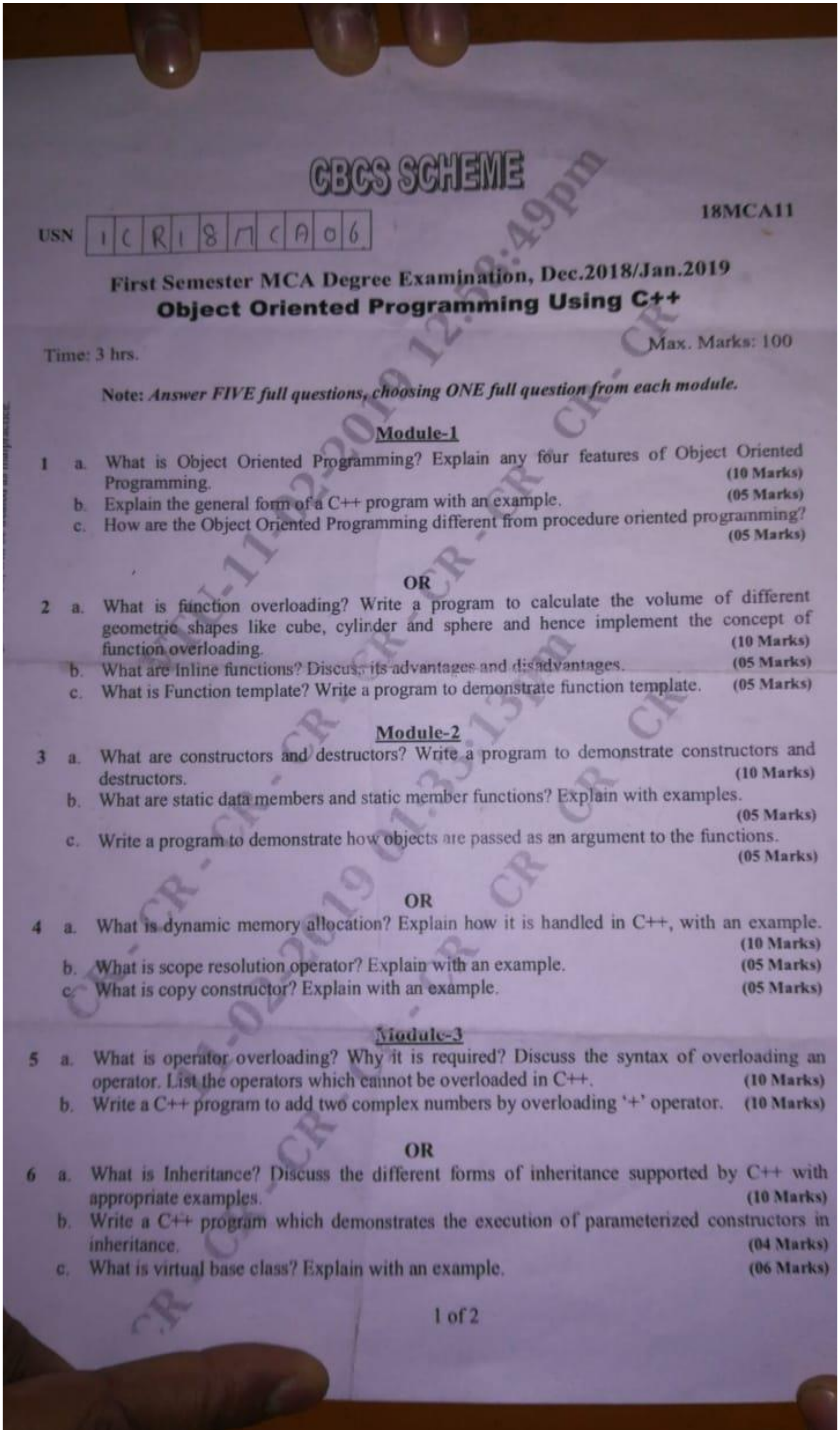
VTU Question Paper- Odd Semester 2018

Object Oriented Programming Using C++



CBCS SCHEME

USN 1CR18MCA06

18MCA11

First Semester MCA Degree Examination, Dec.2018/Jan.2019

Object Oriented Programming Using C++

Time: 3 hrs. Max. Marks: 100

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

Module-1

1 a. What is Object Oriented Programming? Explain any four features of Object Oriented Programming. **(10 Marks)**

