VTU Question Paper- Even Semester 2018
Object Oriented Programming Using C++

CBCS Scheme

16/17MCA22

USN 1 C R 1 7 M C A 0 6

Second Semester MCA Degree Examination, June/July 2018

Object Oriented Programming using C++

Time: 3 hrs. Max. Marks: 80

Note: *Answer FIVE full questions, choosing one full question from each module.*

Module-1

1 a. What is the difference between procedure oriented and object oriented programming? (08 Marks)

b. Explain the salient features of object oriented programming languages. (08 Marks)

OR

2 a. What is constructor? Explain two different types of constructors with programe example. (08 Marks)

b. Describe the following:
i) Scope resolution operator
ii) Inline function. (08 Marks)

Module-2

3 a. What are reference? Explain the three ways of using the reference with example. (08 Marks)

b. Describe the following:
i) Array of object
ii) this pointer. (08 Marks)

OR

4 a. Explain the concept of passing reference to objects function, with an example. (08 Marks)

b. Write a program to calculate the volume of difference geometric shapes like cube, cylinder and sphere and hence implement the concept of function overloading. (08 Marks)

Module-3

5 a. Describe the operator prefix ++ and postfix ++ operator with an example programe. (08 Marks)

b. Create a class called MATRIX using two-dimensional array of integers, implement the following operations by overloading the operator = = which checks the compatibility of two matrix to be added and subtracted. Perform the addition and subtraction by overloading the + and – operators respectively. Display the results by overloading the operator << if ($m_1 == m_2$) then $m_3 = m_1 + m_2$ and $m_4 = m_1 - m_2$ else display error. (08 Marks)

OR

6 a. What is inheritance? Explain any two types of inheritance with program example. (08 Marks)

b. Write a C++ perform which demonstrate how parameters are passed to base-class constructor. (04 Marks)

c. Write a C++ program simple inheritance concept by a base class FATHER with data members FirstName, SurName, DOB and BankBalance and creating 3 derived class SON, which inherits SurName and BankBalance feature from base class but provides its own feature FirstName and DOB, create and initialize F_1 and S_1 objects with appropriate constructor and display the father and son details. (04 Marks)

Module-4

7 a. What are virtual functions? With example demonstrate the use of virtual functions. (08 Marks)

b. Write a C++ program to create a template function for bubble sort and demonstrate sorting of integers and doubles. (08 Marks)

OR

8 a. What is abstract class? With example demonstrate the use of abstract class with an example. (08 Marks)

b. What is exception? How exceptions are handled in C++ with program example. (08 Marks)

Module-5

9 a. Discuss the four built-in streams that are automatically opened when a C++ program begins execution. (08 Marks)

b. Define the following:
   i) seekg( )
   ii) seekp( )
   iii) tellg( )
   iv) tellp( )
   v) precision( )
   vi) width( )
   vii) fill( )
   viii) self( ). (08 Marks)

OR

10 a. Write a program to implement FILE I/O operations on characters. I/O operations includes inputting a string, calculating length of string in a life, fetching the stored characters from it, etc. (08 Marks)

b. What is STL? Describe vector with program example. (08 Marks)

* * * * *

1a.	Explain the difference between procedure oriented and object oriented programming language.	08 Marks

Ans:

Difference between OOP and POP

Definition

OOP stands for Object-oriented programming and is a programming approach that focuses on data rather than the algorithm, whereas POP, short for Procedure-oriented programming, focuses on procedural abstractions.

Programs

In OOP, the program is divided into small chunks called objects which are instances of classes, whereas in POP, the main program is divided into small parts based on the functions.

Accessing Mode

Three accessing modes are used in OOP to access attributes or functions – ‘Private’, ‘Public’, and ‘Protected’. In POP, on the other hand, no such accessing mode is required to access attributes or functions of a particular program.

Focus

The main focus is on the data associated with the program in case of OOP while POP relies on functions or algorithms of the program.

Execution

In OOP, various functions can work simultaneously while POP follows a systematic step-by-step approach to execute methods and functions.

Data Control

In OOP, the data and functions of an object act like a single entity so accessibility is limited to the member functions of the same class. In POP, on the other hand, data can move freely because each function contains different data.

Security

OOP is more secure than POP, thanks to the data hiding feature which limits the access of data to the member function of the same class, while there is no such way of data hiding in POP, thus making it less secure.

