OBJECT ORIENTED PROGRAMMING

PUBLIC AND PRIVATE CLASS

Classes are mainly used to hide data,
As they contain public and private class.
And also they are very effecient as compared to structures,
As you run many function in classes , which is a major limitation of structures

```
#include <iostream>
using namespace std;
class employee {
   private: int a,b,c ;

   public: int d , e;
   void setdata(int a1,int b1 ,int c1);//decleration
   void getdata(){
      cout <<"the value of a is "<<a<<endl;
      cout <<"the value of b is "<<b<<endl;
      cout <<"the value of c is "<<c<<endl;
      cout <<"the value of d is "<<d<<endl;
      cout <<"the value of e is "<<e<<endl;
   }

};

void employee :: setdata(int a1 , int b1 ,int c1){
   a=a1;
   b=b1;
   c=c1;
}
int main(){
   employee vashu;
   //vashu.a=134;
   //vashu.b=175;      ------> these will throw error
   //vashu.c=197;            because they are private
   vashu.d=109;
```

```
    vashu.e=169;
    vashu.setdata(1,2,3);
    vashu.getdata();

    return 0;
}
```

USING ARRAYS IN CLASSES

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

class Shop
{
    int itemId[100];
    int itemPrice[100];
    int counter;

public:
    void initCounter(void) { counter = 0; }
    void setPrice(void);
    void displayPrice(void);
};

void Shop ::setPrice(void)
{
    cout << "Enter Id of your item no " << counter + 1 << endl;
    cin >> itemId[counter];
    cout << "Enter Price of your item" << endl;
    cin >> itemPrice[counter];
    counter++;
}

void Shop ::displayPrice(void)
{
    for (int i = 0; i < counter; i++)
    {
        cout << "The Price of item with Id " << itemId[i] << " is " << itemPrice[i] <<
endl;
    }
}

int main()
{
    Shop dukaan;
    dukaan.initCounter();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.displayPrice();
    return 0;
}
```

STATIC DATA MEMBERS or STATIC VARIABLES

Also known as class variable
It retains the value

STATIC FUNCTION

Is function which deals only with static variables or static data members

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

class Employee
{
    int id;
    static int count;

public:
    void setData(void)
    {
        cout << "Enter the id" << endl;
        cin >> id;
        count++;
    }
    void getData(void)
    {
        cout << "The id of this employee is " << id << " and this is employee
number " << count << endl;
    }

    static void getCount(void){
        // cout<<id; // throws an error
        cout<<"The value of count is "<<count<<endl;
    }
};

// Count is the static data member of class Employee
int Employee::count; // Default value is 0

int main()
{
    Employee harry, rohan, lovish;
    // harry.id = 1;
    // harry.count=1; // cannot do this as id and count are private

    harry.setData();
    harry.getData();
    Employee::getCount();

    rohan.setData();
    rohan.getData();
    Employee::getCount();

    lovish.setData();
    lovish.getData();
    Employee::getCount();

    return 0;
}
```

ARRAYS OF OBJECT

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

class Employee
{
    int id;
    int salary;

public:
    void setId(void)
    {
        salary = 122;
        cout << "Enter the id of employee" << endl;
        cin >> id;
    }

    void getId(void)
    {
        cout << "The id of this employee is " << id << endl;
    }
};

int main()
{
    // Employee harry, rohan, lovish, shruti;
    // harry.setId();
    // harry.getId();
    Employee fb[4];
    for (int i = 0; i < 4; i++)
    {
        fb[i].setId();
        fb[i].getId();
    }

    return 0;
}
```

PASSING OBJECT AS FUNCTION ARGUMENTS

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

class complex{
    int a;
    int b;

    public:
        void setData(int v1, int v2){
            a = v1;
            b = v2;
        }

        void setDataBySum(complex o1, complex o2){
            a = o1.a + o2.a;
            b = o1.b + o2.b;
        }

        void printNumber(){
            cout<<"Your complex number is "<<a<<" + "<<b<<"i"<<endl;
        }
};

int main(){
    complex c1, c2, c3;
    c1.setData(1, 2);
    c1.printNumber();

    c2.setData(3, 4);
    c2.printNumber();

    c3.setDataBySum(c1, c2);
    c3.printNumber();
    return 0;
}
```

FRIEND FUNCTION

Used to ascess private data of a class by a function
Which is not present in the class.