b. Explain the general form of a C++ program with an example. **(05 Marks)**

c. How are the Object Oriented Programming different from procedure oriented programming? **(05 Marks)**

OR

2 a. What is function overloading? Write a program to calculate the volume of different geometric shapes like cube, cylinder and sphere and hence implement the concept of function overloading. **(10 Marks)**

b. What are Inline functions? Discuss its advantages and disadvantages. **(05 Marks)**

c. What is Function template? Write a program to demonstrate function template. **(05 Marks)**

Module-2

3 a. What are constructors and destructors? Write a program to demonstrate constructors and destructors. **(10 Marks)**

b. What are static data members and static member functions? Explain with examples. **(05 Marks)**

c. Write a program to demonstrate how objects are passed as an argument to the functions. **(05 Marks)**

OR

4 a. What is dynamic memory allocation? Explain how it is handled in C++, with an example. **(10 Marks)**

b. What is scope resolution operator? Explain with an example. **(05 Marks)**

c. What is copy constructor? Explain with an example. **(05 Marks)**

Module-3

5 a. What is operator overloading? Why it is required? Discuss the syntax of overloading an operator. List the operators which cannot be overloaded in C++. **(10 Marks)**

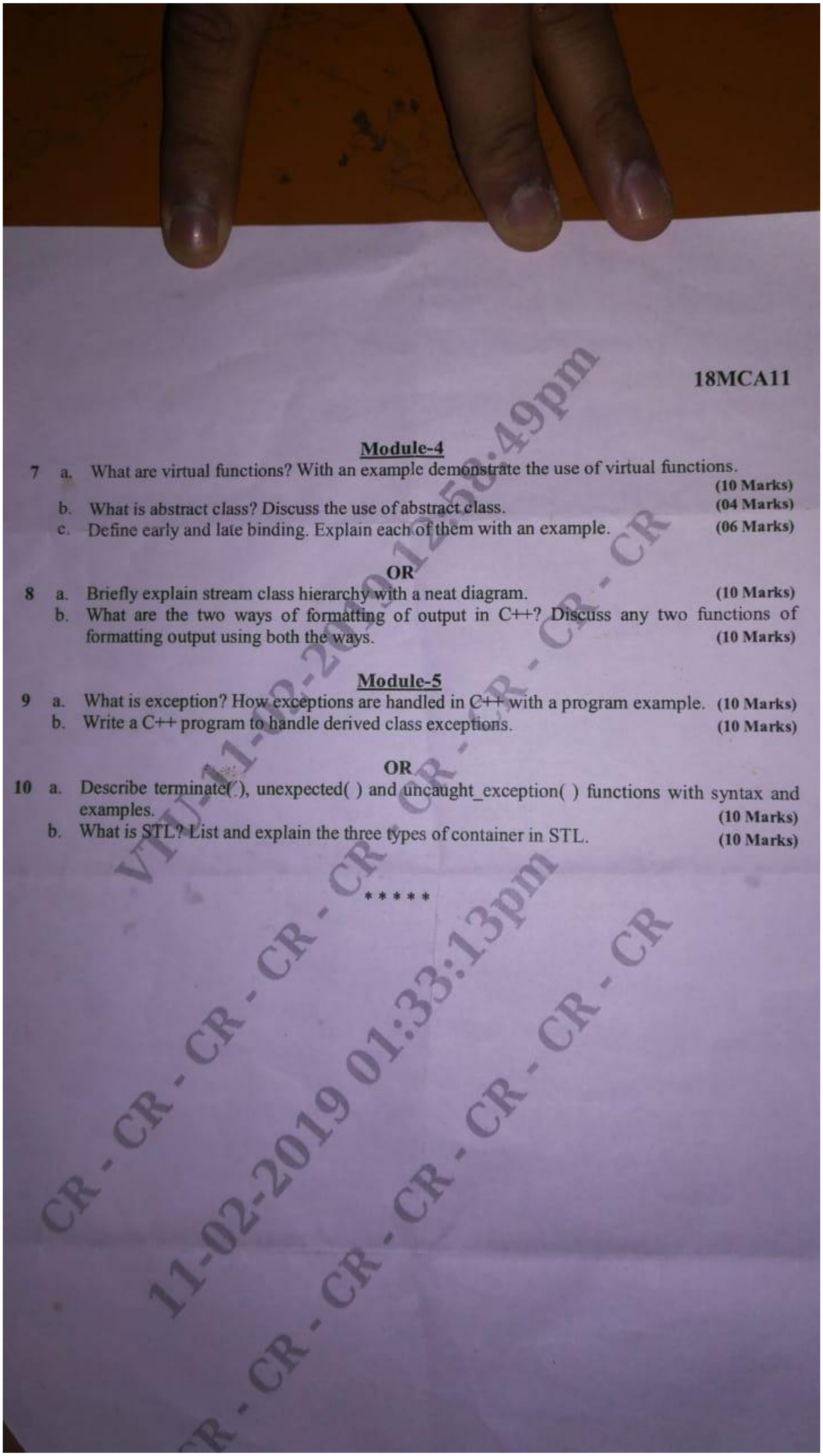
b. Write a C++ program to add two complex numbers by overloading '+' operator. **(10 Marks)**

