Static is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

#include <iostream>

using namespace std;

#include<string.h>

class Account {

public:

int accno;

char name[10];

static float rateOfInterest;

Account(int accno, char\* name)

{

this->accno = accno;

strcpy(this->name,name);

}

void display()

{

cout<<accno<<" "<<name<<" "<<rateOfInterest<<endl;

}

};

float Account::rateOfInterest=6.5;

int main() {

Account a1 =Account(201, "Sanjay");

Account a2=Account(202, "Nakul");

a1.display();

a2.display();

return 0;

}

===========================================================================================

Excercise 1:

#include<iostream>

using namespace std;

class complex

{

int real,img;

public:static const int cnt;

private:static const int cnt1;

public:

complex()

{

cout<<"default is invoked\n";

real=5;

img=5;

// cnt++;//not allowed

}

static int getcnt()

{

return cnt1;

}

};

const int complex::cnt=5;//memory alloaction takes place

const int complex::cnt1=10;

int main()

{

complex c1;

cout<<"no of obj created is "<<complex::getcnt();

cout<<"sizeoof obj is "<<sizeof(c1)<<endl;

cout<<complex::cnt;

//cout<<complex::cnt1;

}

========================================================

Excercise 2:

#include <iostream>

using namespace std;

class Test

{

static int x;

public:

Test() { x++; }

static int getX() {return x;}

};

int Test::x;

int main()

{

cout << Test::getX() << " ";

}

===================================================================

Excercise 3:

#include <iostream>

using namespace std;

class Player

{

private:

int id;

static int next\_id;

public:

int getID() { return id; }

Player() { id = next\_id++; }

};

int Player::next\_id = 1;

int main()

{

Player p1;

Player p2;

Player p3;

cout << p1.getID() << " ";

cout << p2.getID() << " ";

cout << p3.getID();

return 0;

}

Excercise 4:

===========================================================================================================

=============================================================================

Some interesting facts about static member functions in C++

1)Keyword virtual cannot be used with static member functions

2)static member functions cannot have the same name as a non static function that has same argument types

within the class.

3)A static member function can not be declared with const.

#include<iostream>

class Test {

static void fun() const

{ // compiler error

return;

}

};

int main()

{

}

4)'this' keyword is unavailable for static member functions

#include <iostream>

using namespace std;

class Test

{

int x;

public:

Test() {

x=10; }

static int getX()

{

return this->x;

}

};

int Test::x;

int main()

{

cout << Test::getX() << " ";

}

=====================================================================================

Array of Pointer to Strings

==>An array of pointers to strings is a data structure that holds multiple pointers, each pointing to a string. It allows you to store and manipulate a collection of strings in memory.

==>An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

==>An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.

==>In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

==========================================================

#include <iostream>

using namespace std;

int main()

{

char \*names[5] = {"rohan","am","rahul","rohit","rachana"};

for(int i=0;i<=5;i++)

{

//cout << names[i] << endl; //\*(names+i)

// cout << &names[i] << endl; //(names+i)

// cout<<\*(\*(names+i)+0)<<endl;

}

}

=================================================================

Excercise 1:

#include <iostream>

using namespace std;

class Point {

Point() { cout << "Constructor called"; }

};

int main()

{

Point t1;

return 0;

}

Excercise 2:

#include<iostream>

using namespace std;

class Point {

public:

Point() { cout << "Constructor called"; }

};

int main()

{

Point t1, \*t2;

return 0;

}

Excercise 3:

#include<iostream>

using namespace std;

#include<stdlib.h>

class Test

{

public:

Test()

{ cout << "Constructor called"; }

};

int main()

{

Test \*t = (Test \*) malloc(sizeof(Test));

return 0;

}

Unlike new, malloc() doesn’t call constructor.

