4BCA2 Programming in C++

Syllabus :-

Unit 1: Introduction to object oriented concept, Inheritance. Class object, Polymorphism, Overloading, Dynamic binding. Advantages of OOP Object Oriented Analysis and Design.

Unit 2: Fundamental of C++ Programming, History of C++, Structure of C++, Declaration of class and object Definition and declaration of members, objects as data type, objects as function arguments, structure and classes.

Data type, Basic Data type. User defined data types, Variables and Constants, Dynamic initialization of variables reference variable, enumerated variable. Operators and expression. Coding central construct. sequence selection iteration, Arrays and Strings

Input/output mechanism.

Concept of function declaration, calling, definition.

Scope rules of members of class.

Data and Type Conversion, between basic type, object and object of different types. Limitation of type conversion.

Unit 3:- Constructor and Destructors — Basic Constructors, Parameter Constructors, Constructors with default arguments. Dynamic Initialization of objects. Copy constructors Destructor Limitation on Constructors and Destructors.

Unit 4:- Overloading — Function overloading operator overloading — unary, binary.

Unit 5:- Inheritance — Derived classes and base classes. Access specifiers, Derived class constructors — Member function ~~overloading~~ overriding, types of inheritance Public and Private, Multiple Inheritance

Unit 6:- Pointers and Virtual function — pointer concept. Pointer to objects, Memory management using new and delete operators.
Virtual function — pure and polymorphism, Friend function and Static function. Macros and inline function.

Unit 7:- Streams — stream classes — hierarchy, header files, stream manipulators, string stream, character stream.
Object Input/Output and file stream Disk I/O with member function.

Unit 8 :- Templates — Function templates and class templates.

Unit 9 :- Exception Handling —
Try block, catch block handler,
throw statement.
Exception specification.

Unit 1 :- Introduction to OOP Concept

A sequence of instructions that a computer can interpret and execute to complete task is called computer program. People start learning programming by writing small and simple programs consisting only sequence of commands. This technique is called unstructured programming technique. When program becomes large, the process provides tremendous disadvantages

To overcome these problems functions were adopted for better understanding. Functions are called subroutines, subprograms or procedures in different languages this is called structured programming. or POP (procedure Oriented Programs). POP has two major drawbacks — data security and it does not model very well the real world problems (Cobol, Fortran, C).

OOP was invented to overcome the drawbacks of the POP.

The fundamental idea behind Object Oriented Languages is to combine into a single unit both data and the functions that operate on the data, such a unit is called an 'Object'. An object is a self contained unit or module of data and code. All data and procedures related to the object are defined within it. So, methods in OOP terminology and member functions in C++ are tied to the data.

The advantages off OOP fall into two broad categories, increased programming productivity and decrease maintenance cost.

Classes and objects may be re-used in other applications. Re-used code would have already been tested, so it should need little testing when used in a new program.

Classes → In OOP classes contains the necessary information to create instances. There are three commonly used views on the definition of class —

- A class is a pattern, template, or blueprint for a category for structurally identical items. The items created using the class are called instances.
- A class is a thing that consists of both a pattern and a mechanism for creating items based on that pattern, i.e., class as an instance factory.
- A class is the set of all items created using a specific pattern. In other words, the class is the set of all instances of that pattern.

metaclass → It is a class whose instances themselves are classes. The instance creation mechanism of this class can, in turn, be used to create instances - although these instances may or may not themselves be classes.

parameterized class → It is a template for a class wherein specific items have been identified as being required to create non-parameterized classes based on the template. One cannot directly use the instance creation of a parameterized class.

Inheritance — It is the process by which objects of one class acquire the properties of objects of another class. Inheritance provides the idea of reusability. We can add additional features to an existing class without modifying it.

Polymorphism — Polymorphism, a greek term, means the ability to take more than one form. By using polymorphism, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way. Example :— we may have a shape object that draw a circle on the screen and can create another shape object that draws a rectangle instead.

There are two ways to apply polymorphism — by operator and function overloading and by implementing inheritance feature.

