F.Y. B.Sc.(IT) : Sem. II

Object Oriented Programming

Time : 2½ Hrs.] Prelim Question Paper Solution **[Marks : 75**

Q.1 Attempt the following (any THREE) [15]

Q.1(a) Differentiate between Object Oriented Programming and Procedure Oriented Programming. [5]

Ans.:

i) In POP Main program is divided into small parts depending on the functions.
In OOPS Main program is divided into small object depending on the problem.

ii) In POP There is no perfect way for data hiding.
In OOPS Data hiding possible in OOP which prevent illegal access of function from outside of it. This is one of the best advantages of OOP also.

iii) In POP Top down process is followed for program design.
In OOPS Bottom up process is followed for program design.

iv) In POP Importance is given to the sequence of things to be done.
In OOPS Importance is given to the data.

v) In POP Mostly functions share global data i.e data move freely around the system from function to function.
In OOPS mostly the data is private.

vi) In POP No access specifier.
In OOPS There are public, private, protected specifier.

vii) In POP Operator cannot be overloaded.
In OOPS Operator can be overloaded

viii) In POP C, Pascal, FORTRAN.
In OOPS C++ , Java.

Q.1(b) Explain benefits and applications of OOP. [5]

Ans.: Benefits of OOP :

OOP offers several benefits to both the program designer and the user, Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology' promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems. Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

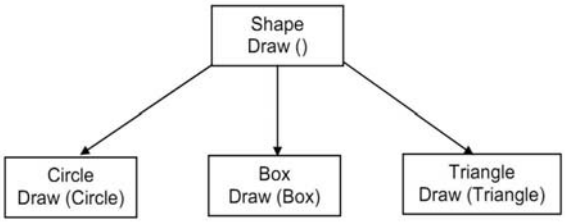
Applications of OOP :

- Real-time systems
- Simulation and modeling

- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

Q.1(c) Explain Polymorphism. Give example. **[5]**

Ans.:
- Polymorphism is important oops concept. It means ability to take more than one form.
- In polymorphism an operations may shows different behavior in different instances. The behavior depends upon the type of data used in the operation. For Ex- Operation of addition for two numbers, will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.
- The process of making an operator to show different behavior in different instance is called as operator overloading. C++ support operator overloading.
- For Example :



The above figure shows concept of function overloading. Function overloading means using a single function name to perform different types of tasks.

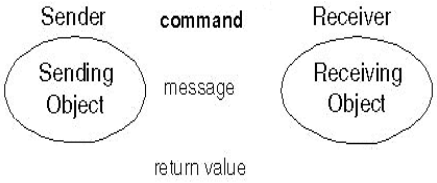
Q.1(d) Explain Class and Object. **[5]**

Ans.: **Object :**

An object can be considered a "thing" that can perform a set of activities. The set of activities that the object performs defines the object's behavior. For example, a "**StudentStatus**" object can tell you its grade point average, year in school, or can add a list of courses taken. A "**Student**" object can tell you its name or its address.

The object's interface consists of a set of commands, each command performing a specific action. An object asks another object to perform an action by sending it a message. The requesting (sending) object is referred to as sender and the receiving object is referred to as receiver.

Control is given to the receiving object until it completes the command; control then returns to the sending object. For example, a **School** object asks the **Student** object for its *name* by sending it a *message* asking for its name. The receiving **Student** object returns the *name* back to the sending object.



Class :

"A class is the implementation of an abstract data type (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively. Instances of classes are called objects. Consequently, classes define properties and behaviour of sets of objects." Class is a collection of member data and member functions. objects contain data and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of type class. Once a class has -been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created : A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit: Classes are user-defined data

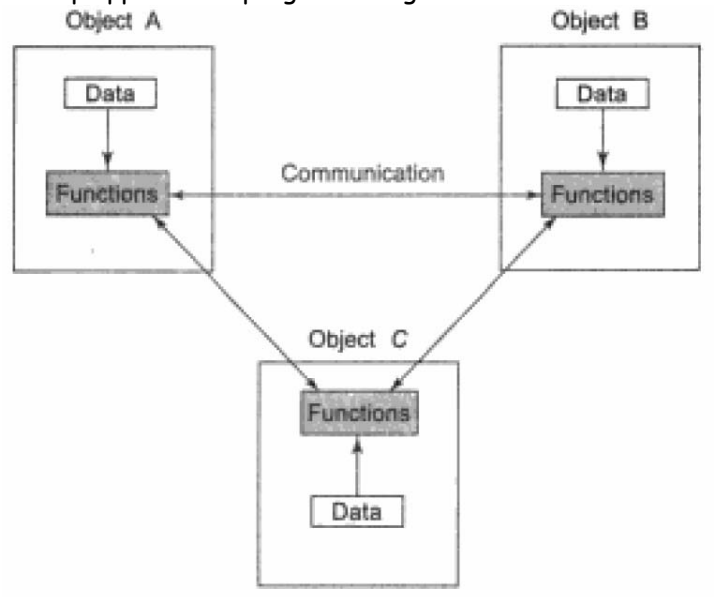
types and behave like the built-in types of a programming language. For example, the syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement fruit mango; will create an object mango belonging to the class fruit.