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

// 1 + 4i
// 5 + 8i
// -------
// 6 + 12i
class Complex{
    int a, b;
    friend Complex sumComplex(Complex o1, Complex o2);
    public:
        void setNumber(int n1, int n2){
            a = n1;
            b = n2;
        }

        // Below line means that non member - sumComplex funtion is allowed to
do anything with my private parts (members)
        void printNumber(){
            cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;
        }
};

Complex sumComplex(Complex o1, Complex o2){
    Complex o3;
    o3.setNumber((o1.a + o2.a), (o1.b+o2.b))
    ;
    return o3;
}

int main(){
    Complex c1, c2, sum;
    c1.setNumber(1, 4);
    c1.printNumber();

    c2.setNumber(5, 8);
    c2.printNumber();

    sum = sumComplex(c1, c2);
    sum.printNumber();

    return 0;
}
```

Properties of friend functions

1. Not in the scope of class
2. since it is not in the scope of the class, it cannot be called from the object of that class. c1.sumComplex() == Invalid
3. Can be invoked without the help of any object
4. Usually contains the objects as arguments
5. Can be declared inside public or private section of the class
6. It cannot access the members directly by their names and need object_name.member_name to access any member.

CONSTRUCTORS

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

```
class Complex
{
    int a, b;

public:
    // Creating a Constructor
    // Constructor is a special member function with the same name as of the
class.
    //It is used to initialize the objects of its class
    //It is automatically invoked whenever an object is created

    Complex(void); // Constructor declaration

    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};

Complex ::Complex(void) // ----> This is a default constructor as it takes no
parameters
{
    a = 10;
    b = 0;
    // cout<<"Hello world";
}
int main()
{
    Complex c1, c2, c3;
    c1.printNumber();
    c2.printNumber();
    c3.printNumber();

    return 0;
}
```

PARAMETERIZED AND DFAULT CONSTRUCTORS

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

#include<iostream>
using namespace std;

class Point{
    int x, y;
    public:
        Point(int a, int b){
            x = a;
            y = b;
        }

        void displayPoint(){
            cout<<"The point is ("<<x<<", "<<y<<")"<<endl;
        }

};

int main(){
    Point p(1, 1);
    p.displayPoint();

    Point q(4, 6);
    q.displayPoint();
    return 0;
}
```

CONSTRUCTOR OVERLOADING

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

class Complex
{
    int a, b;

public:
    Complex(){
        a = 0;
        b =0;
    }

    Complex(int x, int y)
    {
        a = x;
        b = y;
    }

    Complex(int x){
        a = x;
        b = 0;
    }



    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};
int main()
{
    Complex c1(4, 6);
    c1.printNumber();

    Complex c2(5);
    c2.printNumber();

    Complex c3;
    c3.printNumber();
    return 0;
}
```

CONSTRUCTORS WITH DEFAULT ARGUMENTS

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

class Simple{
   int data1;
   int data2;
   int data3;

   public:
      Simple(int a, int b=9, int c=8){
         data1 = a;
         data2 = b;
         data3 = c;
      }

      void printData();

};

void Simple :: printData(){
   cout<<"The value of data1, data2 and data3 is "<<data1<<", "<< data2<<" and "<< data3<<endl;
}
int main(){
   Simple s(12, 13);
   s.printData();
   return 0;
}
```

DYNAMIC INSTALIZATION OF OBJECTS USING CONSTRUCTORS

You can put many constructors in a class but it is not defined
That which one is going to work .
You will get to know that which constructors work only when you run the code.
Depending on the user input.

