COMPOSITION :-

Has-a relationship.

(Types of inheritance discussed ahead) **INHERITANCE :-**

IS-a-relationship.

PolyMorphism :-

The study of polymorphism involves.

- ✓(i) Pointers to objects.
- ✓(ii) Virtual functions.
- ✓(iii) Pure virtual functions & Abstract Classes.
- (iv) Early Binding / compile time polymorphism. (funct overload, operator overload)
- ✓(v) Dynamic Binding / runtime polymorphism.

▷ **Pointer to objects**

(i) single object

a pointer would alway require some memory location: given by dynamic memory allocation.

```
class test
{ private:
    int x;
  public:
    void show()
    {cout<<"Demo"<<endl;
    cout<<x;
    }
int main
{ test *ptr; ptr=new test(), ptr->show(); }
```

(ii) arrays of object.

Now accordingly if an array of pointer obj is created. Each index would require some memory; allocated with a loop.

```
class test
{ public
  void show()
    {cout<<"Demo"<<endl;
    };
int main(){
✓ test *ptr[5];
  for (int i=0; i<05; i++)
  {
    ptr[i] = new test();
  }
  ptr[1]->show; }
```

CODE:- Dynamic Binding.

SINGLE INHERITANCE → each derived class inherits one base class.

PURE VIRTUAL FUNCT.

ABSTRACT CLASS:- → Abstract class

```
class fooditems
{private: // will give error in inheritance ∴ use protected keyword
  ✓ char name[100];
  ✓ int ingredient;
  ✓ float price;

public:
  ✓ fooditem() : price(0), ingredient (0) { }
  ✓ virtual void makepayments() = 0;
  ✓ virtual void placeorder() = 0;
};
```

→ pure virtual functions.

```
✓ class pizza : public fooditems
  { public:
      void placeorder() ✓ override
  ✓ {
      cout << "Enter your name" << endl;
      cin >> name;
      cout << "Enter no. of ingredients" << endl;
      cin >> ingredient;
  }
✓ void placeorder() override
  {
    cout << "Your order is placed" << endl;
  ✓ price = 150;
};  cout << "Price is :" << endl; cout << price;
    cout << endl;
}
```

```
class sushi : public fooditems
{
  ↕ // same code.
};

class Tacos : public fooditems
{
  ↕ // same code.
}
```

VIRTUAL KEYWORD.

VIRTUAL & MULTIPLE INHERIT

Diamond problem occurs when a class inherits two derived class, both of which have one, single base class.

```
#include <iostream>
using namespace std;

class Base
{
 public:
  virtual void display()
   cout << "Base class" << endl;
}
class d1 : virtual public base
  {
   public
     void display overide()
{ cout << "D1 class" << endl;}};
```

base

d1 d2

final class

```
class d2 : virtual public
           Base
  public:
{ void display()
        overide
  {
  cout << "D2 class";
  }
class final: public
             d1, public
             d2
{ public:
    // nothing
  }
{ int main()
{ final obj;
  obj.display();
```

CODE: NON RUN-TIME

∴ COMPILE TIME BINDING (EARLY BINDING).

FUNCTION OVERLOADING.

↳ same funct name differentiated by no. of parametres

data type of parametres

```
class Mathoperations
 public:
int add(int a, int b)
    { int c;
      c=a+b
      return c;
    }
int add(int a, int b, int c)
    { int d
      d= a+b+c;
      return d;
    }
  }; }
int main()
{ mathoperations obj1,obj2

intsum1= obj1.add(12,12);

intsum2= obj2.add(12,12,12);

}
```

OPERATOR OVERLOADING.

```
// output of an array using << operator

friend ostream & operator << (ostream & output, A & obj)
{
    for (int i=0; i< obj.size ; i++)
    { output << obj.array[i];
    }
    return output;
}

istream & operator >> (ifstrem & input, A & obj)
{
    for (int i=0 ; i<obj.size ; i++)
    { input >> obj.array [i];
    }
    return input;
}
```

Templates!

Class templates:-

→ used to generically define class so that it is not restricted to any specific datatype.

```
template <class any>
class test
{ private:
  any n;
  public:
      void get_value()
    {cin>>n;}
      void print_value()
    { cout<< n<<endl;}
};
int main()
{
  test <int> obj;
  obj.getvalue();
  obj.print_value();
}
```

Templates are used in functions where they again, enhance the reuseability of the code.

```
template <class T>
T maximum (Tn,Tm)
{ if (n>m)
    return n;
  else if (m>n)
    return m;
}
int main()
{
int r ;
r= maximum<int>(3,9)
float p;
P= maximum<float>(10.3,11.4)
}
```

STACK:-

→ in it items are accesible to disp from only at one end.

stack is a linear data structure that operates in LIFO (last in first out) or FILO (First in last out).

Push(x) → insert X as the top element of the stack.

Pop () → removes the top element of the stack and returns it.

Top () → return the top element without removing it.

Push-operation

(i) Check if the stack is full.