OR

6 a. What is Inheritance? Discuss the different forms of inheritance supported by C++ with appropriate examples. **(10 Marks)**

b. Write a C++ program which demonstrates the execution of parameterized constructors in inheritance. **(04 Marks)**

c. What is virtual base class? Explain with an example. **(06 Marks)**



Module-4

7 a. What are virtual functions? With an example demonstrate the use of virtual functions. **(10 Marks)**

b. What is abstract class? Discuss the use of abstract class. **(04 Marks)**

c. Define early and late binding. Explain each of them with an example. **(06 Marks)**

OR

8 a. Briefly explain stream class hierarchy with a neat diagram. **(10 Marks)**

b. What are the two ways of formatting of output in C++? Discuss any two functions of formatting output using both the ways. **(10 Marks)**

Module-5

9 a. What is exception? How exceptions are handled in C++ with a program example. **(10 Marks)**

b. Write a C++ program to handle derived class exceptions. **(10 Marks)**

OR

10 a. Describe terminate( ), unexpected( ) and uncaught_exception( ) functions with syntax and examples. **(10 Marks)**

b. What is STL? List and explain the three types of container in STL. **(10 Marks)**

* * * * *

1a. What is Object Oriented Programming? Explain any four features of Object Oriented Programming. 10 Marks

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.

1b. Explain the general form of a C++ program with an example. 05Marks

Ans: Short sample C++ program shown here.

```
#include <iostream>
using namespace std;
int main()
{
int i;
cout << "This is output.\n"; // this is a single line comment
/* you can still use C style comments */
// input a number using >>
cout << "Enter a number: ";
cin >> i;
// now, output a number using <<
cout << i << " squared is " << i*i << "\n";
return 0;
}
```

As you can see, this program looks much different from the C subset programs found in Part One. A line-by-line commentary will be useful. To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>** is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that **<iostream>** is one of the modern-style headers defined by Standard C++. Modern C++ headers do not use the **.h** extension.
The next line in the program is

```
using namespace std;
```

This tells the compiler to use the **std** namespace.

```
int main()
```

Notice that the parameter list in **main( )** is empty. In C++, this indicates that **main( )** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here: int main(void)
This was the way **main( )** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.
The next line contains two C++ features.

```
cout << "This is output.\n"; // this is a single line comment
```

First, the statement

```
cout << "This is output.\n";
```

Next, the program prompts the user for a number. The number is read from the keyboard with this statement:

```
cin >> i;
```

The program ends with this statement:

```
return 0;
```

This causes zero to be returned to the calling process (which is usually the operating system). This works the same in C++ as it does in C. Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value. You may also use the values **EXIT_SUCCESS** and **EXIT_ FAILURE** if you like.

