

**Aurora's pg college
Moosarambagh
Mca Department**

**PROGRAMMING IN DATA STRUCTURES THROUGH C++
Lab manuals**

For

MCA IST YR ISEM

LABORATORY MANUAL CONTENTS

Lab Objective

- At the end of the course students should be familiar with the main features of the C++ language.
- Be able to write a C++ program to solve a well specified problem.
- Understand a C++ program written by someone else.
- Be able to debug and test C++ programs;
- Understand how to read C++ doc library documentation and reuse library code.
- To make the students understand the features of object oriented principles and familiarize them with virtual functions, templates and exception handling.
- To make the students to develop applications using C++.

Programmers Educational Objectives

Graduates will be able to

- To analyze, design and provide optimal solution for Master of Computer Applications and multidisciplinary problems.
- To pursue higher studies and research by applying knowledge of mathematics and fundamentals of computer science.
- To exhibit professionalism, communication skills and adapt to current trends by engaging in lifelong learning.

SUBJECT INDEX

Lab No.	Index	Week involved
1	Introduction to OOP lab (Simple C++ program)	1-2
2	Classes and Objects	5-6
3	Constructors and Destructors Write a program to demonstrate different types of constructors.	7-8
4	Operator overloading Write a program for overloading various unary operators.	9-10
5	Write a program for overloading various binary operators	10-11
6	Type Conversion Write a program for type conversion (basic to class, class to basic ,class to class)	11-12
7	Inheritance Write a program for multiple inheritance Write a program for hybrid inheritance	13-14
8	Polymorphism Write a program for polymorphism(virtual function)	15-16
9	Write a program for templates	17-18
10	Program using files	19-20
11	Program using streams	21-22
12	Program using Arrays and Linked list	22-25
	Programs on stack and Queues	25-30

Appendix - A

Appendix - B

Experiment No. 1

Title: Introduction to the Fundamentals and history of OOP Concepts

Objective:

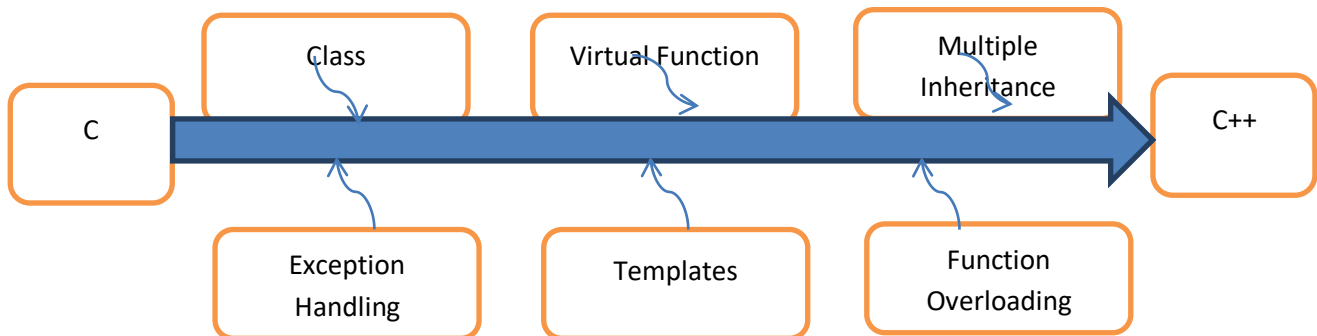
At the end of this experiment, students should be able to understand following points:

1. Basic concepts of c++ like insertion,extraction operator
2. Different operators
3. Array,String,Function
4. Basic object oriented concepts

Theory:

Introduction

Object oriented language was developed by Bjarne Stroustrup in 1979 at Bell Labs. As an enhancement to the C programming language and originally named "*C with Classes*". It was renamed to C++ in 1983. C++ is designed to be a statically typed general-purpose language that is as efficient and portable as C. C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming) C++ avoids features that are platform specific or not general purpose