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


class BankDeposit{
    int principal;
    int years;
    float interestRate;
    float returnValue;

    public:
        BankDeposit(){}
        BankDeposit(int p, int y, float r); // r can be a value like 0.04
        BankDeposit(int p, int y, int r); // r can be a value like 14
        void show();
};
BankDeposit :: BankDeposit(int p, int y, float r)
{
    principal = p;
    years = y;
    interestRate = r;
    returnValue = principal;
    for (int i = 0; i < y; i++)
    {
        returnValue = returnValue * (1+interestRate);
    }
}

BankDeposit :: BankDeposit(int p, int y, int r)
{
    principal = p;
    years = y;
    interestRate = float(r)/100;
    returnValue = principal;
    for (int i = 0; i < y; i++)
    {
        returnValue = returnValue * (1+interestRate);
    }
}

void BankDeposit :: show(){
    cout<<endl<<"Principal amount was "<<principal
        << ". Return value after "<<years
        << " years is "<<returnValue<<endl;
}
int main(){
    BankDeposit bd1, bd2, bd3;
    int p, y;
    float r;
    int R;


    cout<<"Enter the value of p y and r"<<endl;
    cin>>p>>y>>r;
    bd1 = BankDeposit(p, y, r);
    bd1.show();

    cout<<"Enter the value of p y and R"<<endl;
    cin>>p>>y>>R;
    bd2 = BankDeposit(p, y, R);
    bd2.show();
    return 0;
}
```

COPY CONSTRUCTOR

A copy constructor is a member function that
initializes an object using another object of the same class

In short copy a constructor in other constructor

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


class Number{
    int a;
    public:
        Number(){
            a = 0;
        }

        Number(int num){
            a = num;
        }
        // When no copy constructor is found, compiler supplies its own copy
constructor
        Number(Number &obj){
            cout<<"Copy constructor called!!!"<<endl;
            a = obj.a;
        }

        void display(){
            cout<<"The number for this object is "<< a <<endl;
        }
};



int main(){
    Number x, y, z(45), z2;
    x.display();
    y.display();
    z.display();

    Number z1(z); // Copy constructor invoked
    z1.display();

    z2 = z; // Copy constructor not called
    z2.display();

    Number z3 = z; // Copy constructor invoked
    z3.display();
    // z1 should exactly resemble z  or x or y

    return 0;
}
```

DESTRUCTOR

A destructor is **a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete**

✭ A destructor has the same name as the class, preceded by a tilde (~)

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

// Destructor never takes an argument nor does it return any value
int count=0;

class num{
    public:
        num(){
            count++;
            cout<<"This is the time when constructor is called for object 
number"<<count<<endl;
        }

        ~num(){
            cout<<"This is the time when my destructor is called for object 
number"<<count<<endl;
            count--;
        }
};
int main(){
    cout<<"We are inside our main function"<<endl;
    cout<<"Creating first object n1"<<endl;
    num n1;
```

```
   {
      cout<<"Entering this block"<<endl;
      cout<<"Creating two more objects"<<endl;
      num n2, n3;
      cout<<"Exiting this block"<<endl;
   }
   cout<<"Back to main"<<endl;
   return 0;
}
```

INHERITANCE

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.
The existing class is called BASE CLASS.
The new class which is inherited is called DERIVED CLASS.

TYPES

—> SINGLE INHERITANCE : a derived class with only one base class.

—> MULTIPLE INHERITANCE : a derived class with more than one base class.

—> HIERARCHICAL INHERITANCE : several derived derived class from single base class.

—> MULTILEVEL INHERITANCE : deriving a class from already derived class.

—> HYBIRD INHERITANCE : combination of multiple inheritance and multilevel inheritance.
Two or more classes are derived from one base class as ,
And these derived class further combine to give one single class.

BASIC SYNTAX OF INHERITANCE AND VISIBILITY MODE

Note:

- Default visibility mode is private
- Public Visibility Mode: Public members of the base class becomes Public members of the derived class
- Private Visibility Mode: Public members of the base class become private members of the derived class
- Private members are never inherited

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

class employee{

    public :
    int id ;
    float salary ;
    employee (int inpid){
        id=inpid;
        salary=34.0;
        }
        employee (){
        }
};

class programmer : public employee{
    public :
    int languagecode ;
    programmer (int inpId){
        id = inpId;
        languagecode = 9;
    }
    void getdata(){
        cout <<id <<endl;
    }
};
int main ()
{
    employee harry(1), rohan (2);
    cout<<harry.salary<<endl;
    cout<<rohan.salary<<endl;
    programmer skillf(6);
    cout<<skillf.languagecode<<endl;
    cout<<skillf.id<<endl;
    skillf.getdata();

    return 0;
}
```