Q.1(e) Explain Object Oriented Paradigm. **[5]**

Ans.: The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allow decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are :

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions,
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.



Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define "object-oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand* Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions arc independent, the objects can be used in a variety of different programs without modifications.

Q.1(f) What is inheritance? State its types. **[5]**

Ans.: **Inheritance** is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the

bird 'robin1 is a part of the class 'flying bird' which is again a part of the class bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object mango belonging to the class fruit.

Types of inheritance :

- single inheritance
- Multiple inheritance
- multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.

Q.2 Attempt the following (any THREE) [15]

Q.2(a) What is friend function? Give example. [5]

Ans.: Friend Function :

- A non member function cannot have an access to the private data of a class. However there could be situation where we would like two classes to share a particular function.
- In such situation, c++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes such a function need not be a member of any of these classes.
- The syntax for friend function is

```
class ABC
{
=======
public:
=======
friend void function1(void);
}
```

- The function is declared with friend keyword. But while defining friend function. It does not use either keyword friend or :: operator. A friend function, although not a member function, has full access right to the private member of the class.
- A friend, function has following characteristics.
  - It is not in the scope of the class to which it has been declared as friend.
  - A friend function cannot be called using the object of that class. If can be invoked like a normal function without help of any object.
  - It cannot access the member variables directly & has to use an object name dot membership operator with member name.

Example : Friend Function

```
#include <iostream>
using namespace std;
class sample
{
    int a;
    int b;
public:
    void setvalueO {a=25; b=40; }
    friend float mean{sample s);
};
float mean(sample s)
{
    return float (s.a + s.b)/2.0;
}

int main( )
{
    sample X;        // object X
    X.setvalue( );
    cout << "Mean value = " << mean(X)  << "\n",

    return ( );
}
```

Q.2(b) Explain Constructors. **[5]**

Ans.: Constructor is a 'special' member function whose task is to initialize the object of it's class.

A ***constructor*** is a special method that creates and initializes an object of a particular class. It has the same name as its class and may accept arguments. In this respect, it is similar to any other function.

If you do not explicitly declare a constructor for a class, the C++ compiler automatically generates a default constructor that has no arguments.

A constructor is declared and defined as follows:

```
// class with a constructor
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, These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return values, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member or union.
- They make 'implicit calls' to the operators new and delete when memory allocation is required.

Q.2(c) Explain Array of pointer to object by giving programming example. [5]

Ans.: Array of pointer to object :

```
#include <iostream>
#include <cstream>

using namespace std;

class city
{
    protected;
        char *name;
        int len;
    public:
        city( )

        {
            len = 0;
            name = new char[len + 1];
        }
        void getname(void)
        {
            char *s;
            s = new char[30];

            cout << "Enter city name:";
            cin >> s;
            len = strlen (s);
            name = new char[len + 1];
            strcopy(name, s);
        }
        void printname(void)
        {
            cout << name << "\n";
        }
};
```

```
int main( )
{
    city *cptr[10];          // array of 10 pointers to cities
    int n = 1;
    int option;

    do;
    {
        cptr[n] = new city;  //create new city
        cptr[n] –> getname( );
        n++;
        cout << "Do you want to enter one more name?\n";
        cout << "(Enter 1 for yes 0 for no):"
        cin >> option;
    }
    while(option);

    cout << "\n\n";
    for(int i = 1; i <= n; i++);
    {
        strcpy[i] –> printname( );
    }
    return 0;
}
```

Q.2(d) Explain Destructors. **[5]**

Ans.: A **destructor**, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde(~). For example , the destructor for the class String can be defined as shown below:

```
~String( ) { }
```

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

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory.

Q.2(e) Write an Object Oriented Program to read and display student information (roll number, name etc.) **[5]**

Ans.:

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
    char name[20];
    int roll;
    public:
    void getdata();
    void display();
};
void student::getdata()
{
```

```
    cout<<"Enter name:";
    cin>>name;
    cout<<"Enter roll:";
    cin>>roll;
}
void student::display()
{
    cout<<"NAME:"<<name<<endl;
    cout<<"ROLL:"<<roll<<endl;
}
void main()
{
    student a,b;
    clrscr();
    a.getdata();
    b.getdata();
    a.display();
    b.display();
    getch();
}
```

