Unit 3 + unit 4

OOPS concept

Unit - IV Object oriented Programming (OOPs)

{C++, Java}

object oriented programming language is paradigm that provide many concepts ⇒ obj. to design many application & computi programs

The programing paradigm where everything is represented as object is clled object oriented programing language.

The main aim of oops is to bind the data and different types of function that operate on them. so that no other part of the call can access that data except that function. following are the features of OOPs :—

* Object → An entity that has state & behaviour is known as an object it can be a physical or logical

* Class → It is a collection of objects and it is logical entity ie a user defined data types which called its data member & member functions

```
class <class name>
{
    Data
    function
};
```

main()
{
class name object name1, object name2 ... ;
}

for eg: There may be many parts with different names & branch but all of them will share some common properties like 4wheels speed limit, mileage, range, etc.

INHERITENCE → when one object acquire all the properties & behaviour of parent object is known as Inheritence

The capability of class to derive the properties from another class. It provide the code reusability & it is used to achive runtime polymorphism

Polymorphism → It means having many forms and has the ability of msg to be display in more than 1 form.

Abstraction → Hiding the internal details & showing the functionality is known as abstraction in C++ we use abstract class & interface to achive an abstraction.
It refer to providing the essential information about data to the outside world and hiding the background detail or implementation

Encaptulation → the wrapping & binding of code & data into a single unit is known as encaptulation

Access Modifiers in C++ →
access modifiers is used to identify access right for the data, & member function of the class.

there are three main types of access modifier or access specifier in C++ programming language

1) PRIVATE → The private member is inaccessible from outside the class, means only member of same class are allow to access

2) PUBLIC → Everyone is allow to access
3) PROTECTED → It is a stage b/w private & public access if member function define in class are protected they can not be accessed from outside of class but can be access from derive class

Access Modifiers in C++
1. C++ Data types → char, int, float, bool
2. Derived data types → array, function, pointer
3. User or abstract data types → class, structure, union, enumeration, typedef

Function Overloading → It is defined as having two or more function with the same name but different parameters. the function is redefined by using either different types of argument or different number of arguments.

* the advantage of function overloading is
1) increase redability of program bcoz you do not need to use different names for the same action

```
#include<iostream>

using namespace std;
class cal
{
    public;
    static int add (int a, int b)
    {
        return {a+b};
    }
    static int add (int a, int b, int c)
    {
        return a+b+c;
    }
};

int main (void)
{
    cal c;
    cout << c.add (10,20) << endl;
    cout << c.add (10,20,25);}
    return 0;}
```

22nd - drupal

Operator overloading in C++

In C++ we can make operator to work for user defined classes. C++ has the ability to provide the operators with a special meaning for a data type this ability is known as operator overloading An operator is overloaded to give user defined meaning to it

```
#include<iostream>
using namespace std;
class Test
{
    Private: int count;
    Public
            Test();
            count(I)
            {
            }
    void operator ++ ()
    { count = count +1; }
     void display()
     { cout<<"count;"<<count; }
};

int main()
{ test t;
    ++t;
    t.display();
    return 0;
}
```

Cout<< 5
3<< 8

	In C++	In C
allocation	new	malloc
Deallocation	Delete	Calloc

Friend Function → The friend function of a class is define outside that class scope but it has the right to access all private & protected member of the class, even though the prototype for friend function appear in the class defination.

The advantage of friend function is do not violate encapsulation bcoz the class itself decide which function are it's friend function.

Function Overriding → It is a feature that allow us to help us in function in child class which is already present in parent class. A child class inherit the data members & member function of parent class. but when you want to override a functionality in the child class then you can use function overriding.