Encapsulation— This feature enables us to hide, inside the object, both the data fields and the methods that act on that data. After we do this, we can control access to the data, forcing programs to retrieve or modify data only through the object's interface. In strict object-oriented design, an object's data is always private to the object. Other parts of a program should never have direct access to that data.

Dynamic Binding →

C++ allows virtual function within a class, whose functionality can be over-ridden in its derived classes. The whole function body can be replaced with a new set of implementation in the derived class. This is different from overloading. It is a member function of a class declared with virtual keyword. It has a different functionality in the derived class. The call of this function is resolved at run-time. This is the different between non-virtual function and virtual function. non-virtual functions are resolved at compile time. This is called static binding.

Virtual functions are resolved ~~at~~ during run-time. This mechanism is known as dynamic binding.

Example:— Window can be a class having Create function to create a window with white background. But a class called CommandButton derived or inherited from window class, may have to use a gray background and write a caption on the center. The create function for CommandButton now should have a different functionality from the one at the class called Window.

Ex :-

```
class Window
{
    public :
    virtual void Create ()
    {
        cout << " Base class Window " << endl;
    }
};

class CommandButton : public Window
{
    public :
        void Create ()
        {
            cout << " Derived class Command
    Button Overriden C++ Virtual function " << endl;
        }
};

void main()
{
    Window *x, *y;
    x = new Window ();
    x -> Create ();
    y = new CommandButton ();
    y -> Create ();
    getch ();
}
```

Object Oriented Analysis and Design →

Object oriented analysis refers to the methods of specifying requirements of the software in terms of real-world objects, their behaviour, and their interactions.
Object oriented Design turns the software requirements into specifications for objects and derives class hierarchies from which the objects can be created.

All the phases in the object-oriented approach work more closely together because of the commonality of the object model. In one phase, the problem domain objects are identified, while in the next phase additional objects required for a particular solution are specified.

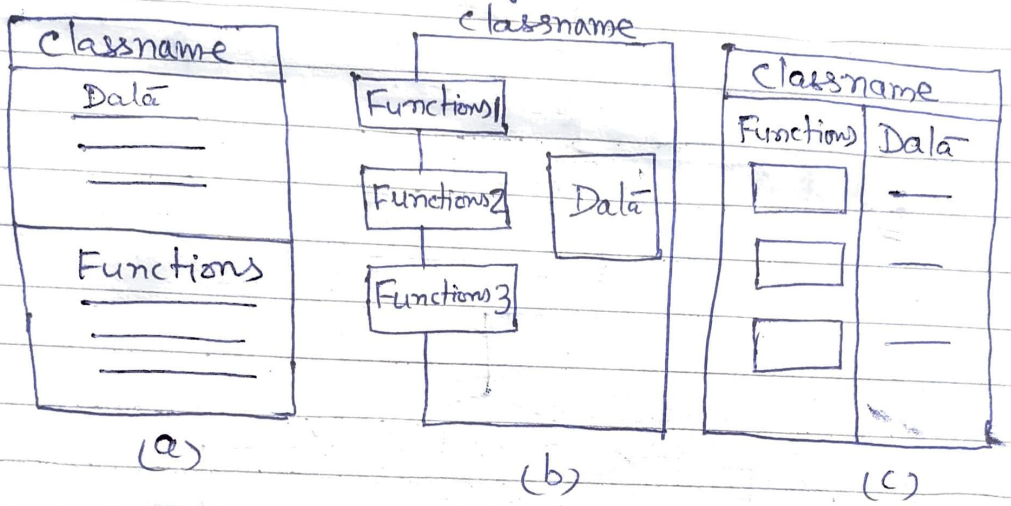
Graphical notations are an essential part of any design and development process, and object-oriented design is no exception. We need notations to represent classes, objects, subclasses, and their inter-relationships. There are no standard notations for representing the objects and their interactions.