Q.2(f) Explain the concept passing object as an argument. **[5]**

Ans.: Objects as Function Arguments

Like any other data type, an object may be used as a function argument . This can be done in two ways :

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes mode to the object inside the function do not affect the object used to call the function. The second method is called pass-by-reference, When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Example : **Object as Arguments :**

```
#include <iostream>
using namespace std;
class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    { hours = h; minutes * m; }
    void puttime(void)
    {
        cout << hours << " hours and ";
        cout << minutes << " minutes " << "\n";
    }
    void sum(time, time);                    //declaration with objects as argume
};
```

```
void time :: sum(time t1, time t2)        // t1, t2 are objects
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = nvinutes%60;
    hours = hours + t1.hours + t2.hours;
}
int main()
{
    time T1, T2, T3;

    T1.gettime(2, 45);      // get T1
    T2.gettime(3, 30);      // get T2
    T3.sum(T1, T2);         //T3 = T1 + T2

    cout <<"T1 = "; T1.puttime( );        //display T1
    cout <<"T2 = "; T1.puttime( );        //display T2
    cout <<"T3 = "; T1.puttime( );        //display T3

    return 0;
}
```

Q.3 Attempt the following (any THREE) : [15]

Q.3(a) Explain static functions with example. [5]

Ans.: Static Member Functions

Like static member variable, we can also have static member functions, A member function that is declared static has the following properties;

- A static function can have access to only other static members (functions or variables) declared in the same class.
- A static member function can be called using the<lass name (instead of its objects) as follows:

  class-nome :: function-name;

Following program :

Illustrates the implementation of these characteristics. The static function showcount( ) displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable count,

The function showcode( ) displays the code number of each object.

Static Member function :

```
#include <iostream>
using namespace std;
class test
{
    int code;
    static int count;         //static member.variable

public:
    void setcode(void)
    {
        code = ++count;
    }
    void showcode(vold)
    {
```

```
            cout << "object number: " << code << "\n";
        }
        static void showcount(void)          //static member function
        {
            cout << "count: * << count << "\n";
        }
};
int test :: count;
int main()
{
    test t1, t2;
    t1.setcode();
    t2.setcode();

    test :: showcount();          // accessing static function
    test t3;
    t3.setcode();

    test :: showcount();

    t1.showcode();
    t2.showcode();
    t3.showcode();

    return();
}
```

Output :
count : 2
count : 3
object number : 1
object number : 2
object number : 3

Q.3(b) What is function overloading? Write a program to find area of circle, triangle and rectangle using the concept of function overloading. [5]

Ans.:

- Function overloading refers to creating multiple functions with the same name but different parameter list.
- Different parameter list can be with reference to the number of parameters or the type of parameters.
- Since, for overloaded functions the functions have same name, a function can be called with the common name but the function that will be invoked is based on the parameter type and number of parameters.
- This concept of function overloading is also called as function polymorphism.

```
# include<iostream.h>
# include<conio.h>
# include<math.h>
float area(float r)
{
    return(3.14*r*r);
}
float area(float 1, float b)
{
```

```
    return(l*b);
}
float area(float a, float b, float c)
{
    float s,ar;
    s=(a+b+c)/2;
    ar=s*(s-a)*(s-b)*(s-c);
    ar=pow(ar,0.5);
    return ar;
}
void main()
{
    clrscr();
    int choice;
    float x,y,z,a;
    cout<<"1.Area of Circle\n2.Area of Rectangle\n3.Area of Triangle\nEnter your choice:";
    cin>>choice;
    switch(choice)
    {
        case 1:cout<<"Enter the radius of the circle:";
        cin>>x;
        a=area(x);
        cout<<"The area of the circle="<<a;
        break;
        case 2:cout<<"Enter the length and breadth of the rectangle:";
        cin>>x>>y;
        a=area(x,y);
        cout<<"The area of the rectangle="<<a;
        break;
        case 3:cout<<"Enter the length of the three sides of the triangle:";
        cin>>x>>y>>z;
        a=area(x,y,z);
        cout<<"The area of the triangle="<<a;
        break;
        default:cout<<"Invalid Choice";
    }
    getch();
}
```

Output:

1. Area of Circle
2. Area of Rectangle
3. Area of Triangle

Enter your choice : 2
Enter the length and breadth of the rectangle : 3.5
2
The area of the rectangle = 7

Q.3(c) What is operator overloading? State the rules for overloading. [5]

Ans.:

- The mechanism of assigning a new meaning to an already existing operator is called as operator overloading.
- There are some rules necessary to be known for operator overloading. Below is a list of these rules.

