Q1. Write a program to implement four function calculator that can Add, Subtract, Multiply and Divide two numbers, taken from user as input. Also allow user to select the operation to be performed.

**Lab Session 02**  
**OBJECT**  
***Study of object and classes in Object Oriented Programming***  
**THEORY** Object-Oriented Programming (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. Object-oriented programming is a paradigm in which a software system is decomposed into subsystems based on objects. Computation is done by objects exchanging messages among themselves. The paradigm enhances software maintainability, extensibility and reusability.  
**Figure 2.1**  
**CLASSES AND OBJECTS** Object is the basic unit of object oriented programming. Object represents a Physical/real entity. All real world objects have three characteristics: • State/Attributes: The states of an object represent all the information held within it • Behavior: Behavior of an object is the set of action that it can perform to change the state of the object. • Identity: Object is identified by its unique name Classes are data types based on which objects are created. Objects with similar attributes and methods are grouped together to form a class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.  
**Object Oriented Programming Lab Session 02**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 20 Classes are created using the keyword class. A class declaration is similar syntactically to a structure. The attributes of object are represented by variables or data structures (arrays, list etc). The behavior is specified by defining methods, also known as member functions.  
*class classname*  
*{*  
*access specifier:*  
*data member;*  
*member functions;*  
*:*  
*access specifier:*  
*data member;*  
*member functions;*  
*};* Here, access-specifier is one of the three C++ keywords**:** • public • private • protected By default, functions and data declared within a class are private to that class and can be accessed only by other members of the class. The public access specifier allows functions or data to be accessible to other parts of your program. The protected access specifier is needed only when inheritance is involved. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.  
**Inline and Out of line methods** Member functions defined within the class definition are called inline functions. Member functions which are only declared in the class definition, and defined outside it, are called out-of-line. The implementation code of inline methods is duplicated for every function call. The **inline** functions allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. However, when a function is expanded in line, none of the calling and return mechanism occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.  
**Sample Program**  
#include <iostream>  
using namespace std;  
class BankAccount{  
**Object Oriented Programming Lab Session 02**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 21  
int Account\_no;  
double Balance;  
public:  
void setBalance(double b);  
void setAccno(int no);  
double getBalance();  
int getAccno();  
void display();  
void withdraw(double amount);  
void deposit(double amount);  
};  
void BankAccount::setBalance(double b)  
{ Balance=b;}  
void BankAccount::setAccno(int no)  
{ Account\_no=no;}  
double BankAccount::getBalance()  
{ return Balance;}  
int BankAccount::getAccno()  
{ return Account\_no;}  
void BankAccount::display()  
{ cout<< "Account no is"<< Account\_no << "\n Balance is"<< Balance;}  
void BankAccount::withdraw(double amount)  
{ if(amount>=Balance)  
Balance-=amount;}  
void BankAccount::deposit(double amount)  
{ Balance+=amount;}  
int main()  
{  
BankAccount b1;  
BankAccount b2;  
b1.setAccno(1);  
b1.setBalance(5000.0);  
b2.setAccno(2);  
b2.setBalance(4500.0);  
b1.display();  
b2.display();  
b1.withdraw(500.0);  
b2.deposit(1000.0);  
b1.display();  
b2.display();  
return 0;  
}  
**CONSTRUCTORS** It is very common for some part of an object to require initialization before it can be used. C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function. A constructor is a special function that is a member of a class and has the same name as that class. It is also possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object For example, we can add a parameterized constructor for  
**BankAccount** class.  
#include <iostream>  