1c. How are the Object Oriented Programming different from procedure Oriented Programming? 05 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.

2a.	What is function overloading? Write a program to calculate the volume of different shapes like cube, cylinder and sphere and hence implement the concept of function overloading.	10 Marks
Ans:	A function with same name and multiple definitions is called function overloading. Function overloading is usually used to enhance the readability of the program. If we have to perform one single operation but with different number or types of arguments, then we can simply overload the function. There are two ways to overload a function *By change in number of arguments	

* By change in type of arguments

***By change in number of arguments:**

In this type of function overloading we define two functions with same names but different number of parameters of the same type

Ex:

```
int sum (int x, int y)
{
cout << x+y;
}
int sum(int x, int y, int z)
{
cout << x+y+z;
}
int main()
{
sum (10,20);  // sum() with 2 parameter will be called
sum(10,20,30);  //sum() with 3 parameter will be called
}
```

*** By change in type of arguments**

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different.

Ex:

```
int sum(int x,int y)

{

cout<< x+y;

}

double sum(double x,double y)

{

cout << x+y;

}

int main()

{

sum (10,20);

sum(10.5,20.5);

}
```

```
#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;
}
```

2b. What are inline functions? Discuss its advantages and disadvantages. 5 Marks

Ans: 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;
}
```

Inline functions provide following advantages:

1) Function call overhead doesn’t occur.
2) It also saves the overhead of push/pop variables on the stack when function is called.
3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.
2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

2c. What is function template? Write a program to demonstrate function template. 5 Marks

Ans: Ageneric function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data.

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

```
#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;
}
```

3a.	What are constructors and destructors? Write a program to demonstrate constructor and destructor.	10Marks

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
```

Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. It should be declared in public section of a class. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Ex:

```
#include<iostream>
Using namespace std;
Class Test
{
          int *a;
public:
         Test(int size)
         {
               a=new int[size];
               cout<<"Constructor created";
           }
           ~Test()
           {
                  delete  a;
                  cout<<"Destructor created";
             }
};
int main()
{
     int s;
     cout<<"Enter the size of an array";
     cin>>s;
     Test t(s);
     return 0;
}
```

3b.	What are static data members and static member functions? Explain with examples.	05 Marks
Ans:	Static Data Members When you precede a member variable's declaration with **static**, then only one copy of that variable will exist and that all objects of the class will share that variable. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created. When you declare a **static** data member within a class, you are *not* defining it. We must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. Example	

```
#include <iostream>
using namespace std;
class shared {
static int a;
int b;
public:
void set(int i, int j) {a=i; b=j;}
void show();
} ;
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}
int main()
{
shared x, y;
x.set(1, 1); // set a to 1
x.show();
y.set(2, 2); // change a to 2
y.show();
x.show(); /* Here, a has been changed for both x and y
because a is shared by both objects. */
return 0;
}
```

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Static Member Functions

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member functions.)