Excercise 4:

#include<iostream>

using namespace std;

class temp

{

int z;

static int y;

public:

temp();

void show();

void display();

};

int x;

int temp::y;

void temp::show()

{

y++;

}

temp::temp()

{

int z=200;

x=100;

cout<<"value of z is "<<z<<endl;

}

void temp::display()

{

cout<<"value of x is "<<x<<endl;

cout<<"value of y is "<<y<<endl;

}

int main()

{

temp t;

t.show();

t.show();

t.display();

cout<<sizeof(t);

}

Excercise 5:

#include<iostream>

using namespace std;

class item

{

public:

static int x;

int number;

public:

void getdata(int a)

{

number=a;

number++;

x++;

}

void getcount()

{

x=10;

cout<<"value is "<<x<<endl;

}

void getcount\_1()

{

x++;

cout<<"value is "<<number<<endl;

cout<<"value is "<<x<<endl;

}

};

int number=9;

int item::x;

int main()

{

item a;

a.getcount();

a.getdata(100);

a.getcount\_1();

cout<<number<<endl;

cout<<a.number;

}

=======================================================================================

Math library functions

C++ being a superset of C, supports a large number of useful mathematical functions. These functions are available in standard C++ to support various mathematical calculations.

Instead of focusing on implementation, these functions can be directly used to simplify code and programs. In order to use these functions you need to include a header file- <math.h> or <cmath>.

sqrt() This function takes a number as an argument and returns its square root value. The number can not be a negative value.

#include <cmath>

#include <iostream>

using namespace std;

int main()

{

double answer;

answer = sqrt(9);

cout << "Square root of " << " is " << answer;

return 0;

}

If we pass a negative value in the argument domain error occurs and the output will be the Square root of -a, which is -nan.

==============================================================================================================

int abs(int) This function takes an integer number as an argument and returns its absolute value. It means the output will always be positive regardless of the sign of the input.

#include <math.h>

#include <iostream>

using namespace std;

int main()

{

int val1, val2;

/// finding absolute value using

/// abs() function.

val1 = abs(2.7);

val2 = abs(-43);

cout << "abs(22) = " << val1 << "\n";

cout << "abs(-43) = " << val2 << "\n";

return 0;

}

=========================================================================================

double pow(double, double) This function takes one argument as base and another as exponent.

Consider a base 'b' and exponent 'e'.

#include <iostream>

#include<cmath>

using namespace std;

int main()

{

int base=4;

int exponent=2;

int power=pow(base,exponent);

std::cout << "Power of a given number is :" <<power;

return 0;

}

Output:

Power of a given number is :16

=============================================================

C++ Math ceil()

It rounds the value to the nearest integer which is not less than the given value.

For example :

ceil(8.1)=9.0;

ceil(-8.8)=-8.0;

Syntax

Suppose a number is 'x'. Syntax would be:

double ceil(double x);

Parameter

x : It is the value that rounds to the nearest integer.

Return value

It returns the smallest integer value not less than x.

Let's see a simple example by considering the positive value of x.

#include <iostream>

#include<cmath>

using namespace std;

int main()

{

float x=10.5;

std::cout << "Initial value of x is :"<<x;

cout<<'\n';

cout<<"final value of x is :"<<ceil(x);

return 0;

}

==================================================================================

Example 2

Let's see a simple example by considering the negative value of x.

#include <iostream>

#include<cmath>

using namespace std;

int main()

{

float x=-2.2;

std::cout << "Initial value of x is :"<<x;

cout<<'\n';

cout<<"final value of x is :"<<ceil(x);

return 0;

}

Output:

Initial value of x is :-2.2

final value of x is :-2

===============================================

C++ Math floor()

It rounds the value to the nearest integer which is not greater than the given value.

floor(8.2)=8.0;

floor(-8.8)=-9.0;

Syntax

Suppose a number is 'x'. Syntax would be :

double floor(double x);

Parameter

x : It is the value that rounds to the nearest integer.

Return value

It returns the value that round to the nearest integer not greater than x.

==========================================

Example 1

Let's see a simple example by considering a positive value.

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

float x=7.8;

std::cout << "Initial value of x is : " << x<<std::endl;

cout<<"Now, the value of x is :"<<floor(x);

return 0;

}

Output:

Initial value of x is : 7.8

Now, the value of x is :7

Example 2

Let's see a simple example by considering a negative value.