using namespace std;  
**Object Oriented Programming Lab Session 02**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 22  
Class BankAccount{  
int Account\_no;  
double Balance;  
public:  
void setBalance(double b);  
void setAccno(int no);  
double getBalance();  
int getAccno();  
void display();  
void withdraw(double amount);  
void deposit(double amount);  
//constructor declaration  
BankAccount(int Accno, double Bal);  
};  
// constructor definition  
BankAccount::BankAccount(int Accno, double Bal)  
{  
Account\_no=Accno;  
Balance=Bal;  
}  
// get, set method definitions  
int main()  
{  
BankAccount b1(1,2000.0);  
BankAccount b2(2, 1000.0);  
b1.display();  
b2.display();  
b1.withdraw(500.0);  
b2.deposit(1000.0);  
b1.display();  
b2.display();  
return 0;  
} Even in the presence of parameterized constructor, you may still need set methods to edit the attributes of an object.  
**Constructor with Default Arguments**  
It’s possible to defined constructor with default arguments. For e.g. BankAccount( ) can be declared as  
BankAccount(int Accno, double Balance=500.0 ); The default value of the argument **Balance** is **zero**. Then, the statement BankAccount B(1); assigns the value 1 to the **Accno** variable and 500.0 to **Balance** (by default).  
**Copy Constructor** A copy constructor is used to declare and initialize an object from another object. For e.g. the following statement would define the object b2 and at the same time initialize it to b1  
BankAccount b2(b1); // b1 is an object of BankAccount type The process of initializing through a copy constructor is known as *copy initialization.* A copy constructor takes a reference to an object of the same class as itself as an argument. Here is the code for copy constructor of BankAccount class  
BankAccount::BankAccount(BankAccount& b)  
{  
Account\_no=b.Account\_no;  
Balance=b.Balance  
**Object Oriented Programming Lab Session 02**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 23  
}  
**DESTRUCTOR** A destructor as name implies, is used to destroy the objects that have been created by a constructor. Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by tilde. For Example, the destructor for the class BankAccount can be defined as shown below ~BankAccount() { } A destructor never takes any argument nor does it returns any value. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. If you want to perform some specific task when an object  
is destroyed, you can define the destructor in the class. If you don’t define, the default one is called.  
**FRIEND FUNCTIONS** It is possible to grant a nonmember function access to the private members of a class by using a friend. A friend function has access to all private and protected members of the class for which it is a friend. To declare a friend function, include its prototype within the class, preceding it with the keyword friend. Consider this program:  
#include <iostream>  
using namespace std;  
class myclass {  
int a, b;  
public:  
friend int sum(myclass x);  
void set\_ab(int i, int j);  
};  
void myclass::set\_ab(int i, int j)  
{  
a = i;  
b = j;  
}  
// Note: sum() is not a member function of any class.  
int sum(myclass x)  
{ /\* Because sum() is a friend of myclass, it can directly access a and  
b. \*/  
return x.a + x.b;  
}  
int main()  
{  
myclass n;  
n.set\_ab(3, 4);  
cout << sum(n);  
return 0;  
} In this example, the sum( ) function is not a member of myclass. However, it still has full access to its private members. Also, notice that sum( ) is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.  
**Exercise**  
**Object Oriented Programming Lab Session 02**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 24 Q1.What is the difference between object and a class? Q2. What is the criterion for defining a functions **inline** or **out of line**? Q3. Define a class **Complex\_No** that has two member variables; **Real** and **Imaginary**. Also include following in the class • A parameterized constructor that takes Real and Imaginary values as argument. • A default constructor that assign zero to Real and Imaginary. • A copy constructor • A method **Display** that shows the value of complex number in appropriate format. • A method **Magnitude** that calculates the magnitude of complex number • A method **Add** that adds two complex numbers and return result; take one complex number as argument. Write a driver program to test your class