1. Only existing operators can be given a new meaning i.e. only existing operators can be overloaded.
2. Even after overloading the basic meaning of the operator remains the same.
3. Overloaded operators follow the same syntax as that of the original operators.
4. Unary operators overloaded by means of member function can accept no parameters.
5. Unary operators overloaded by means of friend function can accept up to one parameter.
6. Binary operators overloaded by means of member function can accept up to one parameter.
7. Binary operators overloaded by means of friend function can accept up to two parameters.
8. Some of the operators cannot be overloaded viz.
   a) sizeof()
   b) member operator (i.e. '.')
   c) pointer to member operator (i.e. ".*")
   d) scope resolution operator (i.e. "::")
   e) conditional operator (i.e. "?.")
9. Some of the operators cannot be overloaded by friend functions viz.
   a) assignment operator (i.e. "=")
   b) function call operator (i.e. "( )")
   c) subscription operator (i.e. "[ ]")
   d) class member access operator (i.e. "–>")
10. Some of the rules are very simple, but some you may not understand now. We will understand these rules with program examples.
11. The declaration of the function that is meant for operator overloading has a special syntax as given below :

    return_type operator op (argument list);

    where "op" is to be replaced by the symbol of the operator to be overloaded.

Q.3(d) Explain pure virtual function. [5]

Ans.: A pure virtual function is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form :

```
virtual return-type function-name (parameter-list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result. Hence definition for pure virtual function must be there in the derived class.

```
#include<iostream.h>
#include<conio.h>
class Base
{
    protected:
    int a,b;
    public:
    virtual void read()
    {
    }
    virtual void display() = 0;
};

class Sub: public Base
{
    protected:
```

```
    int c,d;
    public:
    void read()
    {
        cout<<"Enter 4 values:";
        cin>>a>>b>>c>>d;
    }
    void display()
    {
        cout<<"The values are:"<<a<<"\n"<<b<<"\n"<<c<<"\n"<<d;
    }
};
void main()
{
    clrscr();
    Sub s;
    Base *ptr;
    ptr=&s;
    ptr->read();
    ptr->display();
    getch();
}
```

Output :

```
Enter 4 values:1
2
3
4
The values are:1
2
3
4
```

Q.3(e) Explain data conversion between objects and basic types. [5]

Ans.: Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types, We must, therefore, design the conversion routines by ourselves, if such operations are required,

Three types of situations might arise in the data conversion between uncompatible types:

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

Basic to Class type :

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. These are all examples where constructors perform a defaeto type conversion from the argument's type to the constructor's class type.

Let us consider example of converting an int type to a class type.

```
class time
{
```

```
    int hrs;
    int mins;
public:
    ……
    ……
    time (int t)              // constructor
    {
        hours = t/60;         // t in minutes
        mins = t%60;
    }
};
```

The following conversion statements can be used in a function :

```
    time T1;                  // object T1 created
    Int duration = 85;
    T1 = duration;            // int to class type
```

After this conversion, the hrs member of T1 will contain a value of 1 and mins member a value of 25, denoting 1 hours and 25 minutes.

Class to Basic Type :

The constructors did a flue job in type conversion from a basic to class type. What about the conversion from a class to basic typo? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function, is :

```
operator typename( )
{
    ……
    ……    (Function statements)
    ……
}
```

This function converts a class type data to typename. For example, the operator double() converts a class object to type double, the operator into converts a class type object to type int, and so on.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

One Class to Another Class Type :

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example :

```
    obj X = obj Y;        // objects of different types
```

obj X is an object of class X and obj Y is an object of class Y. The class Y type data is converted to the class X type data and the converted value is assigned to the obj X. Since the conversion takes place from class Y to class X, Y is known as the source class and X is known as the destination class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do

we decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

Conversion Required	Conversion takes place in	
	Source Class	Destination Class
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

Q.3(f) Explain this pointer. **[5]**

Ans.: C++ uses a unique keyword called this to represent an object that invokes a member function, this is a pointer that points to the object for which this function was called. For example, the function call A.max() will set the pointer this to the address of the object A. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions. Consider the following simple example:

```
class ABC
{
    int a;
    ……
    ……
};
```

The private variable 'a' can be used directly inside a member function, like

```
a – 123;
```

We can also use the following statement to do the same job :

```
this–>a = 123;
```

Since C++ permits the use of shorthand form a = 123, we have not been using the pointer this explicitly so far. However, we have been implicitly using the pointer this when overloading the operators using member function.

Example :

```
#include <iostream>
#include <cstring>
using namespace Std;
class person
{
    char name[20];
    float age;
public:
    person(char *s, float a)
    {
        strcpy(name, s);
        age = a;
    }
    person 4 person :: greater(person & x)
    {
        if(x.aqe >= age)
                return x;
        else
                return *this;
    }
    void display(void)
    {
```