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

float x=-10.2;

std::cout << "Initial value of x is : " << x<<std::endl;

cout<<"Now, the value of x is :"<<floor(x);

return 0;

}

Output:

Initial value of x is : -10.2

Now, the value of x is :-11

=========================================================================

C++ Math round()

This function is used to round off the given value which can be either float or double.

Syntax

Suppose a number is 'x'. Syntax would be:

return\_type round(data\_type x);

Parameter

x : The value which can be either float or double.

Return value

It returns the rounded value of x. The return type of the value can be float, double or long double.

Example 1

Let's see a simple example when the value of x is positive

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

float x=8.4;

std::cout << "The value of x is : " <<x<< std::endl;

cout<<"Rounded value of x is : "<<round(x);

return 0;

}

Example 2:

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

float x=8.6;

std::cout << "The value of x is : " <<x<< std::endl;

cout<<"Rounded value of x is : "<<round(x);

return 0;

}

Example 2

Let's see a simple example when the value of x is negative.

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

double x=-9.9;

std::cout << "The value of x is : " <<x<< std::endl;

cout<<"Rounded value of x is : "<<round(x);

return 0;

}

Output:

The value of x is : -9.9

Rounded value of x is : -10

======================================

===========================================================================================

FAQ ON MULTIPLE INHERITANCE:

1)

#include<iostream>

using namespace std;

class Base1 {

public:

Base1()

{ cout << " Base1's constructor called" << endl; }

};

class Base2 {

public:

Base2()

{ cout << "Base2's constructor called" << endl; }

};

class Derived: public Base1, public Base2 {

public:

Derived()

{ cout << "Derived's constructor called" << endl; }

};

int main()

{

Derived d;

return 0;

}

(A) Compiler Dependent

(B) Base1′s constructor called

Base2′s constructor called

Derived’s constructor called

(C) Base2′s constructor called

Base1′s constructor called

Derived’s constructor called

(D) Compiler Error

==============================================================================

2)

#include <iostream>

using namespace std;

class Base1 {

public:

~Base1() { cout << " Base1's destructor" << endl; }

};

class Base2 {

public:

~Base2() { cout << " Base2's destructor" << endl; }

};

class Derived: public Base1, public Base2 {

public:

~Derived() { cout << " Derived's destructor" << endl; }

};

int main()

{

Derived d;

return 0;

}

(A)

Base1's destructor

Base2's destructor

Derived's destructor

(B)

Derived's destructor

Base2's destructor

Base1's destructor

(C)

Derived's destructor

(D) Compiler Dependent

=========================================================================

3)

#include<iostream>

using namespace std;

class P {

public:

void print() { cout <<" Inside P"; }

};

class Q : public P {

public:

void print() { cout <<" Inside Q"; }

};

class R: public Q { };

int main(void)

{

R r;

r.print();

return 0;

}

(A) Inside P

(B) Inside Q

(C) Compiler Error: Ambiguous call to print()

====================================================================

==========================================================================================

#include<iostream>

using namespace std;

class complex

{

int real,img;

public:

complex(int,int);

void display();

complex operator++();

complex operator++(int);

complex();

};

complex::complex()

{

real=img=0;

}

complex::complex(int real,int img)

{

this->real=real;

this->img=img;

}

void complex::display()

{

if(img>0)

cout<<"complex number is "<<real<<"+"<<img<<"i"<<endl;

else

cout<<"complex number is "<<real<<img<<"i"<<endl;

}

complex complex::operator++()//pre

{

++this->real;//this->real=this->real + 1

++this->img;

return (\*this);

}

complex complex::operator++(int)//post

{

complex temp= (\*this);

++this->real;

++this->img;

return temp;

}

int main()

{

int a=10;

int b;

b=++a; //a=11 b=11

complex c1(1,2);

complex c2=++c1;//c2=c1.operator++()

c1.display();

c2.display();

int a=10;

int b;

b=a++;

//b=10 a=11

/\*complex c1(1,2);

complex c2=c1++;//c2=c1.operator++(0);

c1.display();

c2.display();\*/

}

=================================================================

//overloading of decrement operator

#include<iostream>

using namespace std;

class complex

{

int real,img;

public:

complex(int,int);

void display();

complex operator-();

complex operator--();

complex operator--(int);

complex();

