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) Explain encapsulation? [5]

(A)
- The wrapping up of data and functions into a single unit (called class) is known as encapsulation.
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called data hiding.

Q.1(b) List advantages of oop? [5]

(A) **(i) Simplicity :** Software objects model real world objects, so the complexity is reduced and the program structure is very clear.

(ii) Modularity : Each object forms a separate entity whose internal workings are decoupled from other parts of the system.

(iii) Modifiability : It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

(iv) Extensibility : Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

(v) Maintainability : Objects can be maintained separately, making locating and fixing problems easier.

(vi) Re-usability : Objects can be reused in different programs.

Q.1(c) Explain principles of oops. [5]

(A) Object oriented programming gives significance to objects or data rather than the procedure.

1. More emphasis is given to data rather than procedure. It is seen in the real world problems that the data or the objects are more important than the procedure or the method to perform a task. Hence, in object oriented programming, object is given more importance compared to the procedure of doing it.
2. Programs are divided into objects. The objects contain data and functions required to access them. Thus, in an object oriented program, you will notice the objects as shown in the structure.
3. Data structures are designed such that they characterize the object i.e. all the information required for an object are stored in the variables of that object.
4. Functions that operate on the data of an object are tied together in data structures i.e. the functions that operate on a data are associated with them.
5. Data is hidden or cannot be access by external function. The data of an object can be accessed only by the functions of the same object.
6. Objects communicate with each other through functions. If a function of an object wants the data of another object then it can be accessed only through the functions.
7. New data and functions are easily added when required. Whenever new data is to be added, all the functions need not be changed; only that functions are to be changed, which require to access the data.
8. It follows a bottom-up approach. In this type of approach, each element is first specified in detail and then the entire system is made using these elements. You will notice in the programming of C++, that this approach of bottom-up approach makes the programming very simple.

Q.1(d) Explain disadvantages of procedure oriented programming. **[5]**

(A) (i) In POP Main program is divided into small parts depending on the functions.
(ii) In POP There is no perfect way for data hiding.
(iii) In POP Top down process is followed for program design.
(iv) In POP Importance is given to the sequence of things to be done.
(v) In POP Mostly functions share global data i.e data move freely around the system from function to function.
(vi) In POP No access specifier.
(vii) In POP Operator cannot be overloaded.
(viii) In POP C, Pascal, FORTRAN.

Q.1(e) Explain data abstraction. **[5]**

(A)
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes.
- They encapsulate all the essential properties of the objects that are to be created.
- Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

Q.1(f) List applications of oops.

(A) The object oriented programming finds a lot of applications in the real business systems. The various places where object oriented programming is required are:
1. Database management systems
2. Intelligent systems
3. Pattern Recognition
4. Image processing
5. Parallel computing
6. Mobile computing
7. Distributed computing
8. Data security etc.

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

Q.2(a) Explain internally defined member function with an example. **[5]**

(A) Internally Defined functions
When the member functions of a class are defined inside the class itself they are called internally defined member funtions. The syntax for the same is as follows

```
Return_type class_name :: function_name (parameter_list)
{
// body of the member function
}
```

Examples
Program to demonstrate internally defined member functions

```
class book
{
char title[30];
float price;
public:
void getdata(char [],float); II declaration
void putdata()//definition inside the class
{
cout<<"\nTitle of Book: "<<title;
cout<<"\nPrice of Book: "<<price;
} ;
```

Q.2(b) Explain externally defined member function with an example. [5]

(A)
- When the member functions of a class are defined outside the class they are called as externally defined member functions.
- The functions when defined outside must use scope resolution operator to define the scope of the function to be within the class. The syntax of defining a function outside the class using the scope resolution operator is as given below :

```
Return_type Class_name :: Function_name(argument list)
{
    –
    Statements;
    –
}
```

- Also in this case the prototype declaration has to be done inside the class for the externally defined functions.

Q.2(c) Write a program to add two complex numbers using friend function. [5]

(A) 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.

Q.2(d) Explain default constructor with a program example. [5]

(A)
- If a constructor does not have any parameter list i.e. it cannot accept any parameters than it is called as a default constructor.
- Thus, the default constructor has the blank brackets indicating no parameters can be accepted.
- Let us initialize the values of the variable radius of the class circle, using default constructor and the value of the variable 'n' for the below programs.