Q4. Define a class **Counter** having an attribute **value**. Provide a constructor that initializes value to zero. Also provide following methods:

• Increment (): that increment the value by one.

• Decrement (): that decrement the value by one.

Q5. Define a function **Reset** that takes a **Counter** type object as input and resets its **value** to zero. Make this function a friend of **Counter** class

Q6. Define a class **Student** that has following attributes:  
**Name**: allocated dynamically by a character pointer.  
**Rollno**: an integer.  
**Marks**: a double type array of 5 elements.  
**Percentage:** a float Include a constructor that takes values of Name, Rollno and Marks from user as input. Also include following methods:  
**CalculatePercentage:** that adds all 5 elements of array Marks and calculate percentage according to formula Percentage = (Total marks/ 500 )\*100 and stores result in member variable Percentage.  
**Grade**: that calls CalculatePercentage method and displays the grade accordingly Write a driver program to test your class.

**Object Oriented Programming Lab Session 03**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 25  
**Lab Session 03**  
**OBJECT**  
***Working with arrays of objects, pointers to objects and dynamic allocation of objects in***  
***C++***  
**THEORY**  
**Arrays Of Objects** In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. The general format is:  
*classname ob[3]; // declaring array of objects of* ***classname***  
*ob[i]. functionname(); // calling member function using object of array*  
***Initializing Arrays of objects*** If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax  
classname ob[3] = {1, 2, 3}; // initializing one-variable class; short form  
classname ob[3] = { cl(1),cl(2),cl(3) };//initializing one-variable class; long form As each element in the array is created, a value from the list is passed to the constructor's parameter. The short form is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument. Thus, the short form can only be used to initialize object arrays whose constructors require one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example:  
classname ob[3] = {cl(1,2),cl(3,4),cl(5,6)}; *//initializing two-variable class*  
**Pointers to Objects** Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class, given a pointer to an object, use the arrow (–>) operator instead of the dot operator. The following program illustrates how to access an object given a pointer to it:  
#include <iostream>  
using namespace std;  
class cl {  
int i;  
public:  
cl(int j) { i=j; }  
int get\_i() { return i; }  
**Object Oriented Programming Lab Session 03**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 26  
};  
int main()  
{  
cl obj(88), \*p;  
p = &obj; // get address of obj  
cout << p->get\_i(); // use -> to call get\_i()  
// initializing array of objects  
cl objarray[3] = {1, 2, 3};  
int i;  
p = objarray; // get start of array  
for(i=0; i<3; i++) {  
cout << p->get\_i() << "\n";  
p++; // point to next object  
}  
return 0;  
} when a pointer is incremented, it points to the next element of its type. The same is true of pointers to objects. The above program uses a pointer to access all three elements of array  
**objarray** after being assigned **objarray**'s starting address.  
***Initializing Pointer Variables*** C++ does not automatically initialize variables. Pointer variables must be initialized if you do not want them to point to anything. Pointer variables are initialized using the constant value 0, called the **null pointer**. The statement p = 0; stores the null pointer in p, that is, p points to nothing. Some programmers use the named constant NULL to initialize pointer variables. The following two statements are equivalent: p = NULL; p = 0; The number 0 is the only number that can be directly assigned to a pointer variable*.*  
**Dynamic Allocation** C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:  
*p\_var* = new *type*;  
delete *p\_var*; Here, *p\_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed. Here is a short program that creates a class called **balance** that links a person's name with his or her account balance. Inside **main( )**, an object of type **balance** is created dynamically.  
#include <iostream>  
#include <cstring>  
using namespace std;  
**Object Oriented Programming Lab Session 03**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 27  
class balance {  
double cur\_bal;  
char name[80];  
public:  
balance(double n, char \*s) {  
cur\_bal = n;  
strcpy(name, s);  
}  
~balance() {  
cout << "Destructing ";  
cout << name << "\n";  
}  
void get\_bal(double &n, char \*s) {  
n = cur\_bal;  
strcpy(s, name);  
}  
};  
int main()  
{  
balance \*p;  
char s[80];  
double n;  
p = new balance(12387.87, "Ralph Wilson");  
p->get\_bal(n, s);  
cout << s << "'s balance is: " << n;  
cout << "\n";  
delete p;  
return 0;  
} You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructors, one will be parameter less. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program. To allocate an array of object of class **balance**, modify the definition of class **balance** as follows**.**  
#include <iostream>  
#include <cstring>  
using namespace std;  
class balance {  
double cur\_bal;  
char name[80];  
public:  
balance(double n, char \*s) {  
cur\_bal = n;  
strcpy(name, s);  
}  
balance() {} // parameterless constructor  
~balance() {  
cout << "Destructing ";  
cout << name << "\n";  
}  
void set(double n, char \*s) {  
cur\_bal = n;  
strcpy(name, s);  
}  
void get\_bal(double &n, char \*s) {  
n = cur\_bal;  
strcpy(s, name);  
**Object Oriented Programming Lab Session 03**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 28  
}  
};  
int main()  
{  
balance \*p;  
char s[80];  
double n;  
int i;  
p = new balance [3]; // allocate entire array  
// note use of dot, not arrow operators  
p[0].set(12387.87, "Ralph Wilson");  
p[1].set(144.00, "A. C. Conners");  
p[2].set(-11.23, "I. M. Overdrawn");  
for(i=0; i<3; i++) {  
p[i].get\_bal(n, s);  
cout << s << "'s balance is: " << n;  
cout << "\n";  
}  
delete [] p;  
return 0;  
}  
**Exercise**