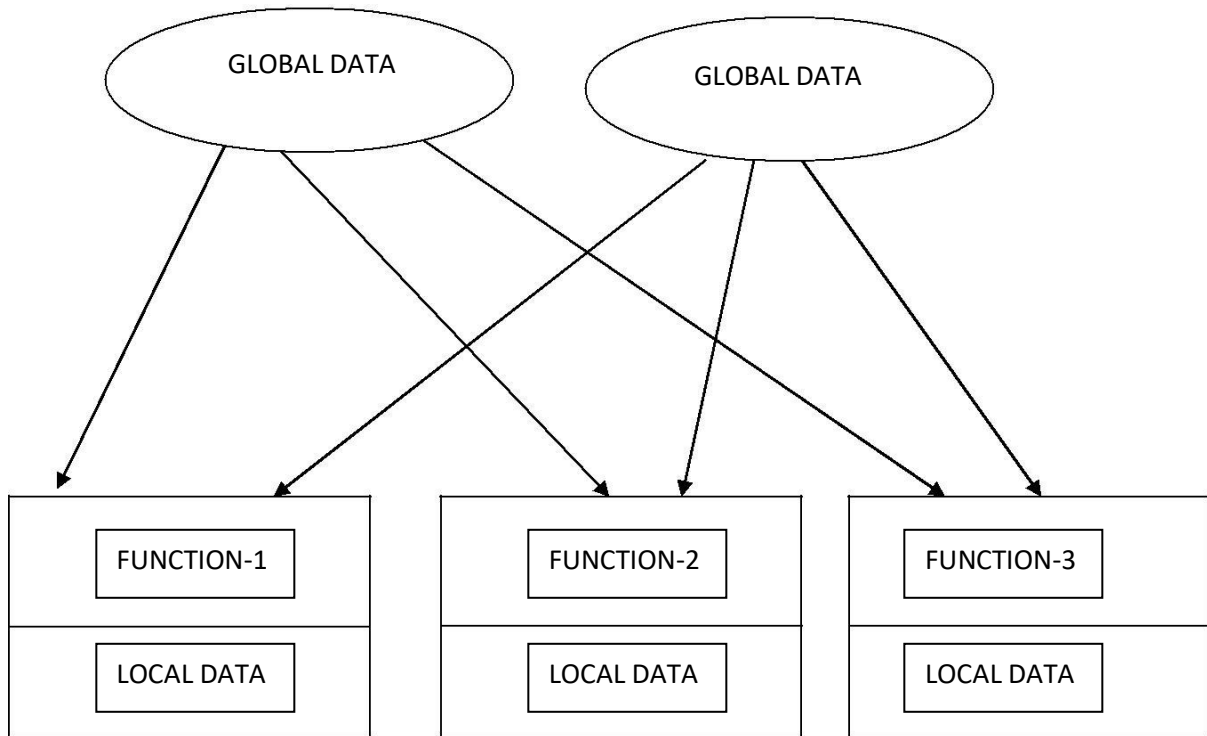


A LOOK AT PROCEDURE ORIENTED LANGUAGES

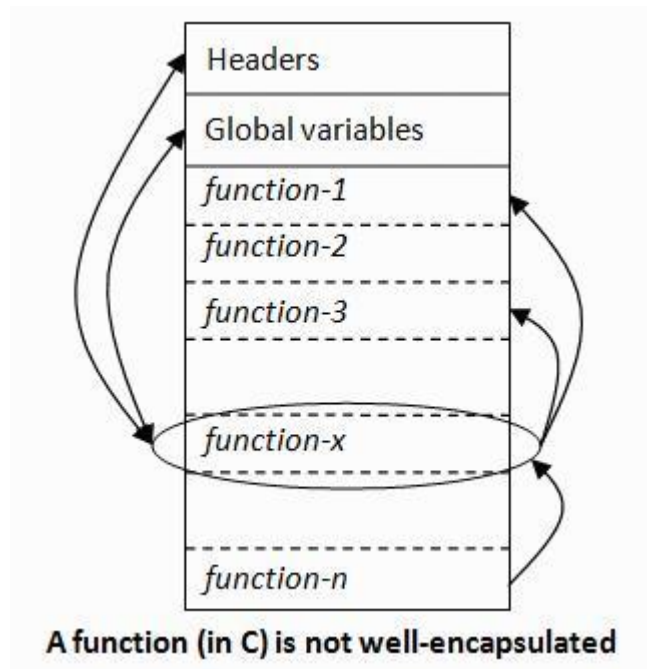
C, PASCAL, FORTRAN commonly known as procedure-oriented programming (POP). That is, each statement in the language tells the computer to do something. A program in the procedural language is a list of instructions. The program creates the list of instructions and the computer carries them out.

Object Oriented	Procedural
Methods	Functions
Objects	Modules
Message	Argument
Attribute	Variable

In multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Fig shows the relationship of data and functions in a procedure

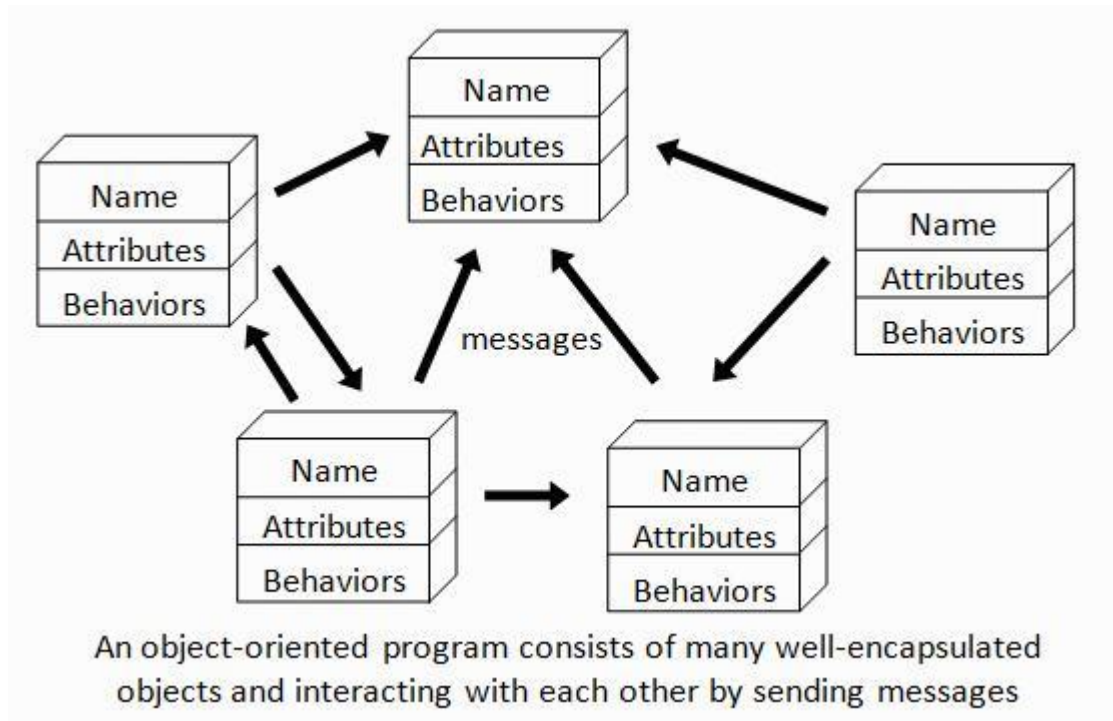


Relationship of data and functions in procedural language



OBJECT ORIENTED LANGUAGE

The fundamental idea behind the object oriented language is to combine into a single unit of data and functions that operate on that data. Such unit is called an object. **Object-oriented programming (OOP)** is a programming paradigm that uses "objects" – data structures consisting of datafields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as information hiding, data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s.¹ Many modern programming languages now support OOP.



BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

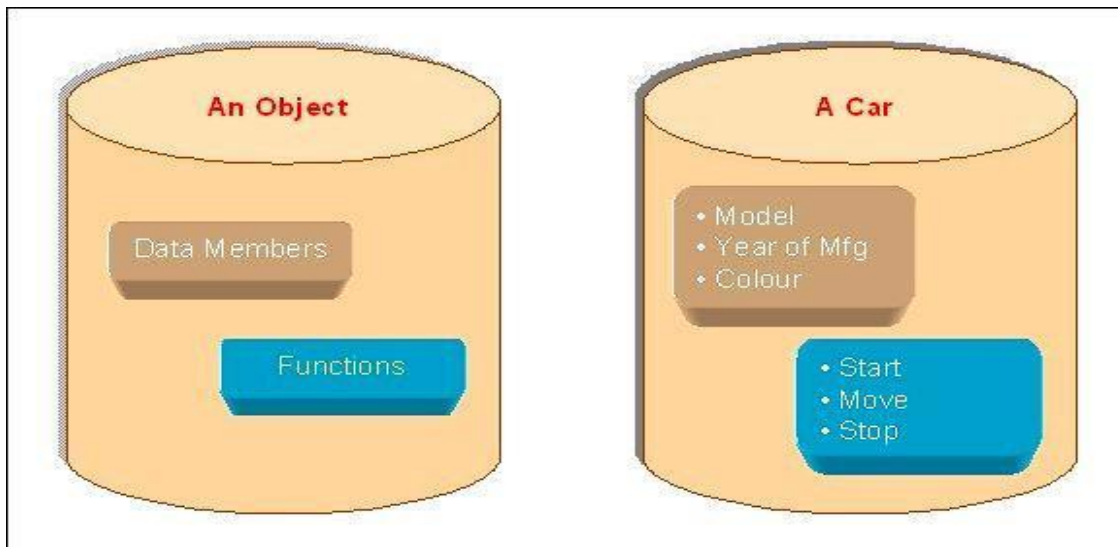
Object

An object is a basic run time entity. Object represents/resembles a Physical/real entity.

An object is simply something you can give a name. All the objects have some characteristics and behavior. The states of an object represent all the information held within it and behavior of an object is the set of action that it can perform to change the state of the object. All real world objects have three characteristics:

- State: How object react?
- Behavior: what we can do with this object?
- Identity: difference between one object to another object?

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

Class

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represent a set of individual objects. Characteristics of an object are represented in a class as **Properties**. The actions that can be performed by objects become functions of the class and is referred to as **Methods**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Color, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, and Stop form the **Methods** of *Car* Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Instance

One can have an instance of a class or a particular object. The instance is the actual object created at runtime.. The object consists of state and the behaviour that's defined in the object's class.

Data hiding:

This is the property in which some of the members are restricted from Outside access. This is implemented by using private and protected access specifiers.

