beggins then there is no need to terminate the execution when type error occurs

Dynamic: Progeramming systems which deals with large amount of hetrogenous data require dynamic type checking

Type System environment

(1) **Non Persistant:** the data which persist for longer than the invocation of progeram is achieved by operating system interface.

(2) **Persistant:** Storage of data beyond a single progeram involation is handled by progeramming language mechanism and no common operating system interface is necessary. The only route by which data may be accessed is through progeramming language.



* Referential Transparency :

An expression is referentially transparent if any sub-expression and its value can be interchanged without changing the value of expression

```
f(x)
{   return x+y;
}
g()
{   a=f(1)
    y = y+3;
    return a+f(1);
}
```

'y' is a global variable in the code. function g() has side effect that it alters the value of 'y' since x's result depend on 'y'. The two calls to f(1) value will return different results even though argument is same. Thus f() is not referentially transparent

`f(x,y)=x+y` this is referentially transparent.

Lisp program to find factorial of a program

```
> defun area()
Area()
    (* 3.14 (* 4 4)
```

// created by ; interp

Q calculate area of circle by taking the value of radius from user.

```
defun area()
→ AREA()
(print "Enter radius"))
→ Enter radius
(setq radius (read))
→ _                    // Enter value
(*3.14 (* radius radius))
```

Q WAP in LISP to convert miles into Kilometer,
5 miles = 8 Kilometer.

```
defun convert()
→ CONVERT()
(print "Enter distance in miles"))
→ Enter distance in miles
(setq miles (read))
→ _
(* miles (/ 8 5))
```

Q Read two integers and return sum of their squares.

Q difference.

Q Solve ackerman function

Q calculate factorial of a number

Q Take one argument & return last element of list.

1. Take an argument & return the list except the last element.
2. Return reverse of a list.
3. To sum even no. b/w 1 & 100.
4. Concatinate any two list.
5. Append a list.
6. Palindrome
7. GCD of no.
8. LCM of no.

Imperative programming

* Statements are executed step wise in sequential manner.
* Programs run much faster. Mostly the language are Compiler based.
* Although BASIC & APL are interpreter based imperative languages.

Procedure Activation (Parameter passing Mechanism)

If some program P, Q, R. all need access to a variable x for this x needs to have some association to local environment and non-local environment. This direct sharing is made by parameter transmission

① Actual Parameter: In this, it is data object ie, shared with caller Sub-program when Sub-program is called with actual parameter the expression is evaluated at the time of call before the Sub program is entered. Actual parameter may be ~~a local data object belonging to the caller~~

- a local data object belonging to the caller.
- formal parameter of the caller.
- Non local data-Object visible to the caller.
- Result invoke by a function invoke by a caller & immidiately transmit to the called Sub-program

SUB(I, B)	I & B are local variables.
SUB(27, True)	27, true both are constant.
SUB(P_1, P_2)	P_1, P_2 are formal parameter.
SUB(G_1, G_2)	G_1, G_2 are global variables
SUB(A{I}, D.B1)	A{I}, D.B1 are components of array.
SUB(I+3, FN(Q))	Result of a defined function

② Formal parameters: It is a kind of local data object within a Sub-program. It is a alias ~~to~~ to actual parameter or it may contain a copy of value of that data object.

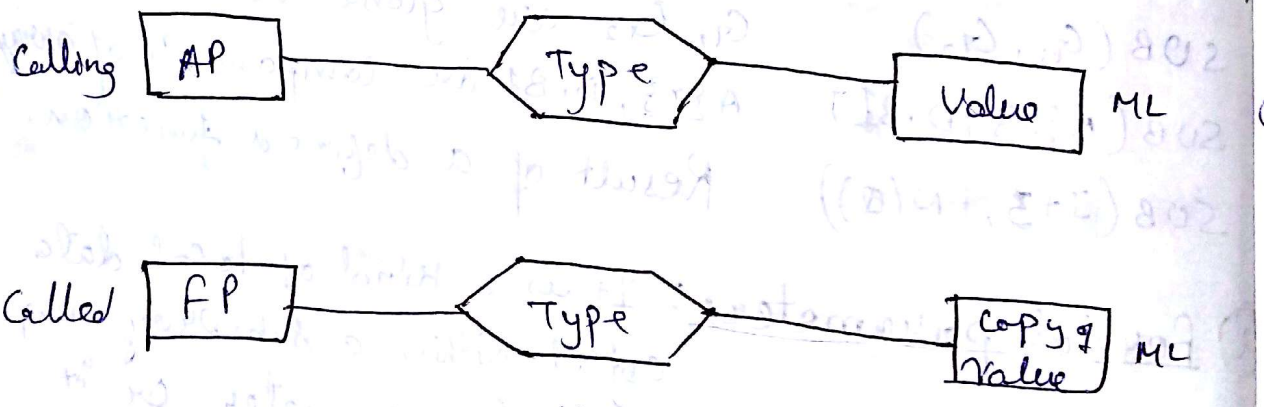
```
SUB(int x, int y)
```

/* In this, x & y are formal parameters which is name of simple identifier with its type */