Q1. Write a program that takes the record of 10 students from user in an array and display all the records.  
[use Student class, defined in lab session 02]

Q2.Define a pointer to student class to access the contents of array defined in Q1. Allow user to search a record in array by means of Rollno

Q3. Develop a class to represent an integer array. The member variables include an integer to represent the size of array and an integer pointer to represent the address of the first element of the array. The user is allowed to create an array at runtime using this class. Include appropriate constructors (parameterized and copy). Also include a method that adds the contents of array.

Q4.Consider following code:  
class myclass  
{int data[2];  
public:  
int\* p;  
public:  
myclass()  
{p=data;}  
};  
int main()  
{ myclass\* cp;  
cp=new myclass[3];  
return 0;  
} How would you access the contents of *data* of each element of *myclass* array? Add code in the above program to do the following:  
a. Assign values to array *data* of each element of *myclass* array.

b. Display contents of *data* of each element of *myclass* array

**Object Oriented Programming Lab Session 04**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 30  
**Lab Session 04**  
**OBJECT**  
***Study of Inheritance in Object Oriented Programming***  
**THEORY** Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class. In keeping with standard C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved. When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form: class derived-class-name : access base-class-name { // body of class }; The access status of the base-class members inside the derived class is determined by access.  
**Access Type Description** Private All public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. Protected All public and protected members of the base class become protected members of the derived class. Private All public and protected members of the base class become private members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.  
**Sample Program**  
#include <iostream>  
using namespace std;  
class base {  
protected:  
int i, j;  
public:  
void set(int a, int b) { i=a; j=b; }  
void show() { cout <<"i="<< i << " j=" << j << "\n"; }  
};  
// i and j inherited as protected.  
class derived1 : public base {  
int k;  
**Object Oriented Programming Lab Session 04**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 31  
public:  
void setk() { k = i\*j; } // legal  
void show() {// overriding base class method show()  
base::show(); // calling base class method  
cout << "k="<< k << "\n";  
}  
};  
// i and j inherited indirectly through derived1.  
class derived2 : public derived1 {  
int m;  
public:  
void setm() { m = i-j; } // legal  
void show() {  
derived1::show();// calling derived1 method  
cout << "m="<< m << "\n";  
}  
};  
int main()  
{  
base ob;  
derived1 ob1;  
derived2 ob2;  
ob.set(1,2);  
ob.show();// calling base class method  
ob1.set(2, 3);  
ob1.setk();  
ob1.show();// calling derived1 method  
ob2.set(3, 4);  
ob2.setk();  
ob2.setm();  
ob2.show(); // calling derived2 method  
return 0;  
} In the above example, if base were inherited as private, then all members of base would become private members of derived1, which means that they would not be accessible by derived2.  
**Function Overriding** A derive class can redefine a method, already defined in its base class. This is called function overriding, as shown in above program. When a method is called by reference of a derived class object, compiler first search it in derived class, if found, compiler executes it and if not found, the compiler will execute the base class method. In the above program, derived1 and derived2 classes override the method show(), already defined in their base classes.  
**Order of Execution of Constructors and Destructors**  
When an object of a derived class is created, the base class’ constructor will be called first,  
followed by the derived class’ constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor i.e. constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation. Following Program demonstrates this.  
#include <iostream>  
**Object Oriented Programming Lab Session 04**  
***NED University of Engineering & Technology – Department of Computer & Information Systems Engineering*** 32  
using namespace std;  
class base {  
public:  
base() { cout << "Constructing base\n"; }  
~base() { cout << "Destructing base\n"; }  
};  
class derived1 : public base {  
public:  
derived1() { cout << "Constructing derived1\n"; }  
~derived1() { cout << "Destructing derived1\n"; }  
};  
class derived2: public derived1 {  
public:  
derived2() { cout << "Constructing derived2\n"; }  
~derived2() { cout << "Destructing derived2\n"; }  
};  
int main()  
{  
derived2 ob;  
// construct and destruct ob  
return 0;  
}  
**OUTPUT**  
Constructing base  
Constructing derived1  
Constructing derived2  
Destructing derived2  
Destructing derived1  
Destructing base  
**Exercise**