A Default constructor is that will either have no parameters, or all the parameters have default values. If no constructors are available for a class, the compiler implicitly creates a default parameterless constructor without a constructor initializer and a null body.

```
#include <iostream.h>
class Defal
{
public:
```

```
int x;
int y;
Defal(){x=y=0;}
};
int main()
{
Defal A;
cout << "Default constructs x,y value::"<<
A.x <<" , "<< A.y << "\n";
return 0;
}
```

Q.2(e) Explain inline function. **[5]**

(A)

```
# include<iostream.h>
# include<conio.h>
class series
{
    int n,i,sum;
    public:
    series()
    {
        cout<<"Enter the value of n:";
        cin>>n;
        sum=0;
    }
    void compute();
    void display();
};
inline void series::compute()
{
    for(i=1;i<=n;i++)
    {
        sum=sum+i*i;
    }
}
inline void series::display()
{
    cout<<"Value of the series = "<<sum;
}
void main()
{
    clrscr();
    series s;
    s.compute();
    s.display();
    getch();
}
```

Output

```
Enter the value of n : 5
Value of the series = 55
```

Q.2(f) Explain parameterized constructor with program example. [5]

(A)

```
# include<iostream.h>
# include<conio.h>
class circle
{
    float r,a;
    public:
    circle(float x)
    {
        r=x;
    }
    void compute();
    void display();
};
inline void circle::compute()
{
    a=3.14*r*r;
}
inline void circle::display()
{
    cout<<"Area="<<a;
}
void main()
{
    clrscr();
    float p;
    cout<<"Enter the radius of the circle";
    cin>>p;
    circle c(p);
    c.compute();
    c.display();
    getch();
}
```

Output :

```
Enter radius:5
Area=78.5
```

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

Q.3(a) Write a program to calculate the area of rectangle, triangle, circle using function overloading. [5]

(A)

```
# 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(b) Explain operator overloading? Write the rules of operator overloading. [5]

(A)
- 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(c) Write a program to negate the values of two variables contained in the object. [5]

(A)
```
#include<iostream.h>
#include<conio.h>
class Negate
{
    int x,y;
    public:z
    void read()
    {
        cout<<"Enter two numbers";
        cin>>x>>y;
    }
    void compute()
    {
        x=–x;
        y=–y;
    }
    void display()
    {
        cout<<"x="<<x<<endl<<"y="<<y;
    }
};
void main()
{
    clrscr();
    Negate n;
    n.read();
    n.compute();
    n.display;
    getch();
}
```

Output
```
Enter two numbers 4
6
x=–4
y=–6
```

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

(A)
```
#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 polymorphism and its types. **[5]**

(A) **Polymorphism means one name, multiple forms.**

For example, the + (plus) operator in C++ will behave different for different data types :

4 + 5 <-- integer addition
3.14 + 2.0 <-- floating point addition
"Good" + "Boy" <-- string concatenation

Polymorphism is of two types:

(i) Compile time (Static) polymorphism (static binding or static linking or early binding)

- In compile time polymorphism, all information needed to call a function is known during program compilation.
- Example: Function overloading and operator overloading are used to achieve compile time polymorphism

(ii) Runtime (Dynamic) polymorphism (late binding or dynamic binding)

- In runtime polymorphism all information needed to call a function is known during program execution.
- Example: Virtual function is used to achieve runtime polymorphism.

Q.3(f) Explain static function. **[5]**

(A)
- When a variable or function name is preceded by the keyword "static", it becomes common to all the objects of that class. When a member becomes static, no separate memory location is allocated for that variable to each of the object i.e. whenever an object accesses static variable, it accesses the same memory location.
- A very important use of static variable is to count the number of objects created of a class.