Methods of Parameter Transmissions

1) Call by value
2) Call by reference
3) Call by name
4) Call by result
5) Call by value result

1) **Call by value**: In this, values are actually passed as an argument. In this, a copy goes into the procedures, values are passed as r-values. The value of actual parameter is passed to formal parameter. Once an actual parameter is passed by value, the formal parameter has no access to change the value of actual parameter.



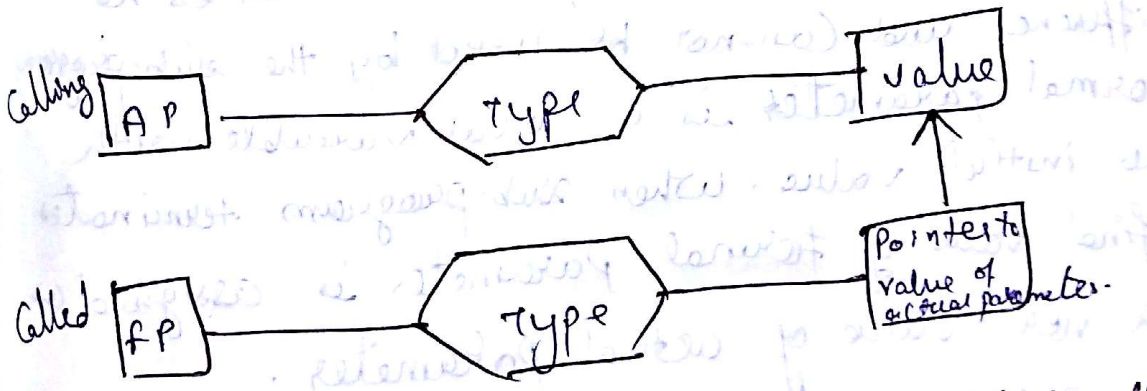
Call by value

Q. WAP to swap two numbers using call by value

```
#include<stdio.h>
#include<conio.h>
void swap (int a, int b)
void main()
{
    int a=10;
```

```
    int b=20;
    clrscr();
    swap(a,b);
    getch();
}
void swap(int x, int y)
{
    int temp;
    temp = x;
    x= y;
    y=temp;
    printf("%d and %d", x,y);
}
```

2) Call by reference : Pass a pointer to actual parameter. A pointer to the location of data object (l-value) is made available to the subprogram. Data object does not change position in memory.



Q WAP to swap two numbers using Call by reference.

```
#include<stdio.h>
#include<conio.h>
void swap(int *x, int *y);
void main()
{
    int a=10;
    int b=20;
```

```
    swap(&a, &b);
    getch();
}
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    printf("%d %d", *x, *y);
}
```

3) Call by result: Implemented in Ada language.

In this, a copy goes out of the procedure. It is used only to transmit a result back from a sub program. Initial value of actual parameter makes no difference and can not be used by the sub-program. formal parameter is a local variable with no initial value. When sub-program terminates final value of formal parameter is assigned as a new value of actual parameter.

```
$ c: array [1 ... 10] of integer
    m, n: integer;
    Procedure r (k=2; j=3; int)
    begin
        k=k+1;
        j=j+2;
    end r
        m = 5
        n = 3   r(m,n)
```

r. write m, n;

}

Output

3, 5

4) Call by value result : Implemented in fortran

Copy going in and going out. In this formal parameter is a local variable of same data type as actual parameter. Value of actual parameter is copied to formal paramter at the time of call when sub-program terminates the final contents of formal parameters of data object are copied into actual parameters

eg {

```
c: array[1....10] of integer
m,n : integer;
procedure r(K; j:int)
begin
    K=K+1;
    j = j+2;
end r
    m= 5
    n = 3
    r(m, n)
    write to M, n;
```

}

5) Call by name : Implemented in Algol 60. It is complex and inefficient.

Argument values replace the corresponding

parameters names in function body and then the body is executed.