Ease of Modification

New data objects can be created easily from existing objects making object-oriented programs easy to modify, while there’s no simple process to add data in POP, at least not without revising the whole program.

Process

	OOP follows a bottom-up approach for designing a program, while POP takes a top-down approach to design a program. **Examples** Commonly used OOP languages are C++, Java, VB.NET, etc. Pascal and Fortran are used by POP.	
1b.	Explain the Salient features of object oriented programming.	08Marks
Ans:	Object oriented programming can be defined 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". **Concepts of Oops:** **Class:**. Class is user defined data type and behaves like the built-in data type of a programming language. Class is a blue print/model for creating objects. **Object:** Object is the basic run time entities in an object oriented system. Object is the basic unit that is associated with the data and methods of a class.Object is an instance of a particular class. **Data Abstraction:** Abstraction refers to the act of representing essential features without including the background details. In programming languages, data abstraction will be specified by abstract data types and can be achieved through classes. **Encapsulation:** The wrapping up of data and functions into a single unit is known as encapsulation. It keeps them safe from external interface and misuse as the functions that are wrapped in class can access it. The insulation of the data from direct access by the program is called data hiding. **Inheritance:** It provides the concept of reusability. It is a mechanism of creating new classes from the existing classes. It supports the concept of hierarchical classification. A class which provides the properties is called Parent/Super/Base class. A class which acquires the properties is called Child/Sub/Derived class. A sub class defines only those features that are unique to it. **Polymorphism:** Polymorphism is derived from two greek words Poly and Morphs where poly means many and morphis means forms. Polymorphism means one thing existing in many forms. Polymorphism plays an important role in allowing objects having different internal structures to share the same external interfaces. Function overloading and operator overloading can be used to achieve polymorphism.	
2a.	What is constructor? Explain two different types of constructor with program example.	08 Marks

Ans.	A *constructor* is a special function that is a member of a class and has the same name as that class. For example, here is how the **stack** class looks when converted to use a constructor for initialization:
	// This creates the class stack.
	class stack {
	int stck[SIZE];	

```
int tos;
public:
stack(); // constructor
void push(int i);
int pop();
};
```
The constructor **stack( )** has no return type specified. In C++, constructors cannot return values and, thus, have no return type.

The **stack( )** constructor is coded like this:
```
// stack's constructor
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
}
```

Types of Constructors

1. **Default Constructors:** Default constructor is the constructor which doesn’t take any argument. It has no parameters.

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct
{
public:
    int a, b;

        // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
        // Default constructor called automatically
        // when the object is created
    construct c;
    cout << "a: "<< c.a << endl << "b: "<< c.b;
    return 1;
}
```
Output:

```
a: 10
b: 20
```

Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

```
// CPP program to illustrate
// parameterized constructors
#include<iostream>
using namespace std;

class Point
{
   private:
      int x, y;
   public:
      // Parameterized Constructor
      Point(int x1, int y1)
      {
         x = x1;
         y = y1;
      }

      int getX()
      {
         return x;
      }
      int getY()
      {
         return y;
      }
   };

int main()
{
   // Constructor called
   Point p1(10, 15);

   // Access values assigned by constructor
   cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

   return 0;
}
```

```
p1.x = 10, p1.y = 15
```

2b.	Describe the following: (i) Scope Resolution Operator	08 Mark

(ii) Inline Function s

Ans: (i) Scope Resolution Operator: As you know, the **::** operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name. For example, consider this fragment:

```
int i; // global i
void f()
{
int i; // local i
i = 10; // uses local i
.
.
.
}
```

As the comment suggests, the assignment **i = 10** refers to the local **i**. But what if function **f( )** needs to access the global version of **i**? It may do so by preceding the **i** with the **::** operator, as shown here.

```
int i; // global i
void f()
{
int i; // local i
::i = 10; // now refers to global i
.
.
.
}
```

(ii) Inline Function: In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called,

precede its definition with the **inline** keyword. For example, in this program, the function **max( )** is expanded in line instead of called:

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;
int main()
{
```

```
cout << (10>20 ? 10 : 20);
cout << " " << (99>88 ? 99 : 88);
return 0;
}
```

3a. What are reference? Explain the three ways of using the reference with example. 08 Marks

Ans: A reference is essentially an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference.

(i) As a function parameter:Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing.