Some of the commonly used notations to represent the following are given as —

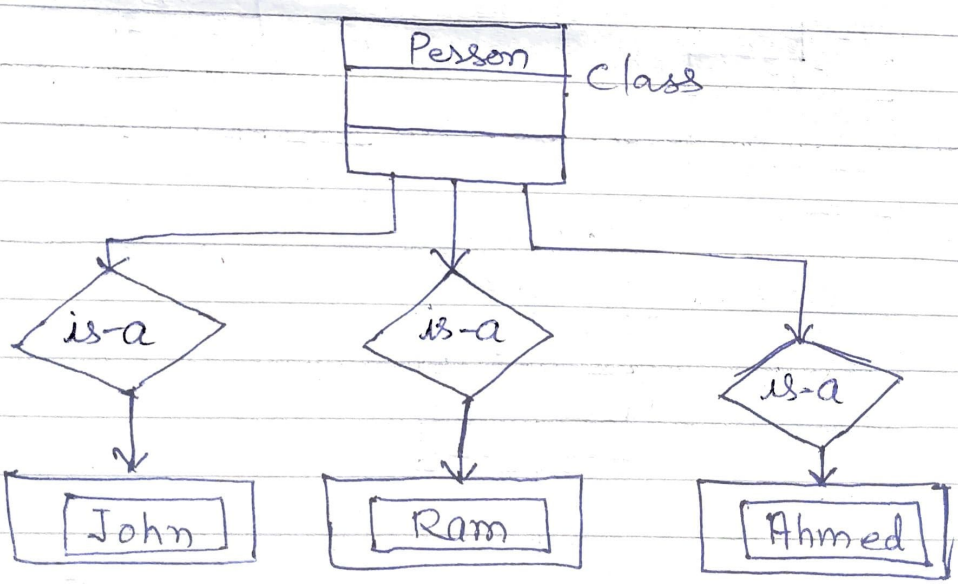
1. Classes and objects
2. Instances of objects
3. Message communication between objects.
4. Inheritance relationship

5. Classification relationship
6. Composition relationship
7. Hierarchical chart
8. Client-server relationship
9. Process layering.

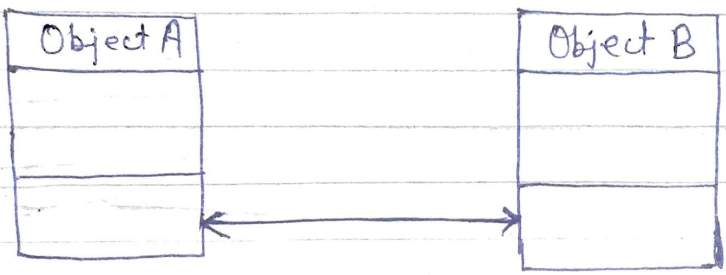
1.



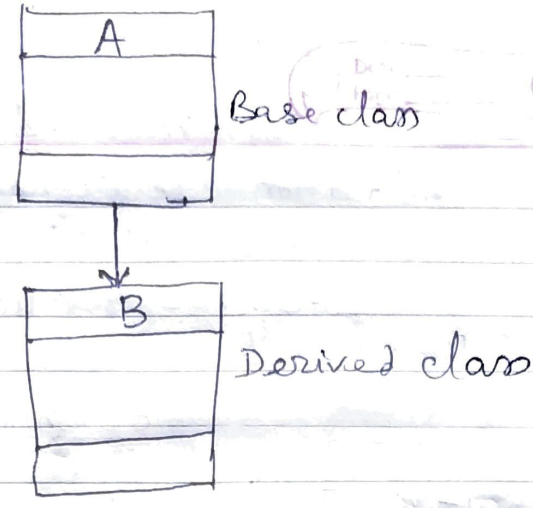
2.