```
        cout << "Name: "  << name << "\n"
             << "Age: " << age << "\n";
    }
};

int main( )
{
    person P1("John", 37.50),
        P2("Ahmed", 29.0),
        P3("Hebber", 40.25);

    person P = P1.greater(P3);          //P3.greater(P1)
    cout « "Elder person is: \n";
    P.display();

    P = P1.greater(P2);                 //P2.greater(P1)
    cout « "Elder person is; \n";
    P.display();

    return ();
}
```

Q.4 Attempt the following (any THREE) **[15]**

Q.4(a) Explain access specifiers/Visibility modes. **[5]**

Ans.:

- The members of a class may be public, protected or private and the class may also be derived in the following ways :
  1. public 2. protected 3. private
- These keywords viz. public, protected and private are called as "access specifiers", as they decide the access of a member.
- The visibility of base class members within the derived class in as shown in Table

Table : Visibility of base class members in the derived class

Base class member visibility	Derived class visibility		
	Public derivation	Protected derivation	Private derivation
Public members	Public	Protected	Private
Protected members	Protected	Protected	Private
Private members	Not inherited	Not inherited	Not inherited

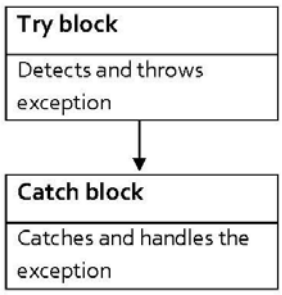
- Hence, you will notice in the Table, in case of public derivation the public and protected members of the base class remain in the same visibility even in the derived class. The private members are never derived.
- In case of protected derivation the public and protected members of the base class become protected in the derived class. The private members are never derived.
- In case of private derivate derivation the public and protected members of the base class become private in the derived class. The private members are never derived.
- There are various types of inheritance namely,
  1. Single Inheritance 2. Multilevel Inheritance 3. Multiple Inheritance
  4. Hybrid Inheritance 5. Hierarchical Inheritance
- To derive a class from another we need to use the following syntax:
  class derived_class_name : access_specifier base_class_name

```
class alpha
{
    private:            //optional
        ……              //visible to member functions
        ……              //within its class
    protected:
        ……              //visible to member functions
        ……              //of its own and derived class
    public:
        ……              //visible to all functions
        ……              //in the program
};
```

Q.4(b) Explain exception handling mechanism. **[5]**

Ans.: Exception handling mechanism

- C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.
- Try block hold a block of statements which may generate an exception.
- When an exception is detected, it is thrown using a throw statement in the try block.



- A try block can be followed by any number of catch blocks.

The general form of **try** and **catch** block is as follows :

```
try
{
    /* try block; throw exception*/
}
catch (type1 arg)
{
    /* catch block*/
}
……
……
catch (type2 arg)
{
    /* catch block*/
}
```

Q.4(c) Explain containership. **[5]**

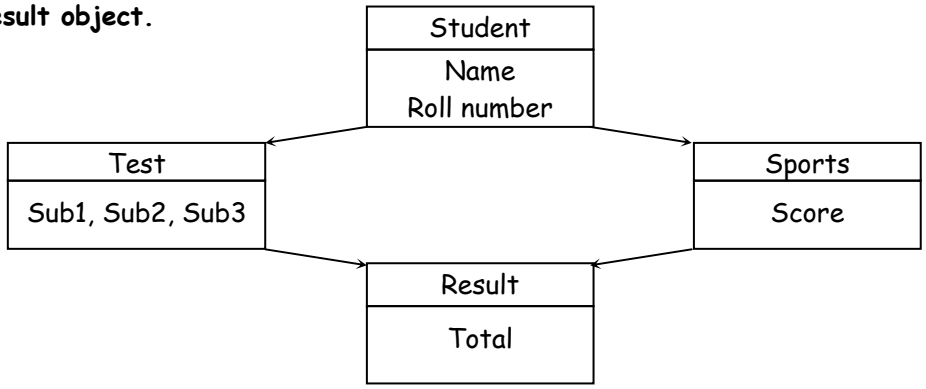
Ans.: Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes, C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below :

```
class alpha {……};
class beta {……};
class gamma
```

```
{
    alpha a;            // a is an object of alpha class
    beta b;             // b is an object of beta class
    ……
};
```

All objects of gamma class will contain the objects a and b. This kind of relationship is called containership or nesting. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Q.4(d) Write a program to implement following inheritance tree to create, read and display result object. **[5]**



Ans.:

```
#include <iostream>
using namespace std;
class student
{
    protected:
        int roll_number;
    public:
        void get_number(int a)
        {
            roll_number = a;
        }
        void put_number(void)
        {
            cout << "Roll No: " << roll_number << "\n";
        }
};
class test : virtual public student
{
    protected:
        float part1, part2;
    public:
        void get_marks(float x, float y)
        {
            part1 = x; part2 = y;
        }
        void put_marks(void)
        {
```

```
            cout << "Marks obtained: " << "\n"
                 << "Part1 =" << part1 << "\n"
                 << "Part2 =" << part2 << "\n";
        }
};
class sports : public virtual student
{
    protected:
        float score:
    public:
        void get_score(float s)
        {
            score = s;
        }
        void put_score(vold)
        {
            cout << "Sports wt:" << score << "\n\n";
        }
};
class result : public test, public sports
{
    float total;
    public :
        void dlsplay(void);
};
void result :: display(void)
{
    total = part1 + part2 + score;

    put_number();
    put_marks();
    put_score();

    cout << "Total Score; " << total << "\n";
}

int main()
{
    result student_1;
    student_1.get_number(678);
    student_1.get_marks(30.5, 25.5);
    student_1,get_score(7.0);
    student_1.display();

    return ();
}
```

Q.4(e) Explain what is an error and its types. Also explain an Exception and its types. [5]

Ans.: We know that it is very rare that a program works correctly first time. It might have bugs. The two most common types of bugs are logic errors and syntactic errors. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language itself We can detect these error- by using exhaustive debugging and testing procedures.

We often come across some peculiar problems other than logic or syntax errors. They are known as exceptions. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space. When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. ANSI C++ provides built-in language features to detect and handle exceptions which are basically run time errors.

Exceptions are of two kinds, namely, synchronous exceptions and asynchronous exceptions. Errors such as "out-of-range index" and "over-flow" belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

Q.4(f) Explain constructors in derived classes. [5]

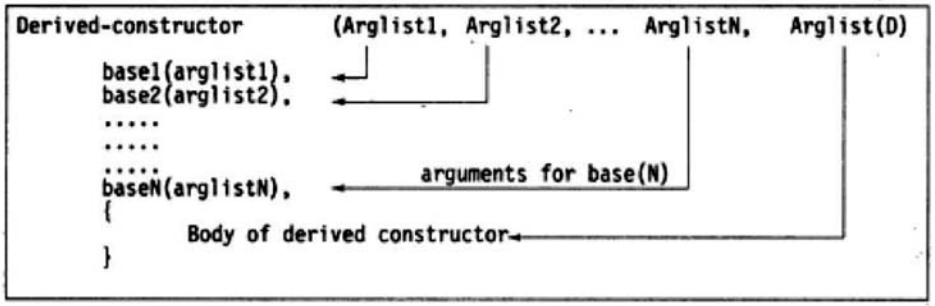
Ans.: As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any hose class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructors will he executed in the order of inheritance.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a deriving structure is :



The header line of derived constructor function contains two parts separated by a colon(:). The first part provides the declaration of the arguments that are passed to the derived-constructor and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2) ... are function calls to base constructors base1(), base2(), ... and therefore arglist1, arglist2, ... etc. represent the actual parameters that are passed to the base constructors. Arglist1 through ArglistN are the argument declarations for base constructors base1 through baseN. ArglistD provides the parameters that arc necessary to initialize the members of the derived class.

Example:

```
D{int a1, int a2, float b1, float b2, int d1):
A(a1, a2),              /* call to constructor A */
B(b1, b2)               /* call to constructor B */
{
    d = d1;             //executes its own body
}
```

A(a1, a2) invokes the base constructor A() and B(b1, b2) invokes another base constructor B(). The constructor D() supplies the values for these four arguments. In addition, it has one argument of its own. The constructor D() has a total of five arguments. D() may be invoked as follows:

```
.....
D objD(5, 12, 2.5, 7.54, 30);
.....
```

These values ore assigned to various parameters by the constructor D() as follows:

5 → a1
12 → a2
2.5 → b1
7.54 → b2
30 → d1

Q.5 Attempt the following (any THREE) [15]

Q.5(a) Explain with example class templates with multiple parameters. [5]

Ans.: We can use more than one generic data type in a class template. They are declared as a comma-separated list within the template specification as shown below:

```
template<class T1, class T2, ...>
class classname
{
    .....
    .....    (Body of the class)
    .....
}
```

Two generic data types in a class definition :

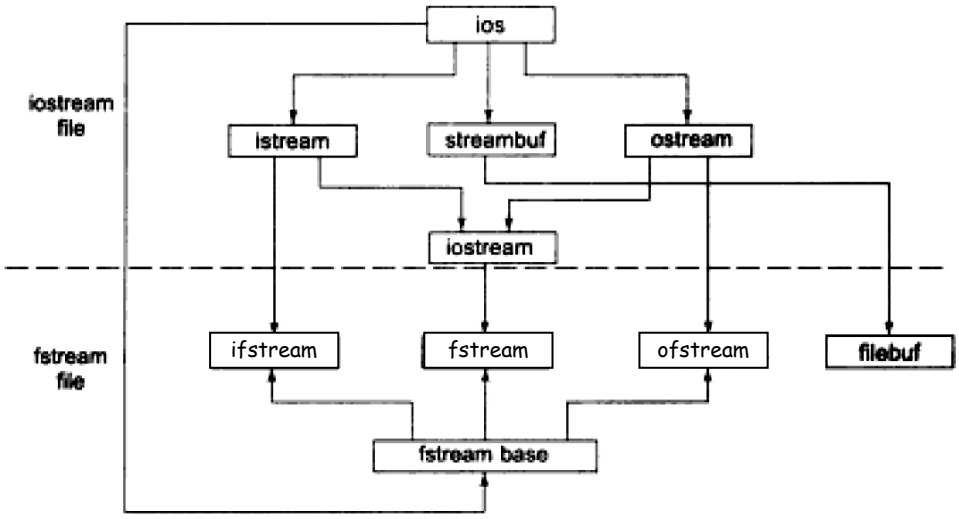
```
#include <iostream>;
using namespace std;
template<class T1, class T2>
class Test
{
    T1 a;
    T2 b;
public:
    Test(T1 x, T2 y)
    {
        a = x;
        b = y;
    }
    void show()
    {
```

```
            cout << a << " and " << b "\n";
        }
};
int main()
{
    Test <float, int> test1 (1.23,123);
    Test <int.char> test2 (100. "W");

    test1.show();
    test2.show();

    return 0;
}
```

The output of Program :
1.23 and 123
100 and W

Q.5(b) Explain the hierarchy of file stream classes. [5]

Ans.: The I/O system of C++ contains a set of classes that define the file handling methods, These include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class as shown in Figure. These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.



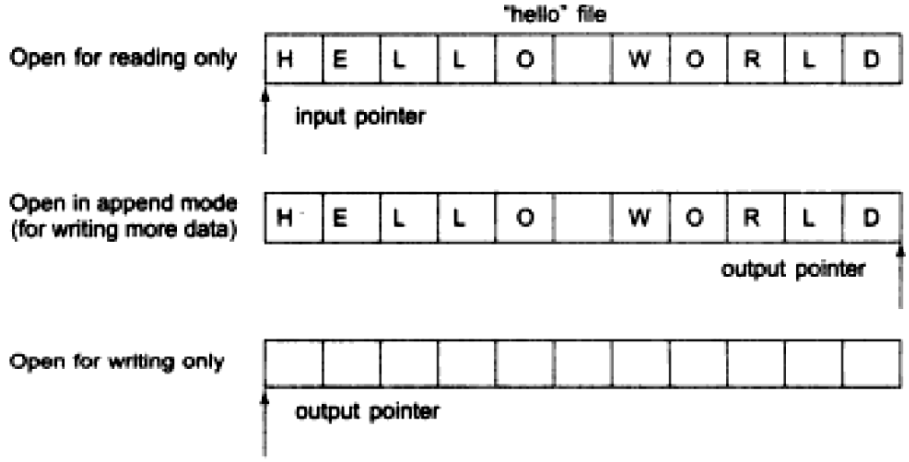
Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open() of file stream classes, Also contain close() and open() as members.
fstreambase	Provides operations common to the file streams, Serves as a base for fstream, ifatrcam and ofstream class. Contain open() and clone() functions.
ifstream	Provides input operations. Contains open() with default input mode. Inherits the functions get(), getlined, read(), seekg() and tellg() functions from istream
Ofstream	Provides output operations. Contains open() with default output mode Inherits put(), seekp(), tellp(), and write(), functions from ostream.
fstream	Provides support for simultaneous input and output operations. Contains open() with default input mode Inherits all the functions from istream and ostream classes through iostream

Q.5(c) Explain file pointers and their manipulations. [5]

Ans.: Each file has two associated pointers known as the file pointers. One of them is called the input pointer (or get pointer) and the other is called the output pointer (or put pointer). We can use these pointers to move through the files while reading or writing, The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

Default Actions :

When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start. Similarly, when we open a file in write-only mode, the existing contents are deleted and the output pointer is set at the beginning. This enables us to write to the file from the start. In case, we want to open an existing file to add more data, the file is opened in append' mode. This moves the output pointer to the end of the file (i.e. the end of the existing contents).



The File :

stream class support the following functions to manage such situation :

- seekg() Moves get pointer (input) to a specified location.
- seekp() Moves put pointed (output) to a specified location.
- tellg() Gives the current position of the get pointer.
- tellp() Gives the current position of the put pointer.

For example, the statement