Message passing

"The process by which an object sends data to another object or asks the other object to invoke a method." Also known to some programming languages as interfacing

Inheritance

Inheritance is the process of forming a new class from an existing class or *baseclass*. The base class is also known as *parent class* or *super class*, the new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class*. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

Basics of C/C++ Programming

C++ is a high level language with certain low-level features as well. Remember that C++ is a **case-sensitive** language. A C++ program is actually a collection of statements and data on which various operations can be performed. C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, most all C programs are also C++ programs. However, there are a few minor differences that will prevent a C programs to run under C++ compiler .the object oriented features in C++ allow programmers to build large programs with clarity , extensibility and ease of maintenance incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down structured design, to one that provides bottom-up, object-oriented design. Kindly refer to *listing 1.1* to see what a simple C++ program looks like.

Comments

C++ introduces a new comment symbol //(double slash).A comment may start anywhere in the line. Note that there is no closing symbol

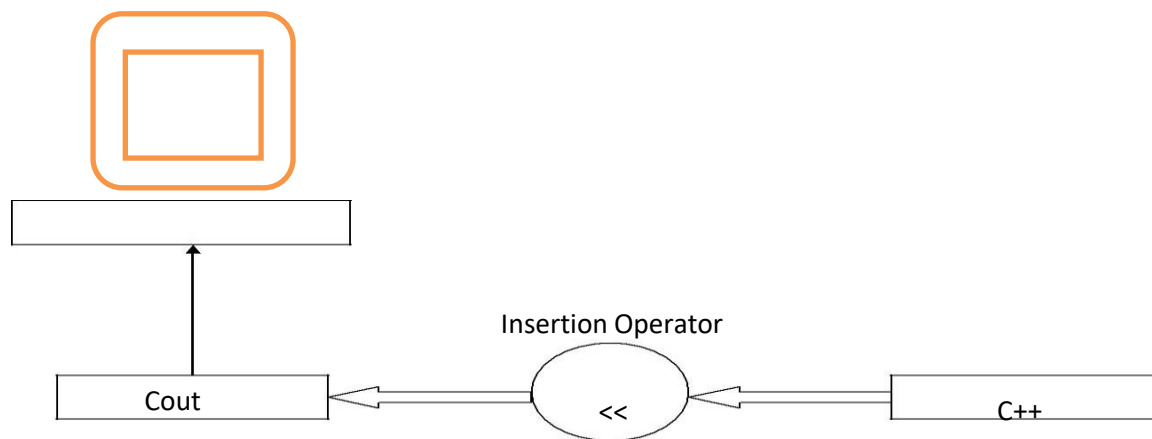
//This is an example of C++ //
program

The C comment symbol /* */ are still valid for multi line comments. /*
this is an example of C++ */

Output Operator

```
cout<<"C++ is better than C";
```

Causes the string quotation marks to be displayed on the screen. This statement introduces new C++ features, cout and <<. The identifier cout is a predefined object that represents output stream in C++. The standard output stream represents the screen. The operator << is called the insertion operator. It inserts (or sends) the contents of the variable on its right to the object on its left. It is same as printf() in C++.



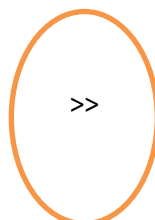
Output using insertion operator

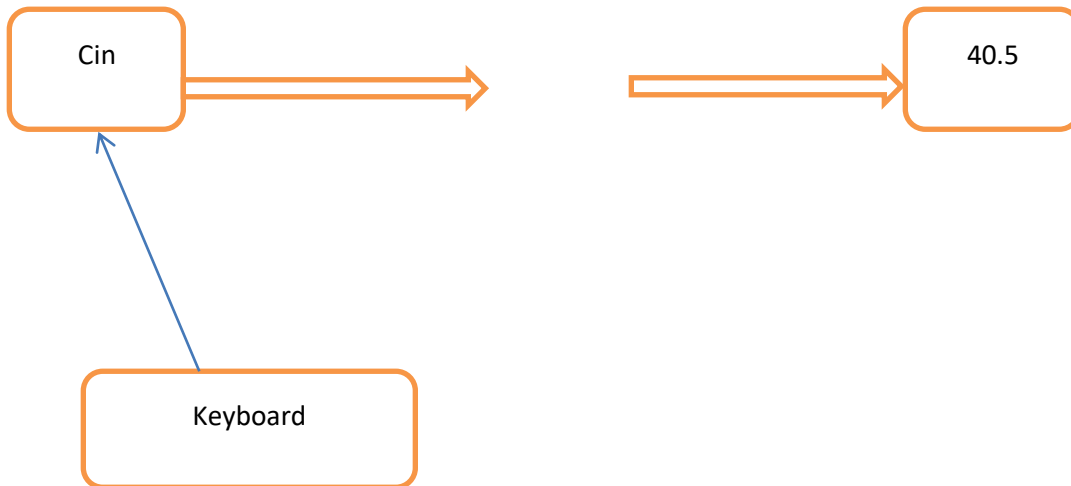
Input Operator

The statement **cin<<num1;** is an input statement and causes program to wait for the user to type in number. The identifier cin is a predefined object in C++ that corresponds to the standard input stream. Here stream represents the keyboard. The operator >> is known as extraction or get from operator.

Object

Variable





Input

using extraction operator

Data Types

Following are the few basic data types used in C++.

Name	C++	Describes
Integer	Int	numeric data in the range of -32768 to 32768
Floating-Point	Float	floating-point numeric data in the range 8.43×10^{-37} to 3.37×10^{38}
Double	double	floating-point numeric data in the range 2.225×10^{-308} to 1.7976×10^{308} ;
Character	Char	character specified by character codes -128 to 127
Boolean	Bool	has only two values, either true (1) or false (0)
Void	Void	A non existent value

Relational Operators

C++ also uses some relational operators to perform comparison of different values.

Some of these are:

Operation	Operator
-----------	----------

Equal to	==
Not equal to	!=
Less than	<
Less than or equal to	<=
Greater than or equal to	>=
Greater than	>

Type Qualifiers Name C

Name	C++	Describes
Long Form	long	It requests a long form of an item. Can be used with both int and double
Short Form	short	It requests a short form of an item. Can be only used with int and not double
Signed Number	signed	It describes a variable from its maximum negative to its maximum positive value
Unsigned Number	unsigned	It describes a variable from 0 to a maximum positive value. Valid only with int and char data types
Constant Value	const	It describes value of a variable to be unchangeable

Operators

C++ has a variety of operators to perform various tasks. You came across a few in the previous lab and a few new operators will be discussed in this lab.

Logical Operators

Logical Operators are used to perform logical operations on data. These operators are typically useful to see whether certain conditions are satisfied or not. Logical Operations used in C++ are:

x	y	Ans
---	---	-----

And (&&)

This operator is used to evaluate an expression for logical AND operation. The truth table on the right explains what a logical And (&&) really means.

1	1	1
1	0	0
0	1	0
0	0	0

Example:

```

If (a>b && a>c)
{
cout<<"a is greater than both b & c";
}

```

Or (||)

This operator is used to evaluate an expression for logical OR operation. The truth table on the right explains what a logical and (&&) really means

x	y	Ans
1	1	1
1	0	1
0	1	1
0	0	0

.Example:

```

If (a>b || c>b)
{
cout<<"Either a or c or both are greater than b";
}

```

Negation (!)