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.

```
#include <iostream>

using namespace std;

class Box {
  public:
    static int objectCount;
    // Constructor definition
    Box(double 1 = 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) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(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:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members

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.

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

Q.4(a) Write a note on different access specifiers in C++ [5]

(A)
- 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 4.1.1, 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:

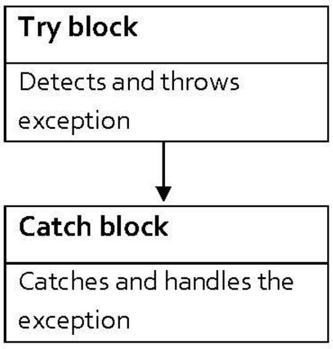
  class derived_class_name : access_specifier base_class_name

Q.4(b) Explain inheritance with its advantages. **[5]**

(A)
- The mechanism of deriving a class from another class is known as inheritance.
- The class from which another class is derived is called as the base class while the class which is derived is called as derived class.
- When a class contains an object of another class, the relationship is called as containership.
- The main advantages of inheritance are as given below :
  - Inheritance allows code reusability. The code that is written once for the objects of a class can be used by the objects off other classes derived from this class. Hence allowing the concept of code reusability.
  - The class that is once compiled need not be compiled again and again for the class using the objects of the base class. Thus reducing the time for compilation of the new code written using objects of the earlier compiled classes.

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

(A) 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.



Try block
Detects and throws exception
Catch block
Catches and handles the exception

- 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(d) What is single inheritance and explain it with program example. **[5]**

(A)
```
# include <iostream.h>
class A
{
    int i, j;
    public:
    void set(int a, int b)
    {
        i=a; j=b;
    }
    void show()
    {
    cout << i << " " << j << "\n";
    }
};
class B : public A
{
    int k;
    public:
    B(int x)
    {
        k=x;
    }
    void showk()
    {
        cout << k << "\n";
    }
};
void main()
{
    B b(3);
    b.set(1, 2);
    b.show();
    b.showk();
}
```

Output :

```
1 2
3
```

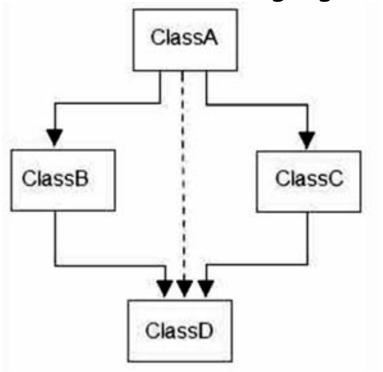
Q.4(e) What are the issues in hybrid and multiple inheritance? How are they resolved? Explain with a program example. **[5]**

(A) A derived class with two base classes and these two base classes have one common base class is called multipath inheritance.

C++ Ambiguity

Ambiguity in C++ occur when a derived class have two base classes and these two base classes have one common base class.

Consider the following figure :

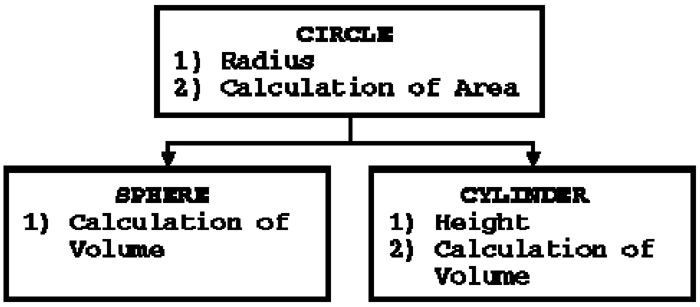


There are two ways to avoid C++ ambiguity.
Using scope resolution operator.
Using virtual base class.

1. **Avoid ambiguity using scope resolution operator.**
   Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.
   obj.ClassB::a = 10; //Statement 3
   obj.ClassC::a = 100; //Statement 4
   Note : still, there are two copies of ClassA in ClassD.

2. **Avoid ambiguity using virtual base class.**

Q.4(f) Implement the following inheritance. **[5]**
(A) Implement the following inheritance



Example Class Hierarchy

```
class Circle {
      protected:
            float Radius;
      public:
            Circle(float ra = 0.0);
            float CalcArea(void);
      };

class Cylinder : public Circle {
      protected:
            float Height;
      public:
            Cylinder(float ra = 0.0,
      float ht = 0.0);
            float CalcVolume(void);
      };

class Sphere : public Circle {
      public:
            Sphere(float ra = 0.0);
            float CalcVolume(void);
      };
```

Q.5 Attempt the following (any THREE) **[15]**
Q.5(a) What is a template? **[5]**
(A) Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

You can use templates to define functions as well as classes, let us see how do they work:
Function Template
The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list) {
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>

#include <string>


using namespace std;


template <typename T>

inline T const& Max (T const& a, T const& b)  {

  return a < b ? b:a;

}


int main () {

  int i = 39;

  int j = 20;

  cout << "Max(i, j): " << Max(i, j) << endl;


  double f1 = 13.5;

  double f2 = 20.7;

  cout << "Max(f1, f2): " << Max(f1, f2) << endl;


  string s1 = "Hello";

  string s2 = "World";

  cout << "Max(s1, s2): " << Max(s1, s2) << endl;

  return 0;

}
```