Constructor → It is a member function of a class which initialize object of class it has the same name as of its class. constructor is automatically called when object create it is a special member function of class. It is of three types

1) Default Constructor →
It has no argument in it for eg.

```
#include <iostream>
using namespace std;
class Test
{ Public
    int a,b;
    Test ()
    { a=10
      b=20
    }
};
int main ()
{
   Test c;
cout <<"a;" << c.a << endl;
cout <<"b" << c.b <<;
}
```

// WAP for a simple calculator

```
#include <iostream>
using namespace std;
class calculator
  { float a, b, c;

  if (a++b)
     { c = a+b; }
  elif (a--b)
     { c = a-b}
  elif (a**b)
     { c = a*b}
  else (a//b)
     { c = a/b}
```

```
return c;
}
int main ( )
{printf (" the output is %d", &c);
 getch ( );
}
```

```
#include <iostream>

puts (" this is program of calculator");
puts (" enter the value of a and b");
gets ("%f %f", &a, &b);
```

2. Copy Constructor →

It is also called one argument constructor the main use of copy constructor is to initalize objects while in creation also used to copy an object. It also allow programmer to create a new object from existing one by the initialization. the constructor must be define in public & must be a public member & overloadeing of constructor is possible

DESTRUCTORS (~) → Destructors are usually used to deallocate memory and do other clean up for class obj. & its class member when the obj. is destroyed. A destructor is called for a class object when that object pass out of the scope or explicitly deleted

A destructor is a minimal function with the

same name as its class

```
class abc
{
    private
      - ———
      - — —
    public
        abc ()
        { ———
          ———
        } — —
        ~abc ()
          {-}
```

A destructor takes no argument and no ~~with~~ return type. A destructor can be declared virtual or pure virtual.

Multiple inheritance in C++ →

In C++ it is perfectly possible that a class inherit the members from more than one class this is done by simply seprating different base class with comma in derived ~~right~~ class declaration.
for eg. If we had a specific class to print on screen & wanted our class rectangle & triangle to also inherit & members in addition to those of polymer.

The direct base class is a class that appear directly as a base specifier in the

declaration of it's derived class.

An indirect base class is a base class that doesn't appear directly in the declaration of derived class, but it is available to the derived class through one of it's base class.

Virtual function & Pure virtual function →

virtual function is a member function ie declare within a base class & redefined by the derived class it is used to support runtime polymorphism. when a virtual function is called by using a base class pointer the compiler decide at runtime which version of function ie base class version or the override derived class version is to be called. It use the virtual key word.

In pure virtual function no difination in base class & all the inherit derived class has to redefine it pure virtual function is virtual function with no body.

Inline Function →

We use a keyword inline it is a powerful concept ie commonly used in the classes if a function is inline the compiler place a copy code of that function at each point where the function is called at compile time.

```
inline void func()
{
    Function body
}
```

code inserted at fn call →

```
void main()
{
    ==
    func();
    ==
}
```

Templelate in C++ →

A templelate is a powerful tool in C++ the simple idea is to pass the data type as a parameter so that we don't need to write the same code for diffrent data type.

foreg. A sw company need sort() for diffrent data types rather than writing & maintaining the multiple codes we can write one sort() and pass the data types as a parameter.

Templelates are expended at compile time this is like macros in C, the deffeence is compiler does type checking before templete expansion. the source code contain only function of class but compile the code may contain multiple copies of same function or class.

1) Function Templete →

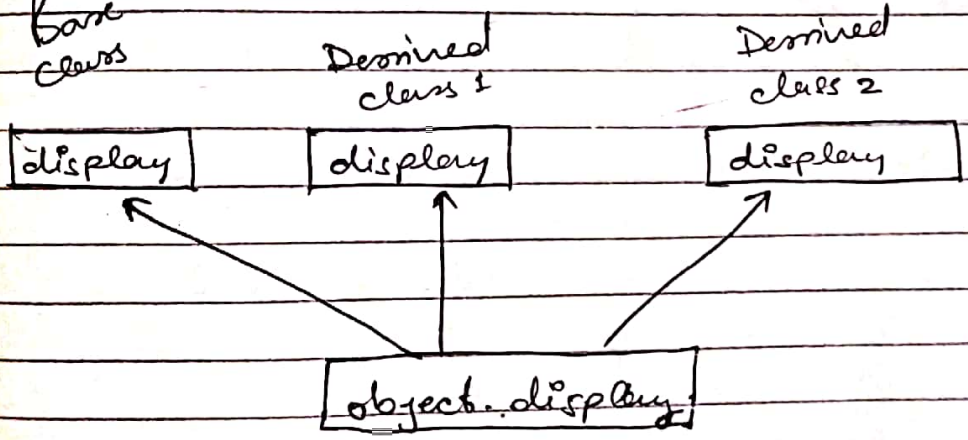
we write a generic function that can be used for diffent data types

```
# include <iostream>
    using namespace std;
    templete <class T, class U>
```

```
class A
{
    I x;
    U y;

Public:
    A()
{ cout << " constructor called " << endl; }};

int main()
{
A class A(char, char) a;
    A (int, double) b;
    return 0;
}
```