};

complex::complex()

{

real=img=0;

}

complex::complex(int real,int img)

{

this->real=real;

this->img=img;

}

void complex::display()

{

if(img>0)

cout<<"complex number is "<<real<<"+"<<img<<"i"<<endl;

else

cout<<"complex number is "<<real<<img<<"i"<<endl;

}

complex complex::operator--()//pre

{

--this->real;//this->real=this->real + 1

--this->img;

return (\*this);

}

complex complex::operator--(int)

{

complex temp= (\*this);

--this->real;

--this->img;

return temp;

}

int main()

{

complex c1(1,2);

complex c2=--c1;//c2=c1.operator--()

c1.display();

c2.display();

/\*complex c1(1,2);

complex c2=c1--;//c2=c1.operator--(0);

c1.display();

c2.display();\*/

}

==================================================

//OVERLOADING OF = OPERATOR

==>one object can be assigned to another object provided the type of both the objects is same.

when object assignment is done,assignment operator function is called.

==>ex:

cstring s1("info");

cstring s2("tech");

s1=s2;

==>compiler provides a default implementation of the assignment operator for any class which performs memberwise copy of the data members. this mechanism of performing memberwise copy of the contents of the objects is called as shallow copying.

==>while performing memberwise copy,the address in the pointer data member of one object is copied into the pointer data member of another object.This means both the objects point to same memory location.it could lead to following problems:

1)Memory leakage:

2)Dangling pointer:

Therefore for classes with a pointer data member ex: class cstring,shallow copying technique cannot be used.

==>inorder to handle such memory related problems,the assignment operator needs to be overloaded.this enables deep copying

=========================================================================

==>user defined overloaded assignment operator function for the class cstring should take care of memory leakage and dangling pointer problems.steps involved in overloading the assignment operator are as follows:

1)The old buffer is deleted. ex:if object s2 is assigned to s1, first the buffer pointed by s1 is deleted.This prevents memory leakage problem.

2)Enough new memory is allocated on heap for s1.The new memory is equal to the size of buffer pointed by s2.

3)The string is copied to the newly allocated memory .The string of s2 is copied to the newly allocated memory.

step2 and step 3 prevent the problem of dangling pointer.

==>call s1=s2 is resolved as s1.operator=(s2).

s1 is an implicit parameter which is the string to be modified. s2 is passed by reference to avoid copy constructor call.

==>

ex:

cstring s1("info");

cstring s2("tech");

s1=s2;//calls assignment operator

cstring s3=s1;//calls copy constructor

==>If a class contains atleast one data member as a pointer then to avoid memory related problems in the applications it is necessary for the programmer to implement the following:

1)userdefined copy constructor

2)userdefined destructor

3)userdefined assignment operator

#include<iostream>

using namespace std;

#include<string.h>

class string1

{

int len;

char\* ptr;

public:

void display();

string1(char\*);

string1& operator=(string1&);

string1(string1&)

{

cout<<"copy constructor is called";

}

};

string1& string1::operator=(string1& s)

{

cout<<"assignment is called\n";

len=s.len;

delete[] ptr;

ptr=new char[len+1];

strcpy(ptr,s.ptr);

return (\*this);

}

void string1::display()

{

cout<<"length is "<<len<<endl;

cout<<"string is "<<ptr<<endl;

}

string1::string1(char \* sptr)

{

len=strlen(sptr);

ptr=new char[len+1];

strcpy(ptr,sptr);

}

int main()

{

string1 s1("abc");

string1 s2("xyz");

s2=s1;//s2.operator=(s1)

s2.display();

}

/\* string1 s4(s1);//give call to copy constructor

s2.display();

string1 s5=s1;//call to copy constrictor

\*/

============================================================================

#include<iostream>

using namespace std;

class complex

{

int real,img;

public:

complex();

complex(int,int);

void display();

complex operator+(int);

friend complex operator+(int,complex&);

};

complex::complex()

{}

complex::complex(int real,int img)

{

this->real=real;

this->img=img;

}

void complex::display()