SINGLE INHERITANCE

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

class base {
    int data1 ;
    public:
    int data2 ;
    void setdata();
    int getdata1();
    int getdata2();

};

void base :: setdata (){
    data1 = 10;
    data2 = 20;
}

int base :: getdata1  (){
    return data1 ;
}

int base :: getdata2  (){
    return data2 ;
}


class derived : public base {
    int data3 ;
    public:
    void process ();
    void display ();

};

void derived :: process (){
    data3 = data2 * getdata1() ;
}

void derived :: display (){
    cout << "value of data1 is "<< getdata1()<< endl ;
    cout << "value of data2 is "<< data2<< endl ;
    cout << "value of data3 is "<< data3<< endl ;

}


int main ()
{
    derived der ;
    der.setdata();
    der.process();
    der.display();

    return 0 ;
}
```

MULTILEVEL INHERITANCE

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

class student {

    protected :
    int rollnumber ;

    public :
    void setrollnumber(int);
    void getrollnumber(void);

};

void student :: setrollnumber (int r){
    rollnumber = r;
}

void student :: getrollnumber (){
    cout<<"the roll number of the student is "<<rollnumber<<endl;
}

class exam : public student {

    protected :
    float maths;
    float physics;

    public :
    void setmarks(float,float);
    void getmarks();
};

void exam :: setmarks (float m1,float m2){
   maths= m1;
   physics= m2;
}


void exam :: getmarks (void){
    cout <<"the marks in maths is "<<maths<<endl;
    cout <<"the marks in physics is "<<physics<<endl;
}

class result : public exam{
    float percentage;
    public :
    void display(){
        cout <<"your percentge is "<<(maths+physics)/2<<"%"<<endl;
    }


};
int main(){
    result harry ;
    harry.setrollnumber(420);
    harry.setmarks(96.0,98.0);
    harry.getmarks();
    harry.display();

    return 0;
}
```

MULTIPLE INHERITANCE

SYNTAX —>   class derived : visibility mode base class 1 , visibility mode base class 2

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


class base1 {

    protected :
    int base1int;

    public:
    void getbase1int (int);
};

void base1 :: getbase1int (int a){
    base1int = a;
}


class base2 {

    protected :
    int base2int;

    public:
    void getbase2int (int);
};
```

```
void base2 :: getbase2int (int b){
    base2int = b;
}


class derived :public base1 , public base2{
    public :
    void show(){
        cout<<"the value of base1 is "<<base1int<<endl;
        cout<<"the value of base2 is "<<base2int<<endl;
        cout<<"the sum of base1 and base2 is "<<base1int+base2int<<endl;
    }
};


int main(){
    derived vashu;
    vashu.getbase1int(6);
    vashu.getbase2int(9);
    vashu.show();
    return 0;
}
```

EXERSICE 1

SIMPLE CALCULATOR AND SCIENTIFIC CALCULATOR USING INHERITANCE

```
#include <iostream>
#include <math.h>
using namespace std ;

class simplecalc{
    int a ,b ;
    public :
```

```
    void getdata (){
        cout<<"enter the value of a "<<endl;
        cin>>a;
        cout<<"enter the value of b "<<endl;
        cin>>b;
    }

    void performoperation (){
        cout <<"the value of a+b is "<<a+b<<endl;
        cout <<"the value of a-b is "<<a-b<<endl;
        cout <<"the value of a*b is "<<a*b<<endl;
        cout <<"the value of a/b is "<<a/b<<endl;