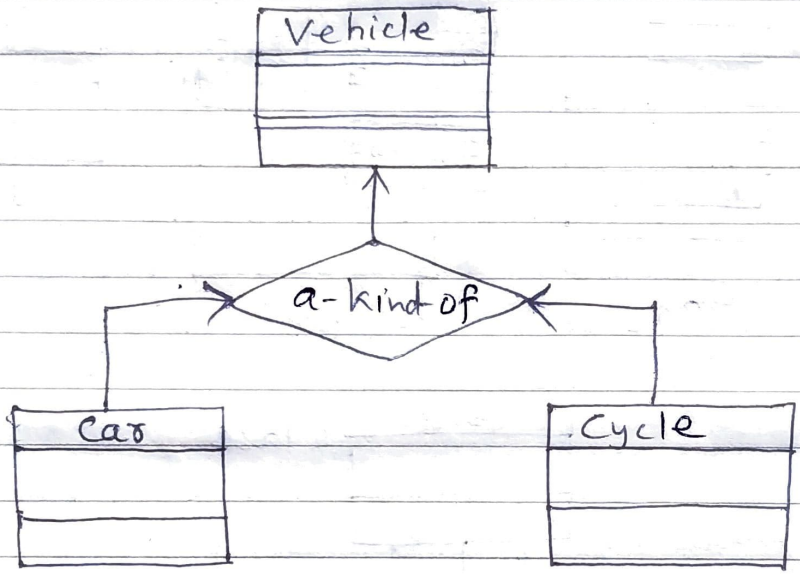
3.



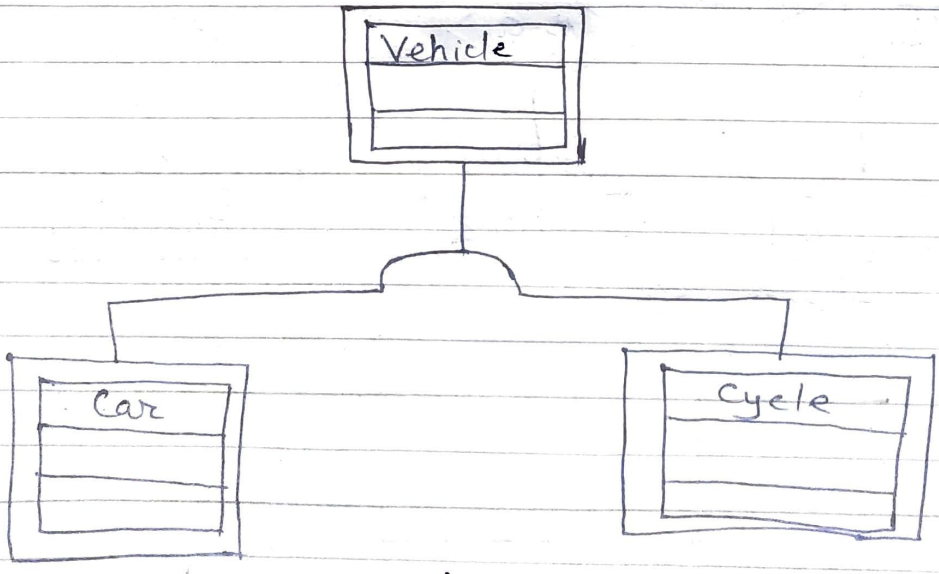
4.


A
Base class
B
Derived class

5.

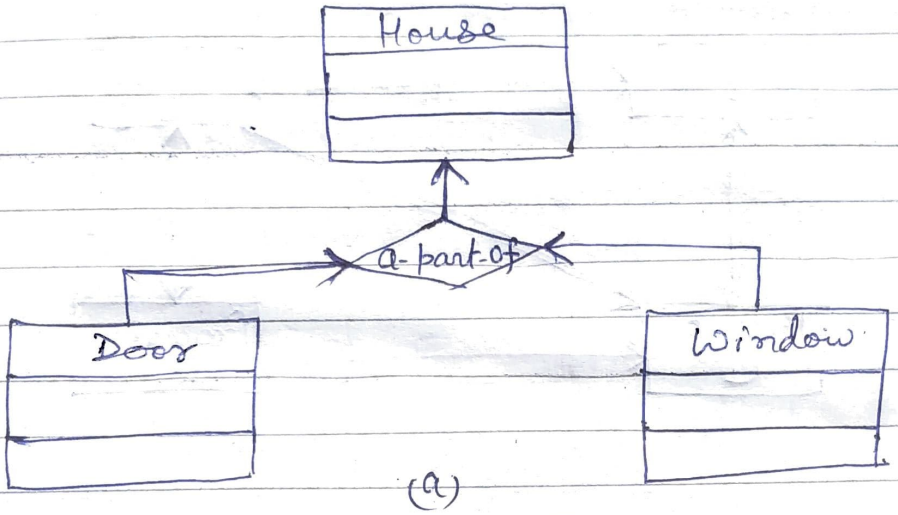

Vehicle
a-kind of
Car
Cycle

(a)