{

if(img>0)

cout<<"complex number is "<<real<<"+"<<img<<"i"<<endl;

else

cout<<"complex number is "<<real<<img<<"i"<<endl;

}

complex complex::operator+(int num)

{

complex temp;

temp.real=real + num;

temp.img=this->img + num;

return temp;

}

complex operator+(int num,complex& c)

{

complex temp;

temp.real=c.real + num;

temp.img=c.img + num;

return temp;

}

int main()

{

/\* complex c1(1,2);

complex c2=c1 + 5;//c2=c1.operator+(5)

c2.display();//6+7i\*/

complex c1(1,2);

complex c2=5+c1;// c2=5.operator+(c1) =====> c2=operator+(5,c1)

c2.display();

}

=========================================================================

//overloading of == operator

// C++ Program to overload == operator

#include<iostream>

#include<string>

using namespace std;

class Car{

private:

string name;

int cost;

public:

Car(string n, int c){

name=n;

cost=c;

}

bool operator == ( Car &c){

if (name == c.name && cost == c.cost)

return true;

return false;

}

};

int main(){

Car car1 ("Santro",500000);

Car car2 ("Safari",1000000);

if (car1 == car2) //car1.operator==(car2)

cout<<"Equivalent"<<endl;

else

cout<<"Not Equivalent"<<endl;

}

====================================================================

//overloading of insertion and extraction operator

In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.

1) cout is an object of ostream class and cin is an object of istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of the class, we must make them friend.

Why these operators must be overloaded as global?

In operator overloading, if an operator is overloaded as a member, then it must be a member of the object on the left side of the operator. For example, consider the statement “ob1 + ob2” (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload ‘+’ in a class of ‘ob1’ or make ‘+’ a global function.

The operators ‘<<‘ and ‘>>’ are called like ‘cout << ob1’ and ‘cin >> ob1’. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user-defined class.

Following is a complete C++ program to demonstrate overloading of <> operators.

#include <iostream>

using namespace std;

class Complex

{

private:

int real, imag;

public:

Complex(int r = 0, int i =0)

{ real = r; imag = i; }

friend void operator << (ostream &out, const Complex &c);

friend void operator >> (istream &in, Complex &c);

};

void operator << (ostream &out, const Complex &c)

{

out << c.real;

out <<"+"<< c.imag <<"i"<< endl;

}

void operator >> (istream &in, Complex &c)

{

cout << "Enter Real Part ";

in >> c.real;

cout << "Enter Imaginary Part ";

in >> c.imag;

}

int main()

{

Complex c1;

cin >> c1;//

cout << "The complex object is ";

cout << c1;

return 0;

}

==============================================================================

#include<iostream>

using namespace std;

class Date1

{

int day;

int month;

int year;

public:

Date1(int d,int m,int y)

{

day=d;

month=m;

year=y;

}

friend void operator-(Date1 &,int);

friend void operator+(Date1 &,int);

void display()

{

cout<<"Date:"<<day<<"-"<<month<<"-"<<year<<endl;

}

};

void operator-(Date1 &x, int y)

{

x.day= x.day-y;

x.month=x.month-3;

x.year=x.year-6;

cout<<"Date:"<<x.day<<"/"<<x.month<<"/"<<x.year<<endl;

}

void operator+(Date1 &x,int y)

{

x.day= x.day+y;

x.month=x.month+3;

x.year=x.year+6;

cout<<"Date:"<<x.day<<"/"<<x.month<<"/"<<x.year<<endl;

}

int main()

{

int dd,mm,yy;

cout<<"Enter Day:";

cin>>dd;

cout<<"Enter Month:";

cin>>mm;

cout<<"Enter Year:";

cin>>yy;

Date1 d1(dd,mm,yy);

d1-1;//d1=d1.operator-(1)

d1+2;

return 0;

}

======================================================

#include<iostream>

using namespace std;

class time

{

int hr;

int min;

int sec;

public:

time(int h,int m,int s)

{

hr=h;

min=m;

sec=s;

}

friend time operator-(time &,int);

friend time operator+(time &,int);

void display()