    }
};

class scientificcalc{
    int a , b ;
    public:
    void getdatascientific (){
        cout<<"enter the value of a "<<endl;
        cin>>a;
        cout<<"enter the value of b "<<endl;
        cin>>b;
    }

    void perform_scientificoperation(){
         cout << "The value of cos(a) is: " << cos(a) << endl;
            cout << "The value of sin(a) is: " << sin(a) << endl;
            cout << "The value of exp(a) is: " << exp(a) << endl;
            cout << "The value of tan(a) is: " << tan(a) << endl;
    }
} ;


class hybridcalc : public simplecalc , public scientificcalc{

};

int main (){
    hybridcalc vashu;
    vashu.getdata();
    vashu.performoperation();
    vashu.getdatascientific();
    vashu.perform_scientificoperation();
    return 0;
}
```

AMBIGUITY RESOLUTION IN INHERITIANCE

Ambiguity means if you have more than one function of same name
And you have to call a function then which function is called ……

EXAMPLE 1

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


class base1 {
   public :
   void greet(){
   cout<<"how are you "<<endl;
   }
};


class base2 {
   public :
   void greet (){
      cout<<"kaise hoe "<<endl;
   }
};


class derived : public base1 , public base2{
   int a ;
   public :
   void greet(){
      base1::greet();
   }

};
```

```
int main (){
    base1 base1obj;
    base2 base2obj;
    base1obj.greet();
    base2obj.greet();
    derived d;
    d.greet();
    return 0;
}
```

EXAMPLE 2

If function is also defined in derived class
Then the object of derived class is able to call the function

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


class B {
    public :
    void say(){
    cout<<" hello world "<<endl;
    }
};


class D : public B{
    int d;
    public:
    void say (){
        cout<<"hello my beautiful people"<<endl;
    }
};


int main (){
    B b;
    b.say();
    D d;
    d.say();
    return 0;
}
```

VIRTUAL BASE CLASS

This class is **A** is inherited by two other classes **B** and **C**.
Both these class are inherited into another in a new class **D**
When any data / function member of class **A** is accessed by an object of class **D**,
ambiguity arises as to which data/function member would be called?
One inherited through **B** or the other inherited through **C**.
This confuses compiler and it displays error.

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

class student {
    protected :
    int rollnum ;
    public :
    void setrollnum (int a){
        rollnum=a;
    }
    void printrollnum (void){
        cout<<"your roll num is "<<rollnum<<endl;
    }
};

class test : virtual public student {
    protected :
    float maths , physics ;
    public :
    void setmarks(float m1 , float m2) {
        maths =m1;
        physics=m2;
    }
    void printmarks (void){
        cout<<"your result is "<<endl
```

```
            <<"maths :"<<maths <<endl
            <<"physics :"<<physics<<endl;
    }
};

class sport : virtual public student {
    protected :
    float score ;
    public :
    void setscore (float sc){
        score = sc;
        }
    void printscore (){
        cout<<"your pt score is "<<score <<endl;
    }
};

class result : public test , public sport {
    private :
    float total;
    public :
    void display(){
        total = maths + physics + score ;
        printrollnum();
        printmarks ();
        printscore();
        cout << "your total score is "<<total <<endl;
    }
};

int main (){
    result vashu;
    vashu.setrollnum(21);
    vashu.setmarks(100,100);
    vashu.setscore(100);
    vashu.display();
    return 0 ;
}
```

CONSTRUCTORS IN DERIVED CLASS

IN MULTIPLE and MULTILEVEL INHERITANCE ——> constructor execute in the order in which class is declared .

Special Case of Virtual Base Class

- The constructors for virtual base classes are invoked before a non-virtual base class
- If there are multiple virtual base classes, they are invoked in the order declared
- Any non-virtual base class are then constructed before the derived class constructor is executed

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

/*
Case1:
class B: public A{
   // Order of execution of constructor -> first A() then B()
};

Case2:
class A: public B, public C{
    // Order of execution of constructor -> B() then C() and A()
};

Case3:
class A: public B, virtual public C{
    // Order of execution of constructor -> C() then B() and A()
};