(ii) If the stack is full the produce an error message and exit the program.

(iii) If the stack is not full, increment on the top of the stack, new value adds at the top position of the stack.

```
#include <iostream>
using namespace std;

class stack
{ private:-
  int top;
  int stk[10];
 public:
 stack()
  { top = -1; }

void push (int a)
  if (top == 9)
  {
   cout<< "Stack is overflown"<<endl;
   System ("pause");
  }
  else
 { top = top + 1;
 //pushing
   stk[top] = a;
 } };
```

```
void display ()
{
  if (top == -1)
{cout << "The stack is
        empty" << endl;
  }
else{int i
  for (i=top; i>=0; i--)
  {    cout << std[i];
       cout<<endl; } } }
};
```

Concept of template and stack push / display.

```
#include <iostream>
using namespace std;
template <class any>
class stack
{ private:
    int top;
  any stk[10];
  public:
   stack ()
   { top = -1; }
void push (any a)
  { if (top == 9)
    {
     cout << "stack is
      overflown "<<endl;
      system ("pause");
      return;
    }
  else
  {
    top = top + 1
    stk [top] = a;
  }
}
void display ()
{
 if (top == -1)
 {
  cout<< "The
    stack is empty";
  cout<< endl;
    system ("pause");
 }
else
for (int i = top; i<=0; i--)
{ cout << stk[i]<<endl
}}}};
```

```
int main ()
{
stack <float> obj ;

↓ // my code

return 0;
}
```

Pop function: Stack

The data item is removed from a stack by the operation of pop

* Before removing check if there is atleast 1 item in the stack. If there is no data item the stack is called empty.

* The situation is called as underflow.

```
void pop()
  if (top == -1)
  {
  cout<< "The stack
    is empty, condition
    of under flow"<<endl;
  system ("pause");

  return;
  }
  else
  {
  cout<< "popping:";
  cout<< "stk[pop];
  cout<<endl;

   stk [pop]=NULL;
  cout<<"popping!";
   cout<<endl;
  }
  }
```

DYNAMIC MEMORY ALLOCATION IN OOP.

1D character/integer array!

```
#include <iostream>
using namespace std;

class test
{
  private:
    char *array;
    int size;
  public:
  test ()
  {
    array = new char[size];
    ——— ①
  }
~test ()
  {
    delete [] array;
  }
void get_value()
{
cout << "This is a demonstration
  of getting values for a
  dynamic 1D array" <<endl;
for (int i=0; i<=size; i++)
  {
    cin>> array[i];
  }
}
↓ // Remaing code
  here.
```

2D dynamic Memory alloc

→ As we know a 2D dynamic array will have rows & coloumns.

CONSTRUCTOR.

(i) Row are to be initialized normally. as in ①

(ii) Coloums are to be initialize using loop of rows

DESTRUCTOR :-

(i) Rows are dealloc using loop

(ii) Remove normally.

```
#include <iostream>
using namespace std

class test {
private:
    int rows;
    int cols;
    char ** array;
public
    test (int a, int b) : rows(a) , cols(b)
    {
        array = new char * [rows];
        for (int i =0 i< rows ; i++)
          array [i] =  new char [cols];
    }
~ test()
    { for (int i=0 ; i<rows ; i++)
      { delete [ ] array [i];
      }
      delete [ ] array;
    }
```

FILE IO :-

```
# include <fstream>
# include <iostream>
using name space std;

int main()
{
int a,b,c;
ifstream fin;
fin. open ("my-input.txt");
fin>>a >>b;
c= a+b;
ofstream fout;
fout.open ("result_store.txt");
fout<<c<<endl;
fin.close();
fout.close();
}
```

--x---

```
# include <iostream>
using namespace std;
typedef int myint;
typedef struct cat
{ my int number;
```

Question:-

Write a cpp code that reads 30 numbers from a file, finds the largest and the smallest among 30 and stores it in a textfile while using only one loop.

```
// use you own logic
just set
smallest= array[0
largest=array[0
```

Typedef:-

A keyword that is used to provide exisitig data type a new user defined name.

```
void getinput {
cout<<"enter cat no.s"<<endl;
cin>>number;
};
int main()
{ ep obj
obj.getvalue();
}
```

This pointer! (beautiful concept!)

```
include <iostream>
using namespace std;

class Rectangle
{
  private:
   float length;
   float width;
 public:
Rectangle () : length (0), width (0)
  { }
void dimension_set (double l, double w)
  {
   this -> length = l;
   this -> width = w; }

float/double area() const {
    return length * width}

void check (Rectangle &obj)
  {
    if (this -> area > other.area())
      cout << "first object has greater
        are of rectangle " << endl;
```

=> very important concept
this whole value can be accessed using this pointer without actually storing it in a area's variable.

PRE-PROCESSING DIRECTIVES!