↑ Same statement different fn call

~~eg. of virtual function~~

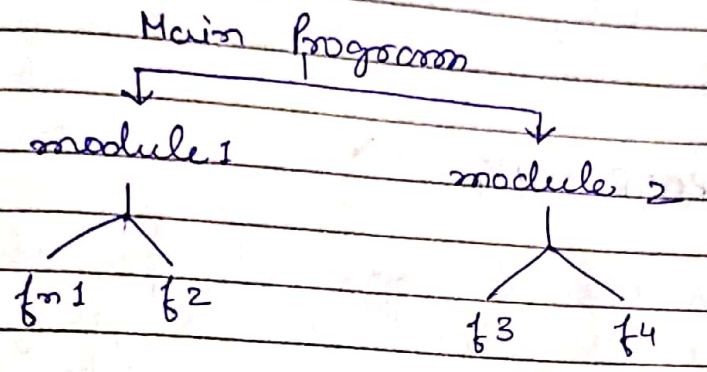
~~2) This is~~

2) Class Template → It is like function template
It is useful when a class defines something
ie independent of the data type it
can be useful for classes like linklist,
binary tree, ~~string~~, array, stack

```
template <class Type> class classname
{
    statement 1;
    statement 2;
    ---
}
```

Programs design with modulus (modular programing)

Structure programming
* selection (if | then | else)
* Repetetion (while | for)
* Subroutines (divide into parts)
* Block



Inst^n level concurrency
statement level "
Unit level "
Program level "

Program design with modulus (modular programming) → it is a sw design technique that separating the function into program in too independent, interchangeable modules such that each with everything

following are the advantage for modular programing
↳ ease of use :— This approach allows simplicity, clarity for each / every subroutine
↳ Reusability :— It allows the user to reuse the

functionality with different interface.

③ Ease of Mentinance :- It helps in less collisons at the time of working with modules. It is concern with the sub division of program into small units. these are usually arrange in a hierarchy of abstraction.

STRUCTURE PROGRAMING →

It is initiated the development of what become known as structure language. the aim of structure programming is to create the programs that are easy to understand & error free it is basically a lower level aspect of coding while modular programing is a higher level aspect.

Structure programing contains the elements like control structure, subroutines that has the procedure functions & methods & blocks that contains the groups of statements

Possibles Levels in Concurrency →

The sw execution can occur at different levels

① Instructions level →
Two or more instructions can be executed in parallel computing.

② statement level → HLL statement can (high level language)

execute simultaneously

2. Unit level →
It execute the two or more sub program simultaneously

3. Program level →
The complete program can execute to perform the desired functionality of each & every modules

Grouping of DATA & OPERATIONS →

1. Array
2. structure

Exception Handling in C++

① Try ② Throw ③ Catch

```
#include <iostream>
using namespace std;
double division (int a, int b)
if (b==0)
{ throw "Division by zero condition"; }
    return (a/b); }

int main()
{ int x=50; int y=0; double z=0;
  try
  { z = division (x,y);
    cout << z << endl;
  }
```

```
catch (const char *message)
{
    cout << message << endl;
}
    return 0;
}
```

Monitor & Semaphore

```
Monitor <monitor_name>
{
  data variables
  Procedure P1 (--)
  { - - - - - }

  Procedure P2 (- - -)
  { - - - - - - - }

  Procedure Pn (- - -)
  { - - - - - }

Initilization code (--)
  { - - - - - - - }
```

Semaphore wait operation

```
wait (s)
{ while (s<=0);
    s---;
}
```

Signal operation

```
signal (s)
{ s++; }
```

Unit IV lambda Calculus (λ)

In mathematical logic & CS λ calculus is a formal system designed to investigate function, function applications and recursion. λ calculus can be used to define what a computation function is. λ calculus has great influence functional programming language such as Lisp, ml & sql.

Following are the important properties of λ calculus

(i) It is a universal model of computation that can be used to simulate any turing machine.

(ii) λ calculus can be called the smallest universal programming language. it consist of a single transformation rule & single function defination scheme.