Call-by-reference passes the address of the argument to the function.
Example:

```
// Use a reference parameter.
#include <iostream>
using namespace std;
void neg(int &i); // i now a reference
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(x); // no longer need the & operator
cout << x << "\n";
return 0;
}
void neg(int &i)
{
i = -i; // i is now a reference, don't need *
}
```

(ii) As a function return value: A function may return a reference. This has the rather startling effect of allowing a

function to be used on the left side of an assignment statement! For example, consider
this simple program:

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

Independent References: you can declare
a reference that is simply a variable. This type of reference is called an *independent reference*. When you create an independent reference, all you are creating is another name for an object. All independent references must be initialized when they are created.
The following program illustrates an independent reference:

```
#include <iostream>
using namespace std;
int main()
{
int a;
int &ref = a; // independent reference
a = 10;
cout << a << " " << ref << "\n";
ref = 100;
cout << a << " " << ref << "\n";
int b = 19;
ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref--; // this decrements a
// it does not affect what ref refers to
cout << a << " " << ref << "\n";
return 0;
}
```

The program displays this output:

```
10 10
100 100
19 19
18 18
```

3b.	Describe the following: (i) Arrays of objects (ii) this pointer	05 Marks

Ans:

(i) Arrays of objects: object array is exactly the same as it is for any other type of array. For example, this

program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
```

```
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

(ii) this pointer: When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. To understand **this**, first consider a program that creates a class called **pwr** that computes the result of a number raised to some power:

```
#include <iostream>
using namespace std;
class pwr {
double b;
int e;
double val;
public:
pwr(double base, int exp);
double get_pwr() { return val; }
};
pwr::pwr(double base, int exp)
{
b = base;
e = exp;
val = 1;
if(exp==0) return;
for( ; exp>0; exp--) val = val * b;
}
int main()
{
pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
cout << x.get_pwr() << " ";
cout << y.get_pwr() << " ";
cout << z.get_pwr() << "\n";
return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr( )**, the statement b = base; means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

```
this->b = base;
```

The **this** pointer points to the object that invoked **pwr( )**. Thus, **this –>b** refers to that object's copy of **b**. For example, if **pwr( )** had been invoked by **x** (as in **x(4.0, 2)**), then **this** in the preceding statement would have been pointing to **x**. Writing the statement without using **this** is really just shorthand. Here is the entire **pwr( )** constructor written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
this->b = base;
this->e = exp;
this->val = 1;
if(exp==0) return;
for( ; exp>0; exp--)
this->val = this->val * this->b;
}
```

4 a. Explain the concept of passing references to object function with an example. 08 Marks

Ans: When an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. However, when you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called. For example, try this program:

```
#include <iostream>
using namespace std;
class cl {
int id;
public:
int i;
cl(int i);
~cl();
void neg(cl &o) { o.i = -o.i; } // no temporary created
};
cl::cl(int num)
{
cout << "Constructing " << num << "\n";
id = num;
}
cl::~cl()
{
cout << "Destructing " << id << "\n";
} int main()
{
cl o(1);
o.i = 10;
o.neg(o);
cout << o.i << "\n";
return 0;
}
```

Here is the output of this program:

```
Constructing 1
-10
Destructing 1
```

4b. Write a program to calculate the volume of different shapes like cube, cylinder and sphere and hence implement the concept of function overloading. 08 Marks

Ans:

```
#include<iostream>
```

```
using namespace std;

int volume(int n)
{
        return n*n*n;
}
double volume(double r, double h)
{
        return 3.14*r*r*h;
}
double volume(double ra)
{
        return 1.33*3.14*ra*ra*ra;
}
int main()
{
        int s;
        double r,h,ra;
        cout<<" Enter the value of side in Integer to calculate the volume of cube:\n";
        cin>>s;
        cout<<" Volume of Cube is :"<<volume(s)<<endl;
        cout<<" Enter the value of radius and height in Double to calculate the volume of cylinder:\n";
        cin>>r>>h;
        cout<<" Volume of Cylinder is :"<<volume(r,h)<<endl;
        cout<<" Enter the value of radius in Double to calculate the volume of Sphere:\n";
        cin>>ra;
        cout<<" Volume of Sphere is :"<<volume(ra)<<endl;
        return 0;
}
```

5a.	Describe the operator prefix++ and postfix++ operator with an example.	08 Marks