```
infile.seekg(10);
```

moves the file pointer to the byte number 10, Remember, the bytes in a file arc numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

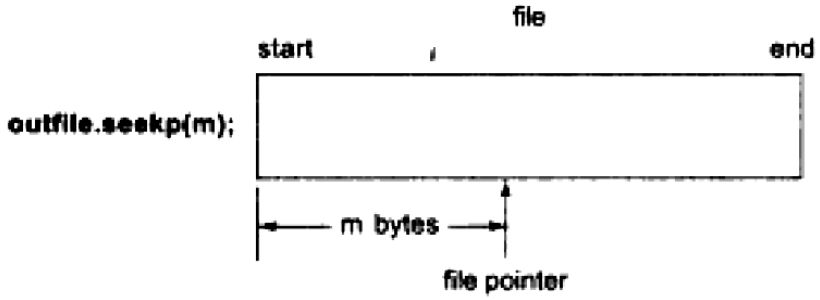
Consider the following statements:

```
ofstream fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of the file 'hello" and the value of p will represent the number of bytes in the file.

Specifying the offset

We have just now seen how to move a file pointer to a desired location using the 'seek' functions. The argument to these functions represents the absolute position in the file. This is shown in Figure.



'Seek' function seekg() and seekp() can also be used with two arguments as follows :

seekg(offset, refposition);

seekp(offset, refposition);

The parameter offset represents the number of bytes the file pointer is to be moved from the locaiton specified by the parameter refposition. The refposition takes one of the following three constants defined in the ios class :

- ios::beg start of the file
- ios:cur current position of the pointer
- ios::end End of the file

Seek call	**Action**
fout.seekg(o, ios::beg);	Go to start
fout.seekg(o, ios::cur);	Stay at the current position
fout.seekg(o, ios::end);	Go to the end of file
fout.seekg(m, ios::beg);	Move to (m + 1)th byte in the file
fout.seekg(m, ios::cur);	Go forward by m byte form the current position
fout.seekg(–m, ios::cur);	Go backward by m bytes from the current position
fout.seekg(–m, ios::end);	Go backward by m bytes form the end

Q.5(d) Write a program with generic function or template to swap multiple types of data. **[5]**

Ans.:

```
#include<conio.h>
#include<iostream.h>
template<class x> void swap(x &a, x &b)
{
    x temp;
    temp=a;
    a=b;
    b=temp;
}
void main()
{
    int a=4,b=5;
    float p=3.5,q=4.2;
    cout<<"a="<<a<<"\tb="<<b<<" before calling swap function"<<endl;
    swap(a,b);
    cout<<"a="<<a<<"\tb="<<b<<" after calling swap function"<<endl;
    cout<<"p="<<p<<"\tq="<<q<<" before calling swap function"<<endl;
    swap(p,q);
    cout<<"p="<<p<<"\tq="<<q<<" after calling swap function"<<endl;
    getch();
}
```

Output :

a=4 b=5 before calling swap function
a=5 b=4 after calling swap function
p=3.5 q=4.2 before calling swap function
p=4.2 q=3.5 after calling swap function

Q.5(e) Write a program to read data from user and write to the file. [5]

Ans.: Program

Uses a single file for both writing and reading the data. First, it takes data from the keyboard and writes it to the file. After the writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

```
// Creating files with constructor function
#include <iostream.h>
#include <fstream.h>
int main()
{
    of stream outf {"ITEM"};          //connect ITEM file to outf
    cout << "Enter item name:";
    char name[30];
    cin >> name;                      //get name from key board and
    outf << name << "\n";             //write to file ITEM

    cout << "Enter Item cost:";
    float cost;
    cin >> cost;                      //get cost from key board and
    outf << cost << "\n";             // write to file ITEM

    outf,close();
    ifstream inf("ITEM");
    inf >> name;
    inf >> cost;
    cout << "\n";
    cout << "Item name;" << name << "\n";
    cout << "Item cost:" << cost << "\n";
    inf.close();                      //Disconnect ITEM front inf
    return();
}
```

Q.5(f) Explain with example overloading of template functions. [5]

Ans.: A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

- Call an ordinary function that has an exact match.
- Call a template function that could be created with an exact match.
- Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions :

```
#include <iostream>
#include <string>
using namespace std;
template <class T>
void display(T x)
{
```

```
    cout << "Template display: " << x << "\n";
}
void display(int x)                 // overloads the generic display ()
{
    cout << "Explicit display: " << x << "\n";
}
int ma>n()
{
    display(100);
    display(12.34);
    display("C");
    return ();
}
```

❑ ❑ ❑ ❑ ❑