This operator is used to evaluate an expression for logical negation operation. Also, there is an operator for the condition "Not equal to" (!=). An example can be:

```

If (!(a>b) && c!=b)
{
cout<<"here a is not greater than b, and c is not equal to b";
}

```

Increment & Decrement (++ and --)

These operators are used to increment or decrement value of a variable. For Example:
a++; //This is same as a=a+1;


```
a - -; //This is same as a=a -1;
```

Assignment (=) and Compound Assignemnts (operator=)

```
A = 1; //Simple assignment operator
```

If a mathematical operator is used in conjunction with the assignment operator we can make the

code better.

```
A += 1; //Same as A=A+1;
```

ARRAYS

An array is like a list or table of any data type. We use arrays for a variety of programming tasks especially when we have to make a list of the same type of data.

UNIDIMENSIONAL ARRAYS

An array with a single dimension is like a list. That is how we define such arrays:

```
int list[10]; //Defining a list of 10 integers
```

This statement actually means that we are declaring a list of arrays from 0 to 9. This means that the starting array element will be referred to as **list[0]** and the last element will be referred to as

list[9].

Elements of this array can be referenced as:

```
list[2]=20;
```

```
list[5]=30;
```

```
cout<<list[2]<<"\n";
```

```
cout<<list[5];
```

Elements of an array can be initialized at the time of its declaration.

```
int list[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; //Defining an array of ten integers
```

MULTIDIMENSIONAL ARRAYS

An array can also be multidimensional. To define a multidimensional array, we follow a similar approach:

```
int list[3][3]; //Define a 3x3 matrix or table
```

To reference various elements of this array, similar approach is used. For example:

```
list[1][2] = 34; //Assigning 34 to row 1 and column 2
```

```
list[0][2] = 20; //Assigning 20 to row 0 and column 2
```

Like the one-dimensional array, we can also initialize a multidimensional array like:

```
int list[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } }; //Initializing a 3x3 matrix
```

You can write the same thing more elegantly as:

```
int list[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

```
1 2 3 4 5 6 7 8 9
```

The initialized matrix will be like this

A text string in C++ is nothing but a one-dimensional array of characters terminated by a

special character „\0“. Let’s see what we can do with string in C++;

DEFINING A STRING

A string can be defined in a similar fashion as an array. That’s how we define a string:

```
char str[30]; //Defining a string of 30 characters.
```

Similarly to access individual characters of a string, we use the following syntax:

```
str[0] = „U“;
str[1] = „m“;
str[2] = „a“;
cin>>str[3];
cout<<str[3];
```

We can also initialize a string while declaring it. This is how we do it: `char str[] = { „a“, „b“, „c“, „\0“ };` //Defining a string initialized to “abc” making it more simple by:

```
char str[] = “abc”; //Defining a string initialized to “abc”
```

To help us in string manipulation, we include two more header files:

string.h

stdio.h

SOME INTERESTING OPERATIONS ON STRINGS

Following are a few interesting string manipulation functions defined in string.h. These functions make the life of a C++ programmer a lot easier and are a part of the standard C++ library. To assign some text to a string, we can use the following function:

```
strcpy ( char dest[ ], char source[ ] )
```

Similarly, to concatenate two strings we use the following function:

```
strcat ( char str1[], char str2[] )
```

To compare two strings, we use the function:

```
strcmp ( char str1[], char str2[] )
```

If this function returns 0, that means both strings are same. find the length of the string, we use the function:

```
strlen(str[]);
```

FUNCTIONS

Basic format of a function prototype:

```
return_type function_name(argument_list)
```

Similar to the concept of a black box, function can be treated like a black box where you give some input to that box and request it to perform something in order to obtain the desired output. Hence, as the creator of the function, one should be careful of what you are creating. Avoid from making careless mistakes in providing the return and the arguments lists.

Common mistakes done by students that can cause syntax and execution errors are:

1. Not passing the correct values for the parameter list. For example: when a function requires a pointer as argument, then you have to place „&“ operator before the variable name, which means you are passing the address of the variable in calling the function. So be sure of writing the correct syntax for such a purpose.
2. Providing a return type in function header/prototype but not defining it inside function definition.
3. Forgetting to place semicolon at the end of function header/prototype.
4. The function call and function prototype do not match to each other.

Some important terminologies

These are some terminologies that you normally encounter in class, books or

website. By referring to the function given below, **Table 3** gives the associated example with its corresponding terminology.

Example function:

```
1 double square (double side)
2 {
3     return side*side;
4 }
```

Terminologies	Example
Function header/signature	Line 1
Function prototype	double square (double)
Function definition	Line 1 to 4
Function call	double a=square(5.6);
Function parameter	double side

Simple C++ Program

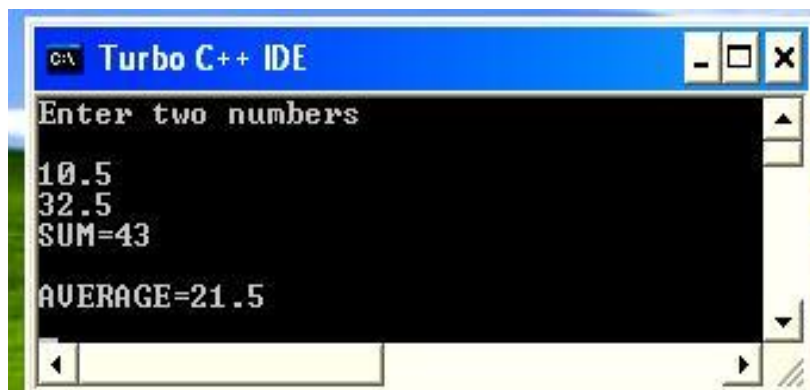
```
#include<iostream.h>
int main()
{
    float num1,num2,sum,avg;
    clrscr();
    cout<<"Enter two numbers\n\n"; //output statement

    cin>>num1; //input statement
    cin>>num2;
```

```
sum=num1+num2 ;
avg=sum/2 ;

cout<<"SUM="<<sum<<"\n\n" ;
cout<<"AVERAGE="<<avg<<"\n" ;
getch() ;
return 0 ;
}
```

Output



Conclusion:

OOP provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has a clearly defined interface. C++ is a versatile language for handling very large programs

Experiment No.2

Title: Write a program which demonstrates Classes and objects.

Objective:

At the end of this experiment, students should be able to understand following points:

1. Class and object.
2. Crating class and object.
3. Defining function in different ways.
4. Concept of Access Specifier like private, public etc.

Theory:

An Overview about Objects and Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an *object*. A *class* is an extended concept similar to that of *structure* in Cprogramming language, this class describes the data properties alone. In C++ programming language, *class* describes both the properties (data) and behaviors (functions) of objects. *Classes* are not *objects*, but they are used to instantiate *objects*.

A typical C++ program would contain four sections as shown in the Fig. 2.1. These sections may be placed in separate code files and then complied independently.

Include Header Files
Class Declaration
Member Function Defination
Main Function Program

Features of Class:

Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, these units of objects make up a class

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol { is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data. Then the class is closed with a flower brace symbol } and concluded with a colon;.

```
class class_name
{
    data;
    member functions;
    .....
};
```

There are different access specifiers for defining the data and functions present inside a class.

Access specifiers:

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

- **private** : A private member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.
- **public** : Public members are accessible from outside the class.

- **protected:** A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class. .

When defining access specifiers, the programmer must use the keywords: *private*, *public* or *protected* when needed, followed by a semicolon and then define the data and member functions under it.