Ans:	Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the **operator++( )** function. One is defined as shown in the foregoing program. The other is declared like this:
	loc operator++(int x);
	If the **++** precedes its operand, the **operator++( )** function is called. If the **++** follows its operand, the **operator++(int x)** is called and **x** has the value zero.
	The preceding example can be generalized. Here are the general forms for the prefix and postfix **++** and **– –** operator functions.
	// Prefix increment
	type operator++( ) {
	  // body of prefix operator
	// Postfix increment
	type operator++(int *x*) {
	// body of postfix operator	

```
}
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
// Postfix decrement
type operator– –(int x) {
// body of postfix operator
}
```

Using a Friend to Overload ++ or – –

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. This is because friend functions do not have **this** pointers. For example, this program uses friend functions to overload the prefix versions of **++** and **– –** operators relative to the **loc** class:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{ op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
```

```
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```

If you want to overload the postfix versions of the increment and decrement operators using a friend, simply specify a second, dummy integer parameter. For example, this shows the prototype for the **friend,** postfix version of the increment operator relative to **loc.**

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

5b.	Create a class called MATRIX using two-dimensional array of integers. Implement the following operations by overloading the operator = = which checks the compatibility of two matrices to be subtracted. Overload the operator '-'for matrix subtraction as m3 = m1-m2 when (m1= =m2).	08 Marks

Ans:

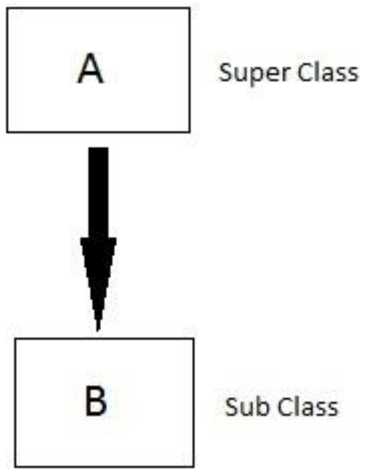
```
#include<iostream>
#define Max 20
using namespace std;
class Matrix
{
        public:
        int a[Max][Max];
        int r,c;
        void getorder();
        void getdata();
        Matrix operator -(Matrix);
        friend ostream& operator <<(ostream &, Matrix);
        int operator==(Matrix);

};
void Matrix::getorder()
{
        cout<<"enter the number of rows\n";
        cin>>r;
        cout<<"enter the number of columns\n";
        cin>>c;
}
void Matrix::getdata()
{
        int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
```

```
                    {
                              cin>>a[i][j];
                    }
          }
}

Matrix Matrix::operator -(Matrix m2)
{
          Matrix m4;
          int i,j;
          for(i=0;i<r;i++)
          {
                    for(j=0;j<c;j++)
                    {
                              m4.a[i][j] = a[i][j] - m2.a[i][j];
                    }
          }
          m4.r = r;
          m4.c = c;
          return m4;
}
ostream & operator <<(ostream & out, Matrix m)
{
          int i,j;
          for(i=0;i<m.r;i++)
          {
                    for(j=0;j<m.c;j++)
                    {
                    out<<m.a[i][j]<<"\t";
                    }
                    out<<endl;
          }
          return out;
}
int Matrix::operator==(Matrix m2)
{
          if((r==m2.r) && (c==m2.c))
                    return 1;
          else
                    return 0;

}
int main()

{
          Matrix m1,m2,m4;
          cout<<"enter the order of the first matrix\n";
          m1.getorder();
          cout<<"enter the order of the second matrix\n";

          m2.getorder();
```

```
            if(m1 == m2)

            {
                        cout<<"enter the elements of the first matrix\n";
                        m1.getdata();
                        cout<<"enter the elements of the second matrix\n";
                        m2.getdata();
                        m4 = m1 - m2;
                        cout<<"Difference of matrices is \n";
                        cout<<m4<<endl;

            } else {

                        cout<<"Order of the matrices is not same";
            }
            return 0;
}
```

6a.	What is inheritance? Explain any two types of inheritance with program example.	08 Marks