(iii) λ calculus emphasizes the use of transformation rules & doesn't care about the actual machine implementing them.

(iv) This is an approach more related to s/w than h/w

(v) It consist of constructing λ & performing reduction operation on them.

(vi) In λ calculus every expression stand for a function with a single i/p called its argument the argument of the function is in turn a function with a single argument & value of function is another function with a single argument. for eg.

Add two functions f such that $f(x) = x+2$ would be expressing λ calculus as

$\lambda x. x+2$.

A lambda expression is defined as one of the following

(a) x (variable) → a character or string representing a parameter or logical value

(b) $(\lambda x, M)$ (Abstraction) → The function definition ('M' is a λ term), variable x become bound in the expression

(c) (M, N) (Application) → Applying a function to an argument (N & M are the λ terms)

Reduction in λ calculus

λ expression is defined by how expression can be reduced three types of reductions are

(a) Alpha conversion → It allows bound variable names to be changed for eg. An alpha conversion of $\lambda x.x$ would be $\lambda y.y$

(b) β reduction → Applying function to their arguments

(c) η conversion → It expresses idea of extensibility in which this context is that two functions are same if & only if they give same result for all the arguments

How to write λ calculus function in Python programming language :→

lambda <a Parameter list> : <a python expression using parameter>

```
add = lambda i,j : i+j
print reduce (lambda i,j : i+j , number)
L = map(lambda i : i , &parm)
print L
```

Arithmetic in λ calculas

There are several ways to define natural no. in λ calculas but by far the most common are the church numerals which can be defined as follows

$$0 = \lambda f x$$
$$1 = \lambda f x \cdot f(x)$$
$$2 = \lambda f(x) \cdot f(f(x))$$
$$3 = \lambda f(x) \cdot f(f(fx))$$
& so on ----

The no. n in λ calculas is a function that takes a function f & argument and returns the n^{th} composition of f.

Addition is defined as

$$PLUS := \lambda mn\, fx \cdot mf(mfx)$$

PLUS 2 3 and 5

Logic and predicates in lambda Calculas

By convention the following two definations are used for the boolean values true & false ie.

$$TRUE := \lambda xy \cdot x$$
$$FALSE := \lambda xy \cdot y$$

NOTE → FALSE is equivalent church numerals 0 defined as ~~above~~ above.

with these two λ terms as TRUE & FALSE we can define some logic operators

① AND := λPQ . PQ FALSE
② OR := λPQ PTRUE Q
③ NOT := λP FALSE TRUE
④ IFTHENELSE := λPxy . Pxy

A predicate is a function which return a boolean value the most fundamental predicate is ZERO which return TRUE. If it's argument is church numerals ZERO & false if it's argument is any other church numerals

Recursion in λ Calculas :—
recursion is defination of function using the function itself on the face of it the λ calculas does not allow this however this imperation is misleading.
Consider for instance the factorial function f(n) recursively defined by f(n) = 1 if n=0 and n·f(n-1) . if n>0

In λ calculas one can not be defined a function which include itself to get around this one may start by defining a function called 'g' which takes a function f as an argument and ~~write~~ return another function that

takes n as a agrument i.e

$g := \lambda fn.(\lambda, \text{if } n=0 \text{ ; and } n\,f(n-1), \text{if } n>0)$

the function that g returns is either constant one or n times the application of function f to n-1

Using the ZERO predicate n boolean n algebric definations the function g can be defined in λ calculas

However, g by itself is still not recursive in order to use g to create recursive factorial function. the function pass to g as f must have specific properties. The function pass as f must expand to function g called with 1 argument and that argument must be the the function that was passed as f again.

In other words. f must expand to g(f) this call to g will then expand to the factorial function, calculate down another level of recursion.

Functional programming Langrages

LISP, Prolog

LISP :- It stands for list processing. It allow updation of program dynamicaly & provide high level debugging.