Q1. If A is derived from B and B is derived from C and an object of class A is created. What will be the sequence of constructor execution? What will be the sequence of destructor execution?

Q2. What would you do if you want the public methods of base class to be accessible by the derived class, but not by objects of the derived class? Mention syntax.

Q3. Extend the **Student** class( defined in lab session 02) to represent science and arts student. Science student has additional practical course of 150 marks while arts student has an optional course of 100 marks. In each class redefine **CalculatePercentage** method.  
Science student: Percentage =Total marks/650\*100   
Art student: Percentage=Total marks/600\*100

Include constructor and display method in each class. Write a driver program to test your classes.

Q4. Develop a class **Post** that has following attributes  
**Name**: a string  
**To** : a string that holds the reciever's address  
**StampCost**: a float that holds the value of postal stamp required The class should include following:

• A constructor that initializes StampCost to 1$ and To to empty address

• **Read():** a method that reads data member's values from user • **Print():** a method that displays data member's values on screen

• **TotalCost():** a method that returns stampCost value.

Develop another class **RegisteredPost** that inherits from **Post** class and has following additional attributes:  
**Weight:** a float that holds the weight of post  
**ReistrationCost:** a float that holds registration charges

Also include following in the class.

• A constructor that initializes weight to 20 gms and RegisterationCost to 10$

• **Read():** a method that reads data member's values from user • **Print():** a method that displays data member's values on screen

• **TotalCost():** a method that returns stampCost+RegistrationCost.

Q5. Develop a class **InsuredRgisteredPost** that inherits from **RegisteredPost** class and has additional attribute **AmtInsured** to hold the insured value of post. The class should include following:

• A constructor that initializes AmtInsured to 20$

• **Read():** a method that reads data member's values from user • **Print():** a method that displays data member's values on screen

• **TotalCost():** a method that returns StampCost+RegistrationCost+AmtInsured.

Q6. Use following driver program to test your classes  
#include <iostream>  
using namespace std;  
int main()  
{ InsuredRegisteredPost envelope;  
envelope.Read();  
cout<<"Post Details..";  
envelope.Print();  
return 0;  
}

Q7. Define classes to represent following products available in a shopping mall.

• **Utility Items** having attributes **name**. **product id** and **discount**

• **Food Items** having attributes **name, product id** and **date of expiry** Also define a method to show data **in** each class.

Write efficient code for handling constructors and methods using