A **static** member function does not have a **this** pointer. There cannot be a **static** and a non-**static** version of the same function. A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile**.

```
#include <iostream>
using namespace std;
class cl {
static int resource;
public:
static int get_resource();
void free_resource() { resource = 0; }
};
int cl::resource; // define resource
int cl::get_resource()
{
```

```
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
}
}
int main()
{
cl ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
if(cl::get_resource()) cout << "ob1 has resource\n";
if(!cl::get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource();
if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
return 0;
}
```

3c.	Write a program to demonstrate how objects are passed as an argument to the function.	05 Marks

Ans:

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass::myclass(int n)
{
i = n;
cout << "Constructing " << i << "\n";
}
myclass::~myclass()
{
cout << "Destroying " << i << "\n";
}
void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2);
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
```

This program produces this output:

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

4a. What is dynamic memory allocation? Explain how it is handled in C++, with an example. 10 Marks

Ans: Dynamic memory allocation operators: new and delete.
new: To allocate the memory.
Syntax:

```
ptr_var = new vartype;
e.g: ptr = new int;
ptr_var = new vartype(initial_value);
```

The type of initial value should be same as the vartype;

e.g: ptr = new int(100);
delete: To free the memory.
delete ptr_var;

If there is insufficient memory then the exception bad_alloc will be raised. This exception is defined in the header <new>. It is available in standard C++.

Advantages of new and delete:

new and delete operators are like malloc() and free() in C Language. But they have more advantages.

1. new automatically allocates enough memory to hold the object.
   (No need to use sizeof operator).
2. new automatically returns the pointer to the specified type.
   It is not needed to typecast it explicitly.

Allocating Arrays:

ptrvar = new arrtype[size];
Delete [ ] ptrvar;

*** The initial values can’t be given during the array allocation.

Example:

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int; // allocate space for an int
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
*p = 100;
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```

4b. What is Scope Resolution Operator? Explain with an example. 05 Marks

Ans: 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
.
.
.
}
```

4c. What is copy constructor? Explain with an example. 05 Marks

Ans: The parameters of a constructor can be of any of the data types except an object of its own class as a value parameter. Hence declaration of the following class specification leads to an error:

```
        class x
        {
                private:
                        ………
                public:
                        x( x obj);
                        ……..
        };
```

A class’s own object can be passed as a reference parameter.

Ex:

```
            class X
            {       ……..
                    public:
                    X()
                    X( X &obj);
                    X(int a);
            };
        is valid
```

Such a constructor having a reference to an instance of its own class as an argument is known as copy constructor.

Ex.

```
            bag b3=b2;   // copy constructor invoked
            bag b3(b2);   // copy constructor invoked
            b3=b2;        // copy constructor is not invoked.
```

A copy constructor copies the data members from one object to another.

5a. What is operator overloading? Why it is required? Discuss the syntax of overloading an operator. List the operator which cannot be overloaded in C++. 10 Marks

Ans: The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating operator functions. An *operator function* defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions.

Creating a Member Operator Function

A member operator function takes this general form:

*ret-type class-name::*operator*#(arg-list)*
{
// operations
}

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The **#** is a placeholder. When you create an operator function, substitute the operator for the **#**. For example, if you are overloading the **/** operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty.

When you are overloading binary operators, *arg-list* will contain one parameter. Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the **+** operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+( )**:

```
 #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);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

Operator Overloading Restrictions

There are some restrictions that apply to operator overloading. You cannot alter the precedence of an operator. You cannot change the number of operands that an operator takes. (You can choose to ignore an operand, however.) Except for the function calloperator (described later), operator functions cannot have default arguments. Finally, these operators cannot be overloaded:

. : : .* ?

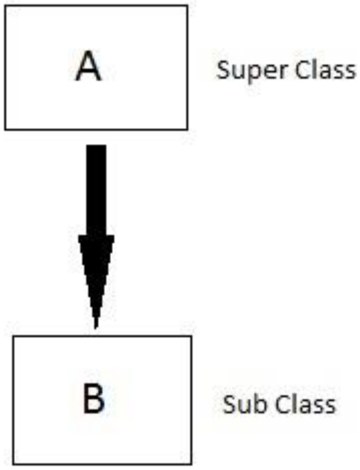
5b. Write a C++ Program to add two complex numbers by overloading '+' operator. 10 Marks

Ans:

```
#include<iostream>

using namespace std;

class complex

{
        int real;
        int imag;
public:
        void read()
        {
                cout<<"enter real and imaginary";
                cin>>real>>imag;
        }
        void display()
        {
                        cout<<real<<"+"<<imag<<"i"<<endl;
        }
 friend complex operator +(complex, complex);
        };

complex operator +(complex a1,complex a2)

        {
                complex temp1;
                temp1.real=a1.real+a2.real;
                temp1.imag=a1.imag+a2.imag;
                return temp1;
        }

int main()
```

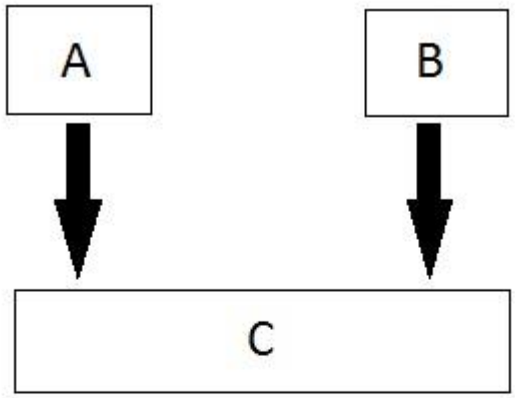
```
{
        int a;
        complex s1,s2,s3;
        s1.read();
        s2.read();
        cout<<"First Complex number";
        s1.display();
        cout<<"Second Complex number";
        s2.display();
        s3=s1+s2;
        cout<<"addition of 2 complex number\n"<<endl;
        s3.display();
        return 0;
}
```

6a.	What is inheritance? Discuss the different forms of inheritance supported by C++ with appropriate example.	10 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. Multilevel Inheritance. **Multiple Inheritance**. Heirarchical Inheritance. **Hybrid** Inheritance. Multipath 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.

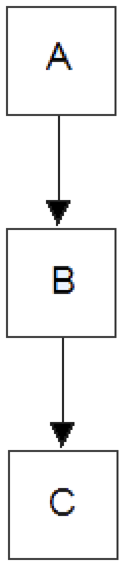


It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// 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;
}
```

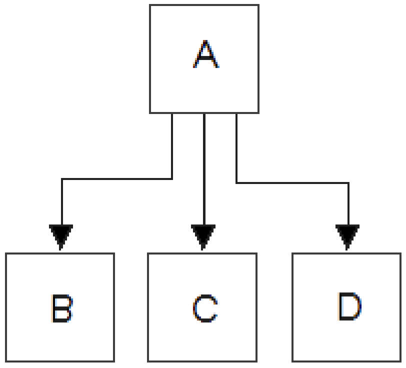
Multilevel Inheritance:

A derived class with one base class and that base class is a derived class of another is called **multilevel inheritance**.



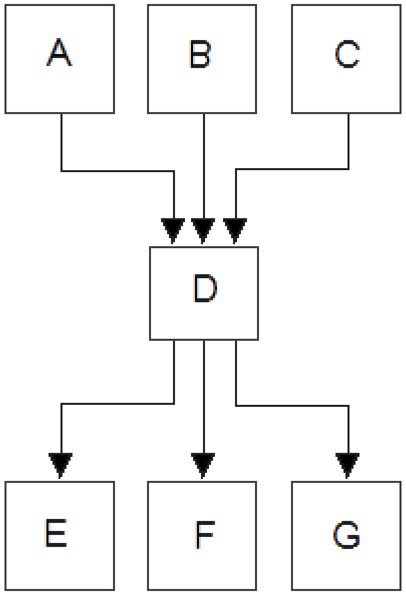
Hierarchical inheritance.

Multiple derived classes with same base class is called **hierarchical inheritance**.



Hybrid inheritance.

Combination of multiple and hierarchical inheritance is called **hybrid inheritance**.