*/

class Base1{
    int data1;
    public:
        Base1(int i){
            data1 = i;
            cout<<"Base1 class constructor called"<<endl;
        }
        void printDataBase1(void){
            cout<<"The value of data1 is "<<data1<<endl;
        }
};

class Base2{
    int data2;

    public:
        Base2(int i){
            data2 = i;
            cout << "Base2 class constructor called" << endl;
        }
        void printDataBase2(void){
            cout << "The value of data2 is " << data2 << endl;
        }
};

class Derived: public Base2, public Base1{
    int derived1, derived2;
    public:
        Derived(int a, int b, int c, int d) : Base2(b), Base1(a)
        {
            derived1 = c;
            derived2 = d;
            cout<< "Derived class constructor called"<<endl;
        }
        void printDataDerived(void)
        {
            cout << "The value of derived1 is " << derived1 << endl;
            cout << "The value of derived2 is " << derived2 << endl;
        }
};


int main(){
    Derived harry(1, 2, 3, 4);
    harry.printDataBase1();
    harry.printDataBase2();
    harry.printDataDerived();
    return 0;
}
```

INITILIZATION LIST IN CONSTRUCTORS

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

class Test
{
   int a;
   int b;

public:
   //Test(int i, int j) : a(i), b(j)
   //Test(int i, int j) : a(i), b(i + j)
   //Test(int i, int j) : a(i), b(i * j)
   //Test(int i, int j) : a(i), b(a + j)
   //Test(int i, int j) : b(j), a(i + b) --> red flag this will create problems because a
will be initialized first.

   {
      cout << "Constructor executed"<<endl;
      cout << "Value of a is "<<a<<endl;
      cout << "Value of b is "<<b<<endl;
   }
};

int main()
{
   Test t(4, 6);

   return 0;
}
```

NEW OPERATOR AND POINTERS

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

```
int main(){

    float *p = new float(40.78);
    cout << "The value at address p is " << *(p) << endl;

    return 0;
}
```

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

int main(){

    int *arr = new int[3];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    cout << "The value of arr[0] is " << arr[0] << endl;
    cout << "The value of arr[1] is " << arr[1] << endl;
    cout << "The value of arr[2] is " << arr[2] << endl;

    return 0;
}
```

DELETE OPERATOR

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

int main(){

    int *arr = new int[3];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    delete[] arr;
    cout << "The value of arr[0] is " << arr[0] << endl;
    cout << "The value of arr[1] is " << arr[1] << endl;
    cout << "The value of arr[2] is " << arr[2] << endl;

    return 0;
}
```

POINTERS TO OBJECT AND ARROW OPERATOR

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

class complex {
    int  real,img ;
    public :
    void getdata(){
cout << "the real part is "<<real<<endl;
cout << "the imaginary part is "<<img<<endl;
    }

    void setdata(int a , int b){
  real=a;
  img=b;
    }
};

int main(){
  complex c1 ;
  complex (*ptr)=&c1;
   (*ptr).setdata(4,6);
  //(*ptr).getdata(); is exacty same as
//also
  ptr->getdata();
    return 0;
}
```

ARRAYS OF OBJECTS

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

class complex {
    int  real,img ;
    public :
    void getdata(){
cout << "the real part is "<<real<<endl;
cout << "the imaginary part is "<<img<<endl;
    }

    void setdata(int a , int b){
  real=a;
  img=b;
    }
};

int main(){
  complex *ptr1 = new complex[4];
    ptr1->setdata(1, 4);
    ptr1->getdata();
    return 0;
}
```

ARRAYS OF OBJECT USING POINTER

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

class ShopItem
{
    int id;
    float price;
    public:
        void setData(int a, float b){
            id = a;
            price = b;
        }
        void getData(void){
            cout<<"Code of this item is "<< id<<endl;
            cout<<"Price of this item is "<<price<<endl;
        }
};
int main(){
    int size = 3;
    ShopItem *ptr = new ShopItem [size];
    ShopItem *ptrTemp = ptr;
    int p, i;
    float q;
    for (i = 0; i < size; i++)
    {
        cout<<"Enter Id and price of item "<< i+1<<endl;
        cin>>p>>q;
        // (*ptr).setData(p, q);
        ptr->setData(p, q);
        ptr++;
    }

    for (i = 0; i < size; i++)
    {
        cout<<"Item number: "<<i+1<<endl;
        ptrTemp->getData();
        ptrTemp++;
    }


    return 0;
}
```

THIS POINTER

This is a keyword which is a pointer which points to the object which invokes the member function.

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

class A {
   int a ;
   public :
   void setdata(int a){
this->  a=a;
   }
   void getdata(){
      cout << " the  value of a is "<<a<<endl;
   }
};
int main (){
A a;
a.setdata(4);
a.getdata();
   return 0;
}
```