① SIMPLE DEFINE

```
#include <iostream>
using namespace std;

#define Max_size 100

int main ()

{
    int arr [max_size];
    for (int i=0; i< Max_size; i++)
    {
        arr [i] = i;
    }
    return 0;
}
```

MACRO

②
```
#include <iostream>
using namespace std;

#define PI 3.1428
#define circle_area (X) (PI*(X)*(X))
#define rectangle (X,Y) ((X)*(Y))

int main()
{ float area = circle_area (4);
  float area_2 = rectangle (2,3);
}
```

ASSERT AND REPLACEMENT

```
#include <iostream>
#include <cassert>

#define hello (x,y) cout << "Hello" #x "and" #y
                    << endl;

int main()
{
    hello ("Haris", "Bilal");
    int x=5;
    assert (x>=0)   // ensuring x>=0.
    cout << "x" << endl;
    return 0;
}
```

// if assert condition is not fulfilled the programme is terminated.

__FILE__ __DATE__ __TIME__

NDEBUG & PREDEFINED SYMBOLIC CONST

date file time

```
#ifndef NDEBUG.
```

using this line above the main, it shall disable assert functionality, helps in debugging.

and should enclose the assert we want to disable would require #end if at the end to the main.

```
class Inlist
{
  struct Node
    {
      int data;
      Node*next;
    };
  Node * Listheadptr;
public :
  Inlist ()
  {  Listhead ptr = NULL;
  }
  void AddNode At Head (int value);
  void Add Node At Tail (int value);
  void display List ();
  void delete Node (int key);
  void displayspecific Node (int key);
  void sort List( );
  void reverse_sorted list ();

  ~ In list ()
  {
    Node *ptr previous;
    while (List Head ptr!=NULL)
      ptr previous = List Head ptr;
      List Head ptr = List head ptr->next;
      delete ptr previous. } }
```

```
void addatHead(int value)
{
    Node *new_Node;
    new_Node = new Node;

    new_Node -> data = value;
    new_Node -> next = ListHeadptr;
    ListHeadptr = new_Node;
}
```

// check if head is NULL the we donnot need to traverse
// else we need to.

```
void addatTail (int value)
{
    Node *newptr;
    newptr = new Node;

    newptr -> data = value;
    newptr -> next = NULL;

if (ListHeadptr == NULL)   // checking if the list is
                              empty.
{   ListHeadptr = newptr;
    return;
}   Node *ptrtemp;
    ptrtemp = newNode;   ptrtemp = ListHeadptr
while (ptrtemp -> next != NULL)

    ptrtemp = ptrtemp -> Next;

}   ptrtemp -> Next = newptr;
```

```
void display list ()
// remember that display function is only for displaying
the data inside of a list.

{Node *ptr temp = new Node;
    ptrtemp = List Headptr;
    while (ptrtemp != NULL)
    {
cout << ptrtemp -> data << endl;
    ptrtemp = ptrtemp -> next;
    }
}

void display specific Node (int key).

{ Node *searchptr = new Node
    searchptr = List Headptr;
while (searchptr != NULL && sparchptr -> data != key)
    searchptr = searchptr -> next;

if (searchptr == NULL)
    {cout << "The value doesnot exisit" << endl;}
else
    { cout << "The value data for the Node is
        << searchptr -> data << endl; }
```

```
void deleteNode (int key)
{
    node* currentptr = ListHeadptr;
    node* previous ptr;

    while (ptrcurrent != Null && ptrcurrent->data != key)
        ptrcurrent = ptrcurrent->next;
        ptr previous = ptr current;
        if (ptr current == NULL) { cout << "No value found";
    if (ptrcurrent == ListHeadptr)
        ListHead ptr = ListHeadptr->next;

    else
    // now either at the middle or at the end;
        ptr previous -> next = ptrcurrent->next;
        delete ptrcurrent;
}
```

// either the key value wont exisit
// or it will be at the head.
// or it will be at the end or middle.

void sortAscending()

if (List Head Ptr == NULL || LIST Head ptr->next == NULL)

{ cout<< "only one or node to sort <<endl; }
return;

Node * ptr current = new Node;

Node * ptrnext = NULL;

int tempdata;

bool swapped;

do

{ swapped = false;

ptr current = Listheadptr;

while (ptrcurrent->next != ptrnext) && ptr current->next->data;

{ tempdata = ptrcurrent->data;

ptrcurrentdata = ptrcurrent->next->data;

ptrcurrent->next->data = tempdata;

swapped = true

}

ptrcurrent = ptrcurrent->next;

iterator of while.

while closes

ptr Next = ptrcurrent

because ptrcurrent by bubble sort is now at its current sorted position. (at the end being the largest. so we dont need to access it.

} while (swapped);

}

```
void reverse list( )
{
  Node ptr previous* = NULL;
  Node ptr current* = listhead ptr;
  Node     *nextptr = NULL.

  while(ptrcurrent != NULL)
  {
    nextptr = current -> next;
    current -> next = ptrprevious;

    ptr previous = ptrcurrent;
    ptrcurrent = nextptr;
  }
  list head ptr = ptrprevious;
}
```