6b. Write a C++ program, which demonstrates the execution of parameterized constructors in inheritance. 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. What is virtual base class? Explain with an example. 06 Marks

Ans: When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base**:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
    ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited **class**' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous. One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;
myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

7a. What are virtual functions? With example demonstrate the use of virtual function. 10 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. What is abstract class? Discuss the use of abstract class. 04 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;
}
```

7c. Define early binding and late binding. Explain each of them with an example. 06 Marks

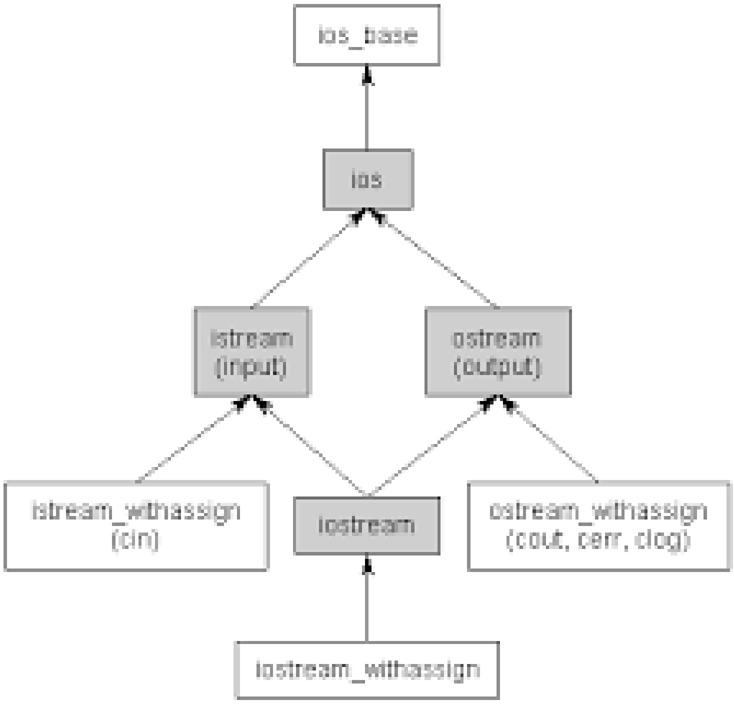
Ans: *Early binding* refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

8a. Briefly explain stream class hierarchy with a neat diagram. 10 Marks

Ans: A*stream* is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. As mentioned, Standard C++ provides support for its I/O system in **<iostream>**. In this header, a rather complicated set of class hierarchies is defined that supports I/O operations. The I/O classes begin with a system of template classes. A template class defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly. The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for **basic_ios** is called **ios_base**, which defines several nontemplate traits used by **basic_ios**.) **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively. As explained, the I/O library creates two specializations of the template class hierarchies just described: one for 8-bit characters and one for wide characters.



8b. What are the two ways of formatting of output in C++? Discuss any two functions of formatting output using both the ways.

10 Marks

Ans: The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. There are two related but conceptually different ways that you can format data. First, you can directly access members of the **ios** class. Specifically, you can set various format status flags defined inside the **ios** class or call various **ios** member functions. Second, you can use special functions called *manipulators* that can be included as part of an I/O expression.

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined. (Technically, these values are defined within **ios_base**, which, as explained earlier, is a base class for **ios**.)

adjustfield basefield boolalpha dec
fixed floatfield hex internal
left oct right scientific
showbase showpoint showpos skipws
unitbuf uppercase

These values are used to set or clear the format flags. If you are using an older compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer. When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded. When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.

By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.

Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.

By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.

When **unitbuf** is set, the buffer is flushed after each insertion operation.

When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.

Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

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);
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. Many of the I/O manipulators parallel member functions of the **ios** class. To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program. Here is an example that uses some manipulators:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << hex << 100 << endl;
cout << setfill('?') << setw(10) << 2343.0;
return 0;
}
```