POINTERS TO DERIVED CLASS

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

class baseclass {
public :
int var_base ;
void display(){
   cout << " displaying base class variable var_base "<< var_base <<endl;
}

};

class derivedclass : public baseclass{
public :
int var_derived ;
void display(){
   cout << " displaying derived class variable var_derived "<< var_derived
<<endl;
}
};


int main (){
baseclass *  base_class_pointer;
baseclass obj_base;
derivedclass obj_derived;
base_class_pointer= &obj_derived;//pointing base class pointer to derived class
   return 0;
}
```

```
#include<iostream>
using namespace std;
class BaseClass{
   public:
      int var_base;
      void display(){
         cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
      }
};

class DerivedClass : public BaseClass{
   public:
         int var_derived;
         void display(){
            cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
            cout<<"Dispalying Derived class variable var_derived
"<<var_derived<<endl;
         }
};
int main(){
   BaseClass * base_class_pointer;
   BaseClass obj_base;
   DerivedClass obj_derived;
   base_class_pointer = &obj_derived; // Pointing base class pointer to derived
class

   base_class_pointer->var_base = 34;
   // base_class_pointer->var_derived= 134; // Will throw an error
   base_class_pointer->display();

   base_class_pointer->var_base = 3400;
   base_class_pointer->display();

   DerivedClass * derived_class_pointer;
   derived_class_pointer = &obj_derived;
   derived_class_pointer->var_base = 9448;
   derived_class_pointer->var_derived = 98;
   derived_class_pointer->display();

   return 0;
}
```

POLYMORPHISM

The word polymorphism means having many forms.

In C++ polymorphism is mainly divided into two types

```
             I                                  I
             I                                  I
             I                                  I
             V                                  V
        COMPILE TIME                         RUN TIME
```

```
          POLYMORPHISM                    POLYMORPHISM
          I             I                       I
          I             I                       I
          V             V                       V
      FUNCTION      OPERATOR                 VIRTUAL
    OVERLOADING    OVERLOADING              FUNCTIONS
```

VIRTUAL FUNCTIONS

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

class BaseClass{
   public:
      int var_base=1;
      virtual void display(){
         cout<<"1 Dispalying Base class variable var_base "<<var_base<<endl;
      }
};

class DerivedClass : public BaseClass{
   public:
         int var_derived=2;
         void display(){
            cout<<"2 Dispalying Base class variable var_base
"<<var_base<<endl;
            cout<<"2 Dispalying Derived class variable var_derived
"<<var_derived<<endl;
         }
};
int main(){
   BaseClass * base_class_pointer;
   BaseClass obj_base;
   DerivedClass obj_derived;

   base_class_pointer = &obj_derived;
   base_class_pointer->display();
```

```
    return 0;
}
```

VIRTUAL FUNCTIONS EXAMPLE

```
#include <iostream>
#include <cstring>

using namespace std ;

class CWH{
    protected:
        string title;
        float rating;
    public:
        CWH(string s, float r){
            title =  s;
            rating = r;
        }
        virtual void display(){}
};
class CWHVideo: public CWH
{
    float videoLength;
    public:
        CWHVideo(string s, float r, float vl): CWH(s, r){
            videoLength = vl;
        }
        void display(){
            cout<<"This is an amazing video with title "<<title<<endl;
            cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;
            cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;
        }
};
class CWHText: public CWH
{
    int words;
    public:
        CWHText(string s, float r, int wc): CWH(s, r){
            words = wc;
        }
     void display(){
      cout<<"This is an amazing text tutorial with title "<<title<<endl;
      cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;
      cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;
         }
};
int main(){
    string title;
    float rating, vlen;
    int words;

    // for Code With Harry Video
    title = "Django tutorial";
    vlen = 4.56;
    rating = 4.89;
    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text
    title = "Django tutorial Text";
    words = 433;
    rating = 4.19;
    CWHText djText(title, rating, words);

    CWH* tuts[2];
    tuts[0] = &djVideo;
    tuts[1] = &djText;

    tuts[0]->display();
    tuts[1]->display();

    return 0;
}
```