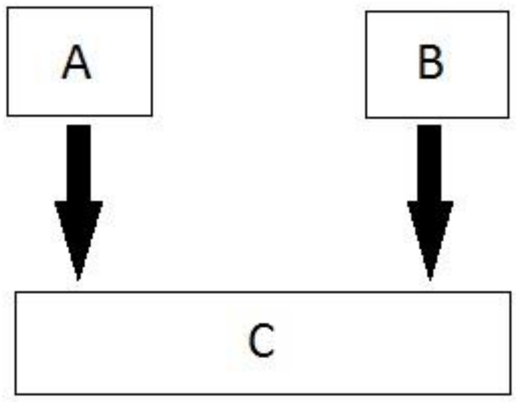
Ans: Inheritance allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.
A class that is inherited is referred to as a *base class*. The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.
Types of Inheritance:
Single Inheritance:
In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



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

```
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }

void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.



It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
        void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

6b.	Write a C++ program which demonstrates how parameters are passed to base class constructor.	04 Marks

Ans: In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax, However,to pass arguments to a constructor in a base class, use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// *body of derived constructor*
}

Example:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
```

```
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

6c.	Write a C++ Program Simple Inheritance concept by creating a base class FATHER with data members FirstName, SurName, DOB and BankBalance and creating a derived class SON, which inherits SurName and BankBalance feature from base class but provides its own feature FirstName and DOB. Create and initialize F1 and S1 objects with appropriate constructors and display the Father & Son details.	04 Marks

Ans:

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

class Father
{
        public:
        string Fname;
        string Surname;
        string DOB;
        double BankBalance;
        Father(string fn, string sn, string dob, double bb)
        {
                Fname=fn;
                Surname=sn;
                DOB=dob;
                BankBalance=bb;
         }
        void Display()
        {
                cout<<"Father's Fname:"<<Fname<<endl;
                cout<<"Father's Surname:"<<Surname<<endl;
                cout<<"Father's DOB:"<<DOB<<endl;
                cout<<"Father's BankBalance:"<<BankBalance<<endl;

        }
};
 class Son : public Father
{
        public: string SFname,SDOB;
        Son( string fn1, string dob1,string fn,string sn, string dob, double bb ):Father(fn,sn,dob,bb)
        {
                SFname=fn1;
                SDOB=dob1;
        }
        void Display()
        {
                Father::Display();
                cout<<"Son's Fname:"<<SFname<<endl;
                cout<<"Son's Surname:"<<Surname<<endl;
                cout<<"Son's DOB:"<<SDOB<<endl;
                cout<<"Son's BankBalance:"<<BankBalance<<endl;

        }
};
int main()
{
        Father F1("Ajit", "Singh","2-1-1944",200000);
        Son S1("Manoj","2-2-2004","Amit", "sharma","1-1-1944",20000);
        F1.Display();
        S1.Display();
}
```

7a.	What are virtual functions? With example demonstrate the use of virtual function.	08 Marks

Ans: A*virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

```
// Virtual function practical example.
#include <iostream>
using namespace std;
class convert {
protected:
double val1; // initial value
double val2; // converted value
public:
convert(double i) {
val1 = i;
}
double getconv() { return val2; }
double getinit() { return val1; }
virtual void compute() = 0;
};
// Liters to gallons.
class l_to_g : public convert {
public:
```

```
l_to_g(double i) : convert(i) { }
void compute() {
val2 = val1 / 3.7854;
}
};
// Fahrenheit to Celsius
class f_to_c : public convert {
public:
f_to_c(double i) : convert(i) { }
void compute() {
val2 = (val1-32) / 1.8;
}
};
int main()
{
convert *p; // pointer to base class
l_to_g lgob(4);
f_to_c fcob(70);
// use virtual function mechanism to convert
p = &lgob;
cout << p->getinit() << " liters is ";
p->compute();
cout << p->getconv() << " gallons\n"; // l_to_g
p = &fcob;
cout << p->getinit() << " in Fahrenheit is ";
p->compute();
cout << p->getconv() << " Celsius\n"; // f_to_c
return 0;
}
```

7b.	Write a C++ Program to create a template function for bubble sort and demonstrate sorting of integers and doubles.	08 Marks