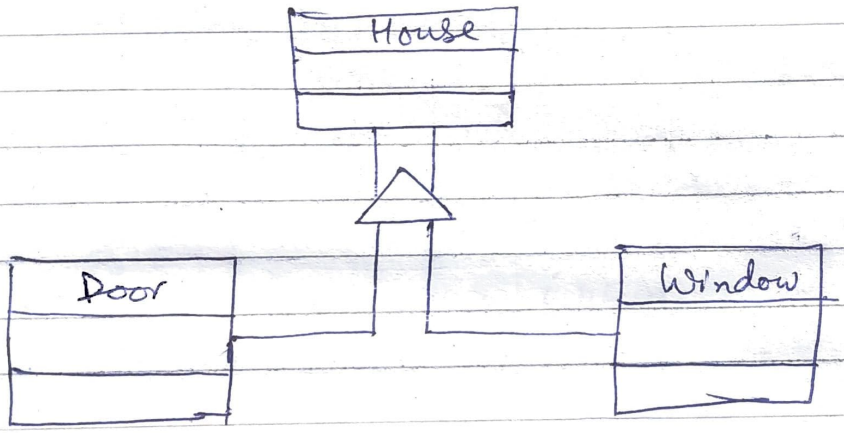

Vehicle
Car
Cycle

(b)

6. Composition relationship

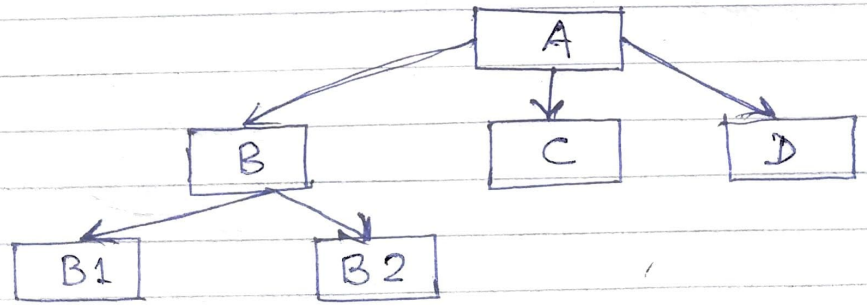


(a)

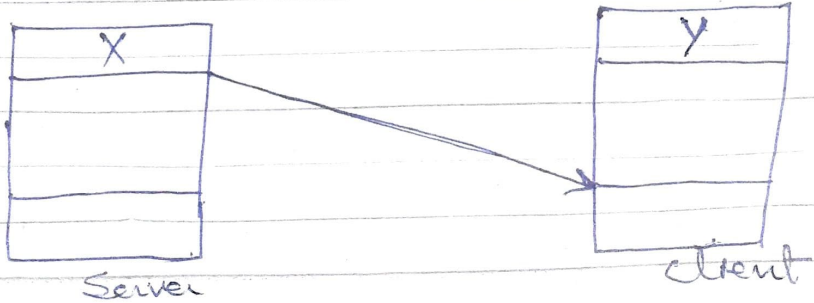


(b)