OPERATOR OVERLOADING

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

class date {

  int day ;
  int month ;
  int year;

  public :

  void setdata(){
      int d;
      int m;
      int y;
  cout <<"enter the day "<<endl;
  cin>>d;
  cout <<"enter the month "<<endl;
  cin>>m;
  cout <<"enter the year "<<endl;
  cin>>y;


  day =d;
  month =m;
  year =y;

  }

  void displaydata (){

    cout <<" date is "<<day<<"/"<<month<<"/"<<year<<endl;

  }

  date operator + (date  const &x){
      date temp;
      temp.day = day + x.day;
      temp.month = month + x.month;
      temp.year = year + x.year;

      return temp ;
  }
};
```

```
int main (){
date d1;
d1.setdata();
d1.displaydata();

date d2;
d2.setdata();
d2.displaydata();

date d3;
d3=d1+d2;
d3.displaydata();
    return 0;
}
```

FILE HANDLING

WRITING IN FILE

```
#include<iostream>
#include<fstream>

using namespace std;

int main(){
    string st = "vashu bhai";

    ofstream out("vashu.txt");
    out<<st;

    return 0;
}
```

READING A FILE

```
#include<iostream>
#include<fstream>

using namespace std;

int main(){
   string st2;

   ifstream in("vashu.txt");
   in>>st2;
   getline(in, st2);
   cout<<st2;

   return 0;
}
```

TEMPLATES

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

template <class t>
class vector {

   public :
   t * arr;
   int size ;

   vector (int m){
      size = m ;
      arr = new t [size];
   }
```

```
    t dotproduct(vector &v ){
        t d=0;
        for (int i=0;i<size;i++ ){
            d+= this ->arr[i]*v.arr[i];
        }
        return d ;
    }

};
int main (){
    vector <float>v1(3);
    v1.arr[0]=4.1;
    v1.arr[1]=3.4;
    v1.arr[2]=3.4;

    vector <float>v2(3);
    v2.arr[0]=1.0;
    v2.arr[1]=0.2;
    v2.arr[2]=1.0;

    int a=v1.dotproduct(v2);
    cout <<a<<endl;


    return 0;
}
```

TEMPLATES WITH MULTIPLE PARAMETERS

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

template<class T1, class T2>
class myClass{
    public:
        T1 data1;
        T2 data2;
        myClass(T1 a,T2 b){
            data1 = a;
            data2 = b;
        }
    void display(){
        cout<<this->data1<<" "<<this->data2;
    }
};
int main()
{
    myClass<int, char> obj(1, 'c');
    obj.display();
}
```

TEMPLATES WITH DEFAULT PARAMETERS

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

template <class T1=int, class T2=float, class T3=char>
class Harry{
    public:
        T1 a;
        T2 b;
        T3 c;
        Harry(T1 x, T2 y, T3 z) {
            a = x;
            b = y;
            c = z;
        }
        void display(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
        }
};
int main()
{
    Harry<> h(4, 6.4, 'c');
    h.display();
    cout << endl;
    Harry<float, char, char> g(1.6, 'o', 'c');
    g.display();
    return 0;
}
```

TEMPLATES IN FUNCTION

```
#include<iostream>
using namespace std;
template<class T1, class T2>

float funcAverage(T1 a, T2 b){
    float avg= (a+b)/2.0;
    return avg;
}

int main(){
    float a;
    a = funcAverage(5,2);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

EXCEPTION HANDLING

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

```
int main(){
    int num , deno , result;

    cout<<"enter numerator and denominator "<<endl;
    cin>>num>>deno;

    try {
        if (deno==0)
        {
            throw deno;
        }
     result = num/ deno;
    }

    catch(int exp){
        cout<<"exception : division with 0 not allowed"<< exp<<endl;
    }
    cout <<"the result is "<<result<<endl;

    return 0;
}
```