Ans:

```
#include<iostream>

using namespace std;
#define Max 100

template <class T>
void sort(T a[],int n)
{
int i,j;
for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-i-1;j++)
                        {
                                if(a[j]>a[j+1])
                                {
                                        T temp=a[j];
                                        a[j]=a[j+1];
                                        a[j+1]=temp;
                                }
                        }
        }
}

int main()

{
        int a[Max],i,n;
        double d[Max];
        cout<<"enter array size\n\n";
        cin>>n;
        cout<<"enter array integer elements\n\n";
        for(i=0;i<n;i++)
                cin>>a[i];
                cout<<"enter array double elements\n\n";
        for(i=0;i<n;i++)
                cin>>d[i];
        cout<<"integer part\n\n";
        sort(a,n);
        for(i=0;i<n;i++)
                cout<< a[i]<<"\n";
        cout<<"double part\n\n";
        sort(d,n);
        for(i=0;i<n;i++)
                cout<<d[i]<<"\n";
        return 0;
}
```

8a.	What is abstract class? With example demonstrate the use of abstract class with an example.	08 Marks

Ans: A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and eferences to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

```
#include <iostream>
using namespace std;
class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
```

```
public:
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays 14 – hexadecimal
o.setval(20);
o.show(); // displays 24 - octal
return 0;
}
```

8b.	What is an exception? How exceptions are handled in C++ with program example.	08 Marks

Ans: Exception is run time error which stop the program execution. *Exception handling* allows you to manage run-time errors in an orderly fashion. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description. Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}
```

This program uses exception handling to manage a divide-by-zero error.

```
#include <iostream>
using namespace std;
void divide(double a, double b);
int main()
{
double i, j;
do {
cout << "Enter numerator (0 to stop): ";
cin >> i;
cout << "Enter denominator: ";
cin >> j;
divide(i, j);
} while(i != 0);
return 0;
}
void divide(double a, double b)
{
try {
if(!b) throw b; // check for divide-by-zero
cout << "Result: " << a/b << endl;
}
catch (double b) {
cout << "Can't divide by zero.\n";
}
}
```

9a.	Discuss the four built in streams that are automatically opened when a C++ program begins execution.	08 Marks

When a C++ program begins execution, four built-in streams are automatically opened.

They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

By default, the standard streams are used to communicate with the console.

However, in environments that support I/O redirection (such as DOS, Unix, OS/2,

and Windows), the standard streams can be redirected to other devices or files.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >>

```
#include <iostream>

using namespace std;

int main( )
{
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;

}
```

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as <<

Ex:

```
#include <iostream>

 using namespace std;

 int main( )

{

   char str[] = "Hello C++";

    cout << "Value of str is : " << str << endl;

}
```

The standard error stream (cerr):

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

```
#include <iostream>
```

```
using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

The standard log stream (clog):

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

Example of setting two format flag:

```
#include <iostream>
using namespace std;
int main()
{
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << 100.0; // displays +100.000
return 0;
}
```

9b.	Define the following: (i) Seekg() (ii) Seekp() (iii) tellg() (iv) tellp() (v) precision() (vi) width() (vii) fill()	08 Mark s

(viii) setf()

Ans. (i) Seekg() : The **seekg( )** function moves the associated file's current get pointer *offset* number

of characters from the specified *origin*.

Format:
istream &seekg(off_type *offset*, seekdir *origin*);

Origin must be one of these three values:
ios::beg Beginning-of-file
ios::cur Current location
ios::end End-of-file
Here, **off_type** is an integer type defined by **ios** that is capable of containing the largest
valid value that *offset* can have. **seekdir** is an enumeration defined by **ios** that determines
how the seek will take place.

(ii) Seekp():The **seekp( )** function moves the associated file's current put pointer *offset* number of characters from the specified *origin*, which must be one of the values

Origin must be one of these three values:
ios::beg Beginning-of-file
ios::cur Current location
ios::end End-of-file
Format:
ostream &seekp(off_type *offset*, seekdir *origin*);

Here, **off_type** is an integer type defined by **ios** that is capable of containing the largest
valid value that *offset* can have. seekdir is an enumeration defined by ios that determines
how the seek will take place.

(iii) Tellg():You can determine the current position of each file pointer by using these functions:

pos_type tellg( );
pos_type tellp( );
Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that
either function can return. You can use the values returned by **tellg( )** and **tellp( )** as
arguments to the following forms of **seekg( )** and **seekp( )**, respectively.
istream &seekg(pos_type *pos)*;
These function allow you to save the current file location, perform other file
operations, and then reset the file location to its previously saved location.