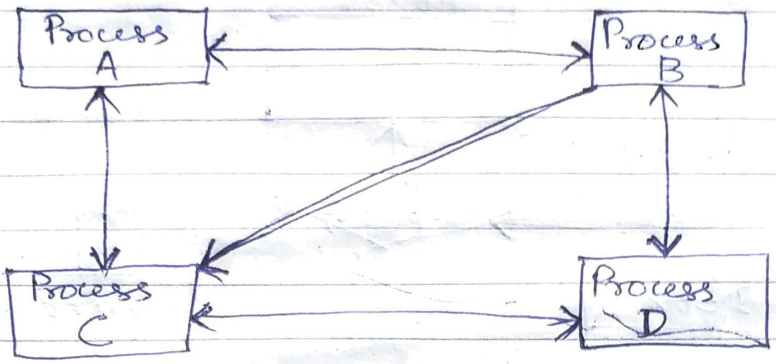
7. Hierarchical chart-



8. Client Server Relationship :-



9. Process layering :-



Steps in object-oriented analysis:—

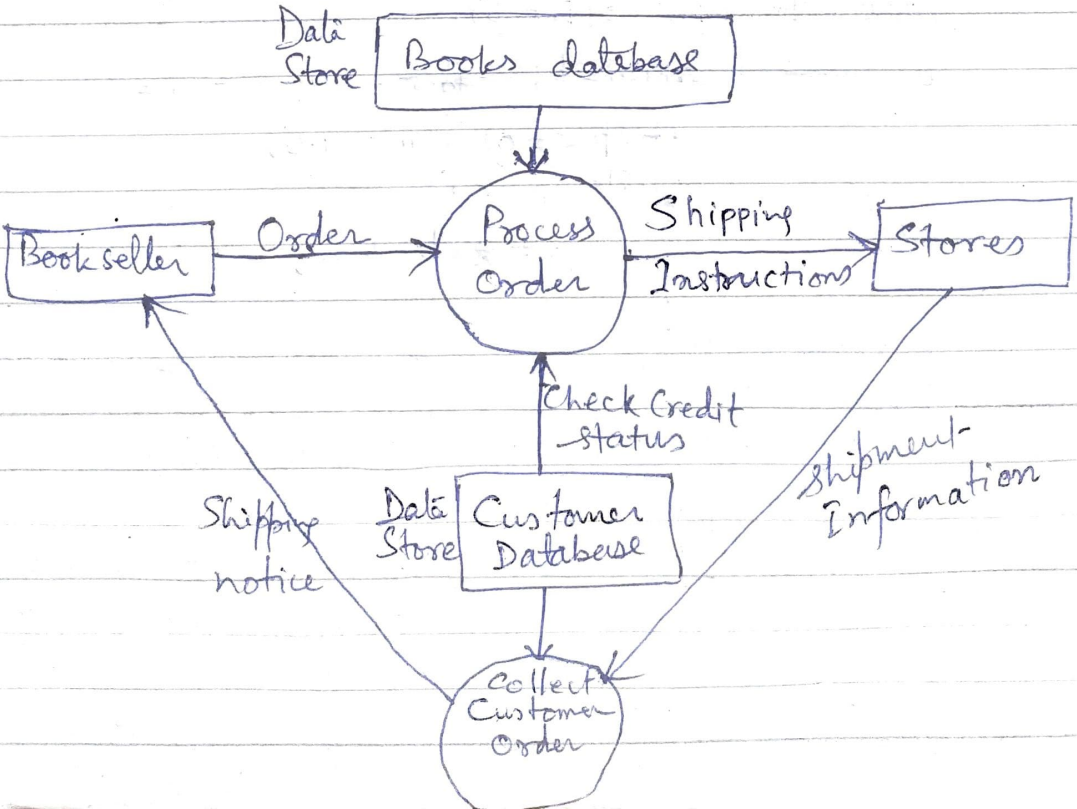
The object-oriented analysis approach consists of the following steps:—

1. Understanding the problem.
2. Drawing the specification of user need and the software
3. Identifying the objects and their attributes.
4. Identifying the services that each object is expected to provide.
5. Establishing collaborations between the objects in terms of services required and rendered.

Steps above may be analyzed by using one of the following two approaches:—

- ✓ Data Flow Diagram (DFD)
- ✓ Textual Analysis (TA)

A sample DFD for order processing and shipping for a publishing company is as:—



Steps in Object-oriented Design:-

1. Review of objects created in the analysis phase
2. Specification of class dependencies
3. Organization of class hierarchies
4. Design of classes
5. Design of member functions
6. Design of driver program — main()

function code known as the driver program.