Q.5(b) Write a c++ program to read any five parameterized data type such as float, integer and print the average. **[5]**

(A)
```
#include<conio.h>
#include<iostream.h>
template<class x> float avg(x a, x b, x c, x d, x e)
{
    float average;
    average=(a+b+c+d+e)/5.0;
    return average;
}
void main()
{
    int p,q,r,s,t;
    float v,w,x,y,z;
    cout<<"Enter five integers:";
    cin>>p>>q>>r>>s>>t;
    cout<<"Average="<<avg(p,q,r,s,t)<<endl;
    cout<<"Enter five float numbers:";
    cin>>v>>w>>x>>y>>z;
    cout<<"Average="<<avg(v,w,x,y,z)<<endl;
    getch();
}
```

Output
```
Enter five integers : 1
2
3
4
5
Average=3

Enter five float numbers : 1.2
2.3
3.4
4.5
5.6
Average=3.4
```

Q.5(c) Explain how to access a file in c++ ? Explain file stream classes. **[5]**

(A) C++ provides the following classes to perform output and input of characters to/from files:
- ofstream: Stream class to write on files
- ifstream: Stream class to read from files
- fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out

ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).

For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.
File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due

Q.5(d) Explain standard template library. **[5]**

(A) The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functional, and iterators.[1]

The STL provides a set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL contains sequence containers and associative containers. The Containers are objects that store data. The standard sequence containers include vector, deque, and list. The standard associative containers are set,multiset, ap, multimap, hash_set, hash_map,hash_multiset and hash_multimap. There are also container daptors queue, priority_queue, and stack,

The STL implements five different types of iterators. These are input iterators (that can only be used to read a sequence of values), output iterators (that can only be used to write a sequence of values), forward iterators (that can be read, written to, and move forward), bidirectional iterators (that are like forward iterators, but can also move backwards) and random access iterators (that can move freely any number of steps in one operation).

It is possible to have bidirectional iterators act like random access iterators, so moving forward ten steps could be done by simply moving forward a step at a time a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.

A large number of algorithms to perform activities such as searching and sorting are provided in the STL, each implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators). Searching algorithms like binary_search and lower_bound use binary search and like sorting algorithms require that the type of data must implement comparison operator < or custom comparator function must be specified; such comparison operator or comparator function must guarantee strict weak ordering.

The STL includes classes that overload the function call operator (operator()). Instances of such classes are called functors or function objects. Functors allow the behavior of the associated function to be parameterized (e.g. through arguments passed to the functor's constructor) and can be used to keep associated per-functor state information along with the function. Since both functors and function pointers can be invoked using the syntax of a function call, they are interchangeable as arguments to templates when the corresponding parameter only appears in function call contexts.

Q.5(e) Explain overloading a template function using a non template function with a program example. **[5]**

(A) Overloading a template using non template

```
#include <iostream>
using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
  f( 1 ,  2 );
  f('a', 'b');
  f( 1 , 'b');
}
```

The following is the output of the above example:

```
Non-template
Template
Non-template
```

Q.5(f) Write a program to write and read string, integer and float from to a file. **[5]**

(A)

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
    char name[20];
    int roll;
    float fee;
    ofstream o;
    o.open("test0",ios::in);
    o<<"Ajay"<<endl;
    o<<24<<endl;
    o<<55988.45<<endl;
    o.close();
    ifstream i;
    i.open("test0",ios::out);
    i>>name;
    i>>roll;
    i>>fee;
    cout<<"Name:"<<name<<endl;
    cout<<"Roll No."<<roll<<endl;
    cout<<"Fee:"<<fee<<endl;
    getch();
}
```

Output

```
Name:Ajay
Roll No.24
Fee:55988.449219
```

❑ ❑ ❑ ❑ ❑