(iv) Tellp():You can determine the current position of each file pointer by using these functions:

pos_type tellg( );
pos_type tellp( );

Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that
either function can return. You can use the values returned by **tellg( )** and **tellp( )** as
arguments to the following forms of **seekg( )** and **seekp( )**, respectively.

	ostream &seekp(pos_type *pos*);
	These function allow you to save the current file location, perform other file
	operations, and then reset the file location to its previously saved location.
	(v) Precision() : When outputting floating-point values, you can determine the number of digits
	of precision by using the **precision( )** function. Its prototype is shown here:
	streamsize precision(streamsize *p*);
	
	(vi) Width() : By default, when a value is output, it occupies only as much space as the number
	of characters it takes to display it. However, you can specify a minimum field width
	by using the **width( )** function. Its prototype is shown here:
	streamsize width(streamsize *w*);
	Here, *w* becomes the field width, and the previous field width is returned. In some
	implementations, the field width must be set before each output. If it isn't, the default
	field width is used. The **streamsize** type is defined as some form of integer by the
	compiler.
	After you set a minimum field width, when a value uses less than the specified
	width, the field will be padded with the current fill character (space, by default) to
	reach the field width. If the size of the value exceeds the minimum field width, the
	field will be overrun. No values are truncated.
	(vii) Fill():By default, when a field needs to be filled, it is filled with spaces. You can specify
	the fill character by using the **fill( )** function. Its prototype is
	char fill(char *ch*);
	After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.
	(viii) Setf():To set a flag, use the **setf( )** function. This function is a member of **ios**. Its most common
	form is shown here:
	fmtflags setf(fmtflags *flags*);
	This function returns the previous settings of the format flags and turns on those flags
	specified by *flags*. For example, to turn on the **showpos** flag, you can use this statement:
	stream.setf(ios::showpos);
	Here, *stream* is the stream you wish to affect. Notice the use of **ios::** to qualify **showpos**.
	Since **showpos** is an enumerated constant defined by the **ios** class, it must be qualified
	by **ios** when it is used.	

10a.	Write a program to implement FILE I/O operations on characters. I/O operations includes inputting a string, Calculating length of the string, Storing the string in a file, fetching the stored characters from it, etc.	08 Marks

Ans:	#include <iostream>
	#include <fstream>
	using namespace std;
	int main()
	{
	double fnum[4] = {99.75, -34.4, 1776.0, 200.1};
	int i;
	ofstream out("numbers", ios::out | ios::binary);
	if(!out) {	

```
cout << "Cannot open file.";
return 1;
}
out.write((char *) &fnum, sizeof fnum);
out.close();
for(i=0; i<4; i++) // clear array
fnum[i] = 0.0;
ifstream in("numbers", ios::in | ios::binary);
in.read((char *) &fnum, sizeof fnum);
// see how many bytes have been read
cout << in.gcount() << " bytes read\n";
for(i=0; i<4; i++) // show values read from file
cout << fnum[i] << " ";
in.close();
return 0;
}
```

10b.	What is STL. Explain vector with program example.	08 Marks

Ans: At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines *associative containers*, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key. Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a *range* of elements within a container.

Iterators

Iterators are objects that act, more or less, like pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators:

Iterator	Access	Allowed
Random Access	Store and retrieve values.	Elements may be accessed randomly.
Bidirectional	Store and retrieve values.	Forward and backward moving.
Forward	Store and retrieve values.	Forward moving only.
Input	Retrieve, but not store values.	Forward moving only.
Output	Store, but not retrieve values.	Forward moving only.

Vectors

Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the

array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements. The template specification for **vector** is shown here:
template <class T, class Allocator = allocator<T> > class vector

Some of the most commonly used member functions are **size( )**, **begin( )**, **end( )**, **push_back( )**, **insert( )**, and **erase( )**. The **size( )** function returns the current size of the vector. This function is quite useful because it allows you to determine the size of a vector at run time. Remember, vectors will increase in size as needed, so the size of a vector must be determined during execution, not during compilation. The **begin( )** function returns an iterator to the start of the vector. The **end( )** function returns an iterator to the end of the vector. As explained, iterators are similar to pointers, and it is through the use of the **begin( )** and **end( )** functions that you obtain an iterator to the beginning and end of a vector. The **push_back( )** function puts a value onto the end of the vector. If necessary, the vector is increased in length to accommodate the new element. You can also add elements to the middle using **insert( )**. A vector can also be initialized. In any event, once a vector contains elements, you can use array subscripting to access or modify those elements. You can remove elements from a vector using **erase( )**