```
class NewClass
{
    private:
        int
        data;
        float
        cost;
    public:
        void getData(int a, float b); //function to initialize data
        void putData(void);          //function to return data
};
```

In the code above, the member *x* and *y* are defined as private access specifiers. The member function *sum* is defined as a public access specifier.

General Template of a class:

General structure for defining a class is:

```
class classname
{
    Access specifier:
        Data members;
        Member Functions

    Access specifier:
        Data members;
        Member Functions
};
```

Comparison between a Class and a Structure

<pre>class Bird { char name[10]; int age; say(); move(); };</pre>	<pre>struct Bird { char name[10]; int age; say(); move(); };</pre>
--	---

If you compare the two declarations, you will not notice any difference except the keyword “**struct**” or “**class**”. The difference between them is in their implementation. A Structure’s components are defined **public** by default while that of a class are defined **private** by default. Here **private** and **public** are called **access specifiers**.

Creating Objects

Once class has been declared, we can create variables of that type by using the class name (like other built-in variables)

Just as we declare a variable of data type *int* as: `int`

```
x;
```

Objects are also declared as:

class name followed by object name;

```
NewClass N;          //memory for N is created
```

This declares *e1* to be an object of class `NewClass`

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol

} of the class declaration. **For example**

```
class NewClass
{
    -----
    -----
}N1 , N2 , N3 ;
```

would create the objects N1 , N2 and N3 of type NewClass

Accessing Class members

The private data of class can be accessed only through the member function of that class. The following is the format for calling a member function.

```
Object-name.function-name (actual-arguments);
```

For example, the function call statement

```
N.getdata (100, 75.5) ;
```

is valid and assigns the value 100 to **data** and 75.5 to **cost** of the object N by implementing **getdata()** function. Similarly the statement

```
N.putdata () ;
```

Would display the values of the data members.

Defining Member Function

Member function can be defined in two places

- Outside the class definition
- Inside the class definition

Outside the class definition:

An important difference between a member function and normal function is that a member function incorporates a membership „identity label,, in the header. This label

tells the compiler which class the function belongs to. The general form of a member function definition is:

```
Return-type class-name :: function-name (argument declaration)  
{  
    Function body  
}
```

`class-name ::` tells the compiler that the function *function-name* belongs to the *class-name*

`::` Is called scope resolution operator.

For example

```
void NewClass :: getdata(int a, float b)
```

```
{
```

```
    data = a; cost
```

```
    = b;
```

```
}
```

```
void NewClass :: purdata(void)
```

```
{
```

```
    cout<<"Data :"<<data<<"\n";
```

```
    cout<<"Cost :"<<cost<<"\n";
```

```
}
```

Inside the Class Definition

Another method of defining the member function is to replace the function declaration

by the actual function definition inside the class. For instance

```
class NewClass
{
    int data; float
    cost;

public:

    void getdata(int a, float b);
        //inline function

    void putdata(void ) {
        //display data }

};
```

when function is defined inside a class, it is treated as an inline function.

friend function

Friend function is a special function which can access the private and protected Members of a class through the object of the same class. Friend functions are not the member functions of a class and they can be declared under any access specify. To make an outside function “friendly” to the class we have to declare this function as friend of the class as shown

```
class ABC
{
    .....
    public:
    .....
    .....
    friend void xyz(void); // Declaration
};
```

Algorithm:

- 1) Create any class(NewClass) , define class variables, member functions of class.
- 2) In main function create object of the class(N,N1)
- 3) Access members of above class in main class.
- 4) Call the methods of the class with the respective class with **obj_name.method_name(N.getdata(actual arguments))**

Output



```
Command Prompt - tc
Object N
Data : 100
Cost : 566.5
Object N1
Data : 200
Cost : 458.5
```

Practice No1.

Design, develop, and execute a program in C++ based on the following requirements:

An EMPLOYEE class is to contain the following data members and member functions:
Data members: EmployeeNumber (an integer), EmployeeName (a string of characters), BasicSalary (an integer), All Allowances (an integer), IT (an integer), NetSalary (an integer).

Member functions: to read the data of an employee, to calculate Net Salary and to print the values of all the data members. (AllAllowances = 123% of Basic; Income Tax (IT) = 30% of the gross salary (= basic Salary _ AllAllowance); Net Salary = Basic Salary + All Allowances – IT)

Algorithm:

1. Create Class Employee.
2. Class Employee Contains following data members
 - a. Employee_Number as integer
 - b. Employee_Name as String
 - c. Basic_Salary as integer
 - d. All_Allowances as integer
 - e. IT as integer
 - f. Net_Salary as integer
 - g. Gross_Salary as integer
3. Class Employee Contains following members functions
 - a. Create function as getdata for accepting information of employee. Like employee name,employee number and basic salary etc.
 - b. Create function Net_salary_Calculation to calculate gross salary.
 - c. Create function displayInformation to display information about employee.
4. Create main function to call this function of class Employee.

Program:

```
#include<conio.h>
#include<iostream.h>
Class Employee
{
    Private:                //Access Specifier
        Int Employee_Number;
        Char Employee_Name[50];
        Int Basic_Salary,Net_Salary,IT,All_Allowances,Gross_Salary;

    Public:
        Void getdata()
        {
            Cout<<"Enter Employee Name:";
            Cin>>Employee_Name;
            Cout<<"Enter Employee Number:";
            Cin>>Employee_Number;
```

```

        Cout<<"Enter Employee Basic Salary:";
        Cin>>Employee_Salary;
    }

    Void Net_salary_Calculation()
    {
        All_Allowances=123/100*Basic_Salary;
        Gross_Salary=Basic_Salary+All_Allowances;
        IT=30/100*Gross_Salary;
        Net_Salary=Basic_Salary+All_Allowances-IT;
    }

    Void displayInformation()
    {
        Cout<<"\n-----Information About Employee-----";
        Cout<<"\nEmployee Name:"<<Employee_Name;
        Cout<<"\nEmployee Number:"<<Employee_Number;
        Cout<<"\nEmployee Basic Salary:"<<Basic_Salary;
        Cout<<"\nEmployee Net Salary:"<<Net_Salary;
    }
};

Void main()
{
    Employee e;           //creating object of employee
    Clrscr();             //clear the screen

    e.getdata();          // calling function
    e.Net_salary_Calculation()
    e.displayInformation();

    getch();
}

```

Output:

```

Enter Employee Name: ABC
Enter Employee Number:123
Enter Employee Basic Salary:10000

-----Information About Employee-----";
Employee Name:ABC
Employee Number:123
Employee Basic Salary:10000
Employee Net Salary:15610

```


Conclusion

A class is an extension to the structure data type. Data member should be grouped in **private** access specifier and member functions are normally clustered together in **public** section. Attempting to access **private** members from outside the class will cause syntax error.

Experiment No.3

Title: Write a program to demonstrate different types of constructors

Objective:

At the end of this experiment, students should be able to understand following points:

1. Concept of Constructor and Destructor.
2. Use of constructor and Destructor.

Theory:

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

```
<class name> { arguments};
```

The default constructor for a class X has the form

X::X()

In the above example the arguments is optional. The constructor is automatically named when an object is created. A constructor is named whenever an object is defined or dynamically allocated using the "new" operator.

The constructor is invoke whenever an object of its class is created. It is called constructor because it constructs the values of data member of the class

A constructor is declared and defined as follows