{

cout<<"time:"<<hr<<":"<<min<<":"<<sec<<endl;

}

};

time operator-(time &x,int y)

{

x.hr= x.hr-y;

x.min= x.min-y;

x.sec= x.sec-y;

return x;

}

time operator+(time &x,int y)

{

x.hr= x.hr+y;

x.min= x.min+y;

x.sec= x.sec+y;

return x;

}

int main()

{

int hr,min,sec;

cout<<"Enter hr:";

cin>>hr;

cout<<"Enter Min:";

cin>>min;

cout<<"Enter sec:";

cin>>sec;

time t(hr,min,sec);

t=t-1;

t.display();

return 0;

}

===================================================================

Object oriented concepts:

Data hiding:

==>hiding of the data

==>our internal data should not go outside directly i..e outside person should not access our internal data directly.i.e we should hide our data..

==>ex: u have to check email.to check ur mail. by typing www.gmail.com,ur inbox mailbox wont get open.compulsory u have to provide username and password,ur gmail data is not coming outside directly,we have to do some sort of validation. after validation that data is going to come.

==>ex:ICICI bank account is there. I visit bank, and ask how much account balance, bank people ask,what is ur account number and id proof, then only bank people will provide details.

our main data should not go directly. some sort of authentication is required then only the data should go is called data hiding.

class Account

{

private :

double bal; //by implementing this bal variable as a private , we can implement data hiding

//I want check account balance then how I will check, If u want to balance information //then I will call getBal().

public:

double getBalance()

{

//validation;

if(valid)

return balance;

}

//user has to call public method, within the method ,we have to perform validation,if validation succed,then only we can be able to provide our data.

Advantage:

whereever hiding is there, the biggest advantage is security

so my account balance ,I can only check, other people are not going to check.

======================================================================

Abstraction:

Abstraction means not complete,partially implemented, not completely provided class.

hiding internal implementation,just highlight the set of services what we are offering.

these are the set of services what I m going to offer,but how we internally implemented, plz dont ask due to security reason.

If I am exposing my internal code to any person then where is security to my code.thats why hide the internal code.

ex:If we go to Atm,swipe the card,enter pinnumber,which operation u want to do(withdraw,),

in this process we dont know which code is executing.

If u click withdraw button,internally how this functionality is implemented,which sql query is executing,where the database server is locating ,all this information bank people will never highlited.

bank people thru ATM gui screen,they are going to highlight the set of services what they are offering.

Ex: I want to develop java application,several services are available(withdraw,deposit),

A gui base .net application want to communicate with our java application. from.net application we have to call withdraw(), in the middle i will expose all my set of services, called as interface.

.net person has to call withdraw(),internally how this functionality is implemented we will never going to expose. because of this abstraction/interface concept two different application can communicate with each other without hightlighting our internal application.

ADVANTAGE:

1)Security because outsider dont have idea internally how it is implemented

2)Enhancement:suppose withdraw() is implemented using java, I want to replace with .net functionality,internally I made changes,but gui screen is same.

without affecting enduser ,we can perform any changes in our internal code.

3)Maintainablity and modularity of the application improves because we are hiding our internal implementation.

===============================================================

Encapsulation:

capsule: whatever the medicine require to cure cold,that medicine is going to be encapsulated. its going to be grouped into this capsule. capsule is nothing but unit, all the required things are grouped

into 1 capsule. in the same way.

ex:for every student some data members and functions are der,

the process of encapsulating or grouping of data members and corresponding behaviour into a single unit. that unit itself is called as class

int age;

int rollno;

int marks;

+

read()

write()

student related data + behaviour

Every c++ class is an example of encapsulation.

==>If any component follows data hiding and abstraction, that component is said to be encapsulated component.

Encapsulation=data hiding + Abstraction

class Account

{

private:

double balance;

public:

public double getBal()

{

//validate

return balance;

public void setBalance(double amt)

{

balance=balance + amt;

}

}

i will never highlight this functionalities to outside person directly, I will provide GUI screen to the user. in that balance enq. whenever user click bal enquiry ,functionality will get executed.but enduser dont know,internally which code is executed.

===================================================================================