Fibonacci Series (C++) Command Line arguments :-

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>

class fibonacci
{       int first;
        int second;
        int next;
    public: fibonacci(){ first=0; second=1; }
            void calc(int);
};
   void fibonacci :: calc(int n)
   {
        for(int c=0; c<n; c++)
        {
            if( c <= 1)
              next=c;
            else { next = first + second;
                   first = second;
                   second= next;
                 }
            cout << next << " ";
        }
   }
 void main(int argc, char *argv[])
  {     int no;
        if(argc==2)
          {   no=atoi(argv[1]);
              fibonacci ff;
              ff.calc(no);
          }
          else{ cout << "\n Pass a Number Please";
                exit(1);
              }
  }
```

Command Line argument factorial C++ (Constructor based)

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
class factorial
{       private:  int num;
                  float fact;
          public:  factorial();
                   factorial(int);
                  float calc(int);
                  void show(){ cout<<"Factorial of "<<
                  num<<" is "<<fact<<endl;
                             }
};
     factorial::factorial(){ }
     factorial::factorial(int n){ num=n; fact=1.0;
                                  int i=1;
                                  if(n<=0) fact=1.0;
                                  while(n>=i)
                                  { fact*=n;
                                    n--;
                                  }
                                }
float factorial::calc(int p)
{               num=p; fact=1.0; int i=1;
                if(p<=0) fact=1.0;
                while(p>=i)
                {   fact*=p; p--;
                }
   return fact;
}
     void main(int argc, char *argv[])
      {
            int number;
            if(argc==2){number=atoi(argv[1]);
                        factorial ff(number);
                        ff.show();
      cout<<"\n value through default Constructor\n";
      factorial ff1; ff1.calc(number); ff1.show();
```

```
} else {
    cout << "\n Par a no from Command line arg.";
    exit(1);
    }
}
```

Menu based factorial & Fibonacci Series Program

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
class factorial
 { private: int num;
            float fact;
   public:
            factorial();
            factorial(int);
            float calc(int);
            void show()
            {
   cout<<"FACTORIAL OF :"<<num<<" is :"<<fact;
            }
};

class fibonacci
 {    private: long first;
               long second;
               long next;
   public:
            fibonacci();
           void calc(int);
};

      fibonacci :: fibonacci()
       { first=0; second=1;}

   void fibonacci::calc(int n)
    {   cout<<"\n The Fibonacci Series upto "<<n<<
                                  "terms are:\n";

        for(int ctr=0; ctr<n; ctr++)
        {       if(ctr<=1)
                  next=ctr;
                else
```

→ Reverse PTO

```
{       next = first + second;
        first = second;
        second = next;
    }
    cout << next << " ";
  }
}

factorial :: factorial()
 {
 }
factorial :: factorial(int n)
 {      num=n;
        fact=1.0;
        int i=1;
        if(n==0) fact=1.0;
        while(n>=i)
        {
            fact *= n;
            n--;
        }
 }

float factorial :: calc(int p)
 {      num=p;
        fact=1.0;
        int i=1;
        if(p==0) fact=1.0;
        while(p>=i)
        {
            fact *= p;
            p--;
        }
  return fact;
 }
```

Reverse pTO

```
void main(int argc, char *argv[])
{
    int number;
    if(argc==2){ number = atoi(argv[1]);
    while(1)
    { cout<<"\n1. Factorial \n";
      cout<<"\n2. Fibonacci \n";
      char choice;
      cout<<"\nEnter your choice(1/2) : ?";
      cin>> choice;
  switch(choice)
   {
      case '1' : clrscr();
        factorial ff(number);
        ff.show();
        factorial ff1;
        ff1.calc(number);
        ff1.show();
        break;
    case '2' : clrscr();
        fibonacci ff2;
        ff2.calc(number);
        break;
    default:
        cout<<"\n Enter avalid choice";
        exit(1);
   }
}
                }else{
cout<<"Pass a No from Com line";
exit(1);
                }
getch();
}
```