```
class integer
{
    int m, n;

    public:

    integer(void);    //constructor declared

    -----
    -----

};

integer :: integer(void)    // constructor defined

{

    m=0;

    n=0;

}
```

The Constructor functions have some special characteristics:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.

- They do not have return types, not even void and therefore they cannot return values.
- Cannot be inherited, through a derived class can call the base class constructor.
- Like other C++ function, they can have default arguments.
- Constructor cannot be **virtual**.
- We cannot refer to their address.
- An object with a constructor (or destructor) cannot be use as a member of union.
- They make implicit calls to the operators **new** and **delete** when memory allocation is required.

PARAMETERIZED CONSTRUCTOR

The constructors that can take arguments are called as *parameterized constructors*.

The constructor `integer()` may be modified to take arguments as shown below

```
class integer
{
    int m, n;
public:
    integer(int x , int y)
    -----
    -----
};

integer :: integer(int x, int y)    //parameterized constructor
{
    m = x; n =
```

```
    y;  
}
```

when constructor has been parameterized, the object declaration statement

```
integer int1;
```

may not work. We must pass initial values as arguments to the constructor function when an object is declared. This can be done in two ways.

- By calling the constructor explicitly

```
integer int1 = integer(0,150); //explicit call
```

this function creates an integer object `int1` and passes the value 0 and 150 to it.

- By calling the constructor implicitly

```
integer int1 = integer(0,150); //implicit call
```

CONSTRUCTOR WITH DEFAULT ARGUMENTS

It's possible to define a constructor with default arguments. For e.g. `complex()` can be declared as

```
complex(float real, float imag=0);
```

the default value of the argument **imag** is **zero**. Then, the statement

```
complex C(5.6);
```

assigns the value 5.6 to the **real** variable and 0.0 to **imag** (by default).

COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object. For e.g. the statement

```
integer I2(I1);
```

Would define the object `I2` and at the same time initialize it to `I1`. Another form of this statement is `integer I2 = I1;`

The process of initializing through a copy constructor is known as *copyinitialization*. A copy constructor takes a reference to an object of the same class as itself as an argument.

MULTIPLE CONSTRUCTORS IN A CLASS

```
class integer
{
    int m, n;

public:
    integer( ) { m = 0 ; n = 0; }           //constructor 1
    integer( int a, int b)                 //constructor 2
    { m = a; n = b; }
    integer(integer & i)                   //constructor 3
    {m = i.m; n = i.n;}

};
```

This declares three constructors for an integer object .

- `integer I1;`

the declaration would invoke the first constructor and set both m and n of I1 to zero.

(Receives no argument)

- `integer I2(10,20);`

initializes the data members m and n of I2 to 10 and 20 respectively.

- `integer I3(I2);`

invokes the third constructor which copies the value of I2 to I3. such a constructor is called as *copy constructor*

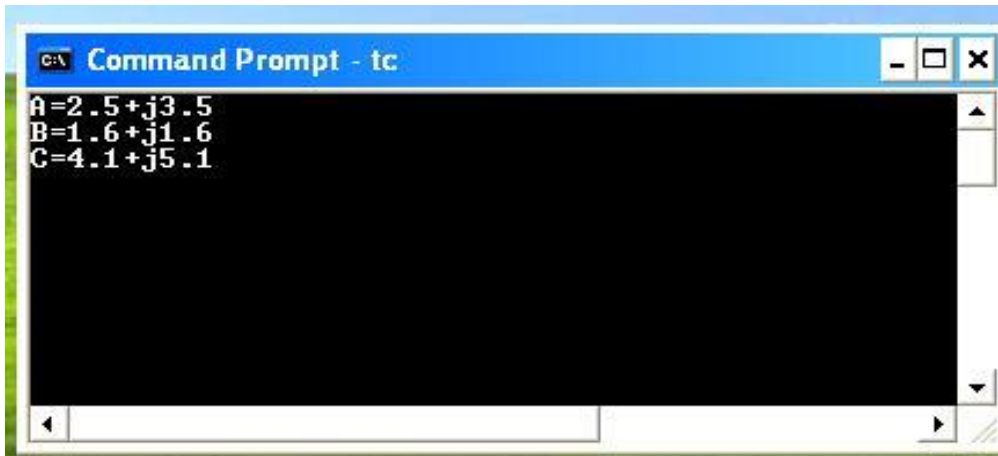
Algorithm

1) Declare class(complex). Declare data members(x, y) and methods (constructor with

no argument ,with one and two argument) also declare friend function if required.(
`friend complex sum(complex, complex);`)

- 2) Define the declared methods with help of scope resolution operator if it is defined outside the class.
- 3) Create an object of the respective class (complex c;) and call the constructor or pass the value to the constructor.

Output



```
C:\> Command Prompt - tc
A=2.5+j3.5
B=1.6+j1.6
C=4.1+j5.1
```

DESTRUCTORS

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 integer can be defined as shown below

```
~integer( ) { }
```

A destructor never takes any argument nor does it returns any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is good practice to declare the destructors in a program since it releases memory space for future use.

What is the use of Destructors

Destructors are also special member functions used in C++ programming language.

Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

General Syntax of Destructors

~ classname();

The above is the general syntax of a destructor. In the above, the symbol tilde ~ represents a destructor which precedes the name of the class.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

For example

```
class alpha
{
    public:
        alpha()
        {
            count++;
        }
};
```

```

cout<<"\nNumber of object created"<<count;
}
~alpha()
{
cout<<"\nNumber of object destroyed"<<count;
}
};

```

Algorithm

- 1) Declare constructor in any class(alpha)
- 2) In constructor declare one variable for keeping the record of created objects
- 3) For releasing the memory of declared constructor define the destructor with „~“ sign, above variable will show the destroyed object.
- 4) Create object of class (alpha)

Output

```

C:\> Command Prompt - tc
Enter Main
Number of object created 1
Number of object created 2
Number of object created 3
Number of object created 4
Enter Block 1
Number of object created 5
Number of object destroyed 5
Enter Block 2
Number of object created 5
Number of object destroyed 5
Re-enter Main
Number of object destroyed 4
Number of object destroyed 3
Number of object destroyed 2
Number of object destroyed 1

```

Conclusion

With help of constructors we have fulfilled one of our requirements of implementation of abstract data types. Initialization at definition time. Providing a constructor to ensure every object is initialized with meaningful values can help eliminate logic errors. We still need a mechanism which automatically destroy same object when it gets invalid. (For e.g. because of leaving its scope.) Therefore classes can define destructors

Experiment No.4

Title: Write a program using static data member as well as member function, create function using default argument concept.

Objective:

At the end of this experiment, students should be able to understand following points:

1. Static Data Member and Static Member Function.
2. Concept of Defalut argument.

Theory:

Static Data Member:

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Declaration Syntax:

```
Static static_variable_name;
```

Initialization of static variable:

Static variable initialize outside the class.

Syntax:

```
Data_type class_name:: static_variable_name=value;
```

Access Static data member outside the class Syntax:

```
class_name:: static_variable_name;
```

Algorithm:

1. Create Class name Box.
2. Declare class member objectCount as static, length, breadth, height.
3. Define member function
 - a. Define constructor.
 - b. Member function Volume for calculate Volume.
4. Create main function to call functions of Box class.

Let us try the following example to understand the concept of static data members:

```
#include<iostream.h>

classBox
{
public:
staticint objectCount; //static variable
// Constructor definition with default argument
```

```

Box(double l=2.0,double b=2.0,double h=2.0)
{
    cout <<"Constructor called."<< endl;
    length = l;
    breadth = b;
    height = h;
// Increase every time object is created
    objectCount++;
}
double Volume()
{
return length * breadth * height;
}
private:
double length;// Length of a box
double breadth;// Breadth of a box
double height;// Height of a box
};

// Initialize static member of class Box
int Box::objectCount =0;

int main(void)
{
BoxBox1(3.3,1.2,1.5);// Declare box1
BoxBox2(8.5,6.0,2.0);// Declare box2

// Print total number of objects.
    cout <<"Total objects: "<<Box::objectCount << endl;

return 0;
}

```

When the above code is compiled and executed, it produces the following result:

Output:

```

Constructor called.
Constructor called.
Total objects:2

```

Static Member Function:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Declaration of Static Function

```
Static data_type function_name(Argument1,argument2,...,argument)
{
    //Access only static data member and static member functions.
    Static_variable=value;
}
```

Calling Static function outside the class

```
Class_name::static_function_name();
```

Algorithm:

1. Create Class name Box.
2. Declare class member objectCount as static,length,breadth,height.
3. Define member function
 - a. Define constructor.
 - b. Member function Volume for calculate Volume.
 - c. getCount Member function as static
4. Create main function to call functions of Box class.

Let us try the following example to understand the concept of static function members:

```
#include<iostream>

usingnamespace std;

classBox
{
public:
staticint objectCount;
// Constructor definition with default argument
Box(double l=2.0,double b=2.0,double h=2.0)
{
    cout <<"Constructor called."<< endl;
    length = l;
    breadth = b;
    height = h;
// Increase every time object is created
    objectCount++;
}
doubleVolume()
{
return length * breadth * height;
}
staticint getCount()
{
return objectCount;
}
private:
double length;// Length of a box
double breadth;// Breadth of a box
double height;// Height of a box
};

// Initialize static member of class Box
intBox::objectCount =0;

int main(void)
{
```


Stack[9]

Basic terminology associated with stacks:

- 1) **Stack Pointer (TOP):** Keeps track of the current position the stack.
- 2) **Overflow:** Occurs when we try to insert (push) more information on a stack than it can hold.
- 3) **Underflow:** Occurs when we try to delete (pop) an item off a stack, which is empty.

Basic Operation Associated with Stacks:

- 1) Insert (Push) an item into the stack.
- 2) Delete (Pop) an item from the stack.

Program:

```
#include<iostream>
using namespace std;
#define MAX 10
int top=-1,ch,i;
template <class T>
class StackADT
{
public:
virtual void Push()=0;
virtual void Pop()=0;
virtual void Top()=0;
virtual void Display()=0;
};
template <class T>
class Stack: public StackADT<T>
{
T stk[MAX],ele;
public:
void Push();
void Pop();
void Top();
void Display();
};
template <class T>
void Stack<T>::Push()
{
if(top==(MAX-1))
```

```

cout<<"\nThe stack is full";
else
{
cout<<"\nEnter an element:";
cin>>ele;
top++;
stk[top]=ele;
cout<<"\nElement pushed successfully\n";
}
}
template <class T> void Stack<T>::Pop(){
if(top== -1)
cout<<"\nThe stack is empty";
else
{
ele=stk[top];
top--;
cout<<"The deleted element is:"<<ele;
}
}
template <class T>
void Stack<T>::Top()
{
if(top== -1)
cout<<"\nThe stack is empty";
else
cout<<"The top element of the stack is:"<<stk[top];
}
template<class T>
void Stack<T>::Display()
{
if(top== -1)
cout<<"\nThe stack is empty";
else
{
cout<<"\nThe elements in the stack are:";
for(i=top;i>=0;i--)
cout<<"\n"<<stk[i];
}
}
int main()

```



```

{
Stack<int> s1;
do
{
cout<<"\n****MENU****";
cout<<"\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit";
cout<<"\nEnter ur Choice:";
cin>>ch;
switch(ch)

{
case 1: s1.Push();
        break;
case 2: s1.Pop();
        break;
case 3: s1.Top();
        break;
case 4: s1.Display();
        break;
case 5: exit(1);
default: cout<<"Enter correct Choice";
}
}while(true);
}

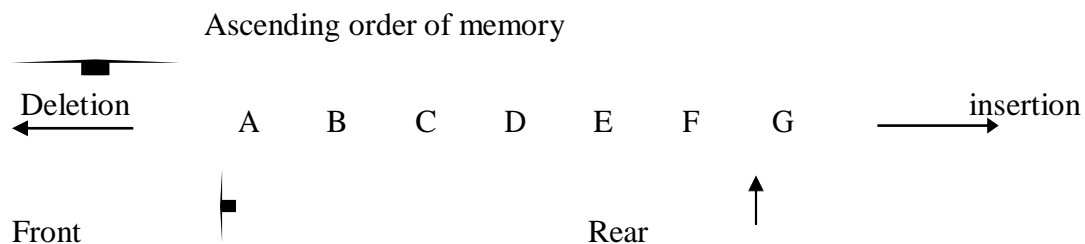
```

Experiment No.6

Aim: A C++ program to implement the Queue ADT using arrays.

QUEUE:

Queue is an ordered collection of data such that the data is inserted at one end and deleted from other end. It is a collection of items to be processed on a First-In-First-Out(FIFO) or First Come First Served(FCFS) basics.



Basic Operation Associated on Queues:

- 1) Insert an item into the Queue.
- 2) Delete an item into the Queue.

Program:

```
#include<iostream>
using namespace std;
#define MAX 10
int front=0,rear=0,ch,i;
template <class T>
class QueueADT
{
public:
virtual void Insert()=0;
virtual void Delete()=0;
virtual void Display()=0;
};
template <class T>
class Queue: public QueueADT<T>
{
T q[MAX],ele;
public:
void Insert()
{
if(rear==MAX)
cout<<"\nQueue is full";
else
{
cout<<"\nEnter an element:";
cin>>ele;
q[rear]=ele;
rear++;
cout<<"\nElement inserted successfully\n";
}
}
void Delete()
{
if(front==rear)
cout<<"\nQueue is empty";
else
{
```

```

ele=q[front];
front++;
cout<<"The deleted element is:"<<ele;
}
}
void Display()
{
if(front==rear)
cout<<"\nQueue is empty";
else
{
cout<<"\nThe elements in the queue are:";
for(i=front;i<rear;i++)
cout<<q[i]<<" ";
}
};
int main()
{
Queue<int> q1;
do
{ cout<<"\n***MENU***";
cout<<"\n1. Insert\n2. Delete\n3. Display\n4. Exit";
cout<<"\nEnter ur Choice:";
cin>>ch;
switch(ch)
{
case 1: q1.Insert();
break;
case 2: q1.Delete();
break;
case 3: q1.Display();
break;
case 4: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);

```

2) } Write C++ programs to implement the following data structures using a singly linked list.

a) Stack ADT

b) Queue ADT

Aim: A C++ program to implement the Stack ADT using singly linked list.

Program:

```
#include<iostream>
using namespace std;
template<class t>
class node
{
public:
t info;
node *link;
};
template <class t>
class StackADT
{
virtual void Push()=0;
virtual void Pop()=0;
virtual void Top()=0;
virtual void Display()=0;
};
template<class t>
class StackLinkImp:public StackADT<t>
{
public:
t item;
node<t> *temp, *top;
StackLinkImp()
{
top=NULL;
}
void Push()
{
temp=new node<t>;
cout<<"Enter the item to be inserted on to the stack:";
cin>>item;
if(top==NULL)
temp->link=NULL;
else
temp->link=top;
temp->info=item;
top=temp;
cout<<"Insertion Completed Successfully";
```

```

}
void Pop()
{
if(top==NULL)
cout<<"Stack is empty";
else
{
item=top->info;
top=top->link;
cout<<"The deleted element is:"<<item;
}
}
void Top()
{
if(top==NULL)
cout<<"Stack is empty";
else
cout<<"The top element is:"<<top->info;
}
void Display()
{
if(top==NULL)
cout<<"Stack is empty";
else
{
temp=top;
cout<<"The elements in the stack are:";
while(temp!=NULL)
{
cout<<temp->info<<" ";
temp=temp->link;
}
}
};
int main()

{
int ch;
StackLinkImp<int> s1;
do

```

```

{
cout<<"\n****Menu****";
cout<<"\n1. Push\n2. Pop\n3. Top\n4. Display\n5. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{
case 1: s1.Push();
        break;
case 2: s1.Pop();
        break;
case 3: s1.Top();
        break;
case 4: s1.Display();
        break;
case 5: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);
}

```

3) Write C++ programs to implement the Double Ended Queue (DEQUE) using array.

Aim: A C++ program to implement the Double Ended Queue (DEQUE) using array.

Program:

```

#include <iostream>
using namespace std;
#define MAX 10
int front=-1,rear=-1,c,i;
template<class t>
class dqeADT
{
public:
virtual void addqatbeg()=0;
virtual void addqatend()=0;
virtual void delqatbeg()=0;
virtual void delqatend()=0;
virtual void display()=0;
};
template <class t>
class dqe: public dqeADT<t>
{

```

```

public:
t q[MAX],item;
deque()
{
front=rear=-1;
for(i=0;i<MAX;i++)
q[i]=0;
}
void addqatbeg()
{
cout<<"Enter an element:";
cin>>item;
if (front==0&&rear==MAX-1)
{
cout<<"\nDeque is full"<<endl;
return ;
}
if(front==-1)
{

```

Aim: A C++ program to implement the Queue ADT using singly linked list.

Program:

```

#include<iostream>
using namespace std;
template<class t>
class node
{
public:
t info;
node *link;
};
template<class t>
class QueueADT
{
virtual void Insert()=0;
virtual void Delete()=0;
virtual void Display()=0;
};
template <class t>
class QueueLinkImp: public QueueADT<t>
{

```

```

public:
t item;
node<t> *temp,*front,*rear;
QueueLinkImp()
{
front=rear=NULL;
}
void Insert()
{
temp=new node<t>;
cout<<"Enter an element to be inserted in to queue:";
cin>>item;
temp->info=item;
temp->link=NULL;
if(front==NULL)
{
front=rear=temp;
cout<<item<<" inserted Successfully";
return;

}
rear->link=temp;
rear=rear->link;
cout<<item<<" inserted Successfully";
}
void Delete()
{
if(front==NULL)
cout<<"Queue is empty";
else
{
item=front->info;
front=front->link;
cout<<"Deleted Element is: "<<item;
}
}
void Display()
{
if(front==NULL)
cout<<"Queue is empty";
else

```



```

{
cout<<"The elements in the queue are:";
for(temp=front;temp!=NULL;temp=temp->link)
cout<<temp->info<<" ";
}
};
void main()
{
int ch;
QueueLinkImp<int> q1;
do
{
cout<<"\n*****MENU*****";
cout<<"\n1. Insert\n2. Delete\n3. Display\n4. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{

case 1: q1.Insert();
break;
case 2: q1.Delete();
break;
case 3: q1.Display();
break;
case 4: exit(1);
default: cout<<"Entered Wrong Choice";
}
}while(1);
}

```

Aim: A C++ program to implement the Double Ended Queue (DEQUE) using array.

Program:

```

#include <iostream>
using namespace std;
#define MAX 10
int front=-1,rear=-1,c,i;
template<class t>
class dqeADT
{
public:

```

```

virtual void addqatbeg()=0;
virtual void addqatend()=0;
virtual void delqatbeg()=0;
virtual void delqatend()=0;
virtual void display()=0;
};
template <class t>
class dqe: public dqeADT<t>
{
public:
t q[MAX],item;
dqe()
{
front=rear=-1;
for(i=0;i<MAX;i++)
q[i]=0;
}
void addqatbeg()
{
cout<<"Enter an element:";
cin>>item;
if (front==0&&rear==MAX-1)
{
cout<<"\nDeque is full"<<endl;
return ;
}
if(front==-1)
{
front=rear=0;
q[front]=item;
return;
}
if(rear!=MAX-1)
{
c=count();
int k=rear+1;
for(i=1;i<=c;i++)
{
q[k]=q[k-1];
k--;
}
}
}

```

```

q[k]=item;
front=k;
rear++;
}
else
{
front--;
q[front]=item;
}
}
void addqatend()
{
cout<<"Enter an element:";
cin>>item;
if(front==0&&rear==MAX-1)
{
cout<<"\nDeque is full"<<endl;
return;
}
if(front==-1)
{
rear=front=0;
q[rear]=item;
return;
}
if(rear==MAX-1)
{

int k=front-1;
for(i=front-1;i<rear;i++)
{
k=i;
if(k==MAX-1)
q[k]=0;
else
q[k]=q[i+1];
}
rear--;
front--;
}
rear++;

```

```

q[rear]=item;
}
void delqatbeg()
{
if(front== -1)
{
cout<<"\nDeque is empty"<<endl;
return;
}
item=q[front];
q[front]=0;
if(front==rear)
front=rear--1;
else
front++;
cout<<"Deleted item is:"<<item;
}
void delqatend()
{
if(front== -1)
{
cout<<"\nDeque is empty"<<endl;
return;
}
item=q[rear];
q[rear]=0;
rear--;

if(rear== -1)
front=-1;
cout<<"Deleted item is: "<<item;
}
void display()
{
cout<<endl<<"front-> ";
for(i=0;i<MAX;i++)
cout<<" "<<q[i];
cout<<" <-rear";
}
int count()
{

```

```

int c=0;
for(i=0;i<MAX;i++)
{
if(q[i]!=0)
c++;
}
return c;
}
};
main()
{
int ch;
deque<int> s1;
do
{
cout<<"\n****Menu****";
cout<<"\n1. Insert at Beginning\n2. Insert at End\n3. Delete from Beginning\n4. Delete from
End\n5. Display\n6. Exit";
cout<<"\nEnter ur choice:";
cin>>ch;
switch(ch)
{
case 1: s1.addqatbeg();
break;
case 2: s1.addqatend();
break;
case 3: s1.delqatbeg();

break;
case 4: s1.delqatend();
break;
case 5: s1.display();
break;
case 6: exit(1);
}
}while(1);
}

```


4) Write C++ programs to implement the Double Ended Queue (DEQUE) using array.