This displays

```
64
??????2343
```

Notice how the manipulators occur within a larger I/O expression. Also notice that when a manipulator does not take an argument, such as **endl( )** in the example, it is not followed by parentheses. This is because it is the address of the function that is passed to the overloaded **<<** operator. As a comparison, here is a functionally equivalent version of the preceding program that uses **ios** member functions to achieve the same results:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout.setf(ios::hex, ios::basefield);
cout << 100 << "\n"; // 100 in hex
cout.fill('?');
cout.width(10);
cout << 2343.0;
return 0;
}
```

9a. What is an exception? How exceptions are handled in C++ with program example. 10 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";
}
}
```

9b. Write a C++ program to handle derived class exceptions. 10 Marks

Ans

We need to be careful how you order your **catch** statements when trying to catch exception types that involve base and derived classes because a **catch** clause for a base class will also match any class derived from that base. Thus, if we want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. If you don't do this, the base class **catch** will also catch all derived classes. For example, consider the following program.

```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main()
{
D derived;
try {
throw derived;
}
catch(B b) {
cout << "Caught a base class.\n";
}
catch(D d) {
cout << "This won't execute.\n";
}
return 0;
}
```

Here, because **derived** is an object that has **B** as a base class, it will be caught by the first **catch** clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the **catch** clauses.

```
#include <iostream>
using namespace std;
class B {
```

	};
	class D: public B {
	};
	int main()
	{
	D derived;
	try {
	throw derived;
	}
	catch(D d) {
	cout << "This won't execute.\n";
	}
	catch(B b) {
	cout << "Caught a base class.\n";
	}
	return 0;
	}	

10a.	Describe terminate(), unexpected() and uncaught_exception() functions with syntax and examples.	10 Marks

Ans:	As mentioned earlier, **terminate( )** and **unexpected( )** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:
	void terminate( );
	void unexpected( );
	These functions require the header **<exception>**. The **terminate( )** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown. The **terminate( )** function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an exception. In general, **terminate( )** is the handler of last resort when no other handlers for an exception are available. By default, **terminate( )** calls **abort( )**.
	
	// Set a new terminate handler.
	#include <iostream>
	#include <cstdlib>
	#include <exception>
	using namespace std;
	void my_Thandler() {
	cout << "Inside new terminate handler\n";
	abort();
	}
	int main()
	{
	// set a new terminate handler
	set_terminate(my_Thandler);
	try {
	cout << "Inside try block\n";
	throw 100; // throw an error
	}	

```
catch (double i) { // won't catch an int exception
// ...
}
return 0;
}
```

The output from this program is shown here.

```
Inside try block
Inside new terminate handler
abnormal program termination
```

The uncaught_exception( ) Function

The C++ exception handling subsystem supplies one other function that you may find useful: U**ncaught_exception( )**. Its prototype is shown here:

```
bool uncaught_exception( );
```

This function returns **true** if an exception has been thrown but not yet caught. Once caught, the function returns **false**.

10b. What is STL? List and explain the three types of containers in STL. 10 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.

Container Classes:

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( )**

List

The list class supports a bidirectional, linear list. Unlike a vector, which supports random access, a list can be accessed sequentially only. Since lists are bidirectional, they may be accessed front to back or back to front.
A list has this template specification:
template <class T, class Allocator = allocator<T> > class list
Here, T is the type of data stored in the list. The allocator is specified by Allocator, which defaults to the standard allocator. It has the following constructors:
explicit list(const Allocator &a = Allocator( ));
explicit list(size_type num, const T &val = T ( ),
const Allocator &a = Allocator( ));
list(const list<T, Allocator> &ob);
template <class InIter>list(InIter start, InIter end, const Allocator &a = Allocator( ));
The first form constructs an empty list. The second form constructs a list that has num elements with the value val, which can be allowed to default. The third form constructs a list that contains the same elements as ob. The fourth form constructs a list that contains the elements in the range specified by the iterators start and end.

Maps

The map class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming.
As mentioned, a map can hold only unique keys. Duplicate keys are not allowed. To create a map that allows nonunique keys, use multimap. The map container has the following template specification:
template <class Key, class T, class Comp = less<Key>, class Allocator = allocator<pair<const key, T> > class map