Application built in LISP
- → Emacan
- → Auto Cad
- → Yahoo store

Basic Building blocks in LISP

① Atoms → It is a no. or string or contiguous character.

eg. - Name
- 123019
- abc123
- b#123

② List → Sequence of atoms &/or other list enclosed in paranthesis

eg. - (I am a list)
- (a(abc) defgh)

③ String → group of characters enclosed in double quotation marks

eg. ("Hello world")
("I am here")

Lisp data types categories :—

① Scalar
eg. no., characters, symbol

② Data structure :—
eg. list, string, vector

Type specifiers in LISP

Array Atom bit character float	list string symbol	number Integer 1

A simple program in LISP :—

sum of 3 numbers :—

(+ 7 9 11)
27

Global Variable

(defvar x 234)
(write x)
o/p 234

Operation in LISP

① Arithmetic :— if A = 10; B = 20

(+ A B) = 30 , (* A B) = 200 , (mod B A) = 0
(- A B) = 10 , (/ B A) = 2

② Comparison :-

= (= A B)
> (> A B)
< (< A B)
max (max A B)
min (min A B)

③ Logical

And (and A B)
or (or A B)
not (not A B)

④ Bitwise

~~(logical ab~~
(logand a b)
(log or a b)
(log nor a b)
(log equ a b)

Defining function in LISP

(defun average sum (x1 x2 x3 x4))
(/ (+ x1 x2 x3 x4) 4)
(write (average sum 10, 20, 30, 40))

o/p 25

(defun (append x y)
(cond (null x) y)
(t (cons (car x) (append Bajau) y)))

factorial

```
(defun factorial (num))
(cond ((zero num) 1)

(t (* num(factorial (-num 1))))
 ))

(setq n 6)
(format t "%d factorial %d is : %d" n(factorial n))
```

Ans: 1. WAP in lisp to find cube of no.
2. WAP in lisp to reverse a list
3. WAP in lisp to search an element in list

~~(defun (cube~~

1. ~~(defun (cube) (num))~~

~~(* num num num)~~

```
(defun cube (num))
 (* num num num)
```

Prolog (logic in programming)

Declarative paradigm :—
what is to be done rather than how to do it

Logic → Relations
↓
(fact + rules)

In prolog logic is expressed as relations called facts and rules it is a pure declarative language a programmer specify a goal to be achive & a prolog system work out to achive the same.

- A prolog program specify relationship among objects. Relationship can also rules such as two peoples are sisters if they both are female & have some parents.
- A rule allows us to find out about a relationship given if the relationship is not stated as fact.
- fact has the properties of objects
- prolog is essentially a query language for db like Sql (structure query language) It is a query language for matching complicating patterns against a db of simple facts.
- All prolog programs consist of three parts a list of facts, a list of pattern matching rules (predicates) and a list of queries.

Prolog fact → fact in prolog are predefined patterns that store in prolog internal db usually in a manner to make searching them more efficient.

There are essentially three types of value in prolog

① Numbers :— 1, 2.0, -5 etc

② Symbols :— which are for all intent & purposes, immutable string of of lower case letters without special character or space for eg hello

③ Link list :— link list of symbols or numbers list are un typed for eg. [1, 2, 3] & [hello, world]

Prolog that defines several facts →

```
hello.
world.
f(hello, world).
g([hello, world])
standard_greeting([hello, world], 2)
```

when several patterns are defined with same name & same no. of arguments prolog will run through them one after another in top to down fashion when trying to match them.

Defining pattern matching rules in prolog →

f(hello, world) :- g(hello, world).

whenever prolog see the special symbol (:-) prolog creates a new pattern matching rule the basic meaning ~~of this~~ (:-) is to match whatever is to left of this symbol & part to right must be matched this allow to decompose pattern into small & more managable one.

To make task practical prolog defines many operator that help us in the task of composing matching rules some of more imp & useful are

Comma (,) a comma denotes sequential matching of patterns this is eq to a short circuit and in many imperative programming language. for eg.

```
f :- a, b, c
```

Semicolon (;) → It denotes the choice ie equivalent to a short circuit or in many imperative programming language

for eg

```
f :- p; q; r.
```

(→) An arrow denotes a conditional pattern rule, in other words this is if then else rule eg.

```
(g → h; i)
```

(+) this is equivalent to the negation operator in programing language.

f :- \+ g

this code means pattern f match whenever pattern g cannot be matched.

Variables in Prolog. → Symbols that begins with capital letter for eg.

Var, A, MATCH_ME

whenever prolog comes upon a variable it starts searching in its internal db of fact & pattern matching rules for substitution such that substituting a value match the variable some facts.