TRIBHUVAN UNIVERSITY
Institution of Science and Technology

Course Title: Object Oriented Programming
Course No: CSC161
Level: B. Sc CSIT First Year/ Second Semester

Full Marks: 60
Pass Marks: 24
Time: 3 Hrs

MODEL QUESTIONS-ANSWERS

Section A

Long Answer Questions [2*10=20]

Attempt any two questions.

1. ***What is object oriented programming? Explain objects, class, encapsulation, data hiding, inheritance, and polymorphism.***

Answer: Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

Object

Any entity that has state and behavior is known as an object. For example a chair, pen, table, keyboard, bike, etc. It can be physical or logical.An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If one task is performed by different ways, it is known as polymorphism. For example: to convince the customer

differently, to draw something, for example, shape, triangle, rectangle, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.In other words, the user will have the information on what the object does instead of how it does it.

By hiding complex steps/details/computations/statements inside your classes and creating public methods to access them.

Encapsulation

Encapsulation is a mechanism of wrapping the data (instance variables) and code acting on the data (methods) together as a single unit like a Class.The main purpose of encapsulation is you would have full control on data by using the code.In encapsulation, the variables of a class can be made hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.For example capsule, it is wrapped with different medicines.

2. ***Explain operator overloading. Write a program that overloads insertion and extraction operators.***

Answer: Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

The operators '<<' and '>>' are called like 'cout<< ob1' and 'cin>> ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
class time1
{
```

```
private:
inthrs;
intmins;
public:
time1():
{
hrs=0;
mins=0;
}

time1(int h, int m)
{
hrs=h;
mins=m;

}
friendistream& operator >> (istream&tin, time1 &t);
friendostream& operator << (ostream&tout, time1 &t);
};
istream& operator >>(istream&tin, time1 &t)
{
tin>>t.hrs;
tin>>t.mins;
return tin;
}
ostream& operator << (ostream&tout, time1 &t)
{
tout<<t.hrs<<" : "<<t.mins<<endl;
return tout;
}
int main()
{
time1 t1;
cout<<"Enter time t1 : ";cin>>t1;
cout<<"Time t1 = "<<t1;
time1 t2(11,23);
cout<<"Time t2 is "<<t2;
getch();
return(0);
}
```

3. ***What is inheritance? Explain the ambiguities associated with multiple inheritances with suitable example programs.***

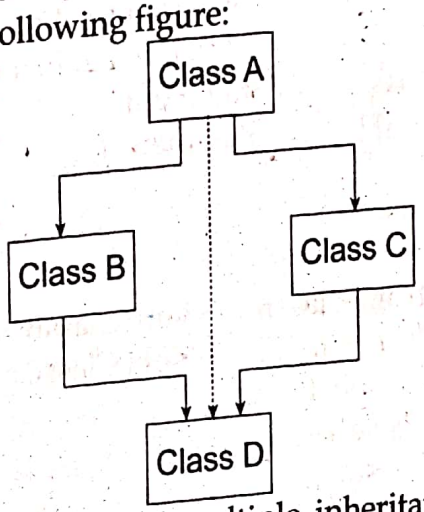
Answer:Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:**The class whose properties are inherited by sub class is called Base Class or Super class.

Ambiguity in multiple inheritances

Ambiguity in C++ occur when a derived class have two base classes and these two base classes have one common base class. Consider the following figure:



Consider an example of multiple inheritances in which there are two base classes having functions with the same name and a class derived from both these base classes having no function with this name.

```
#include<iostream.h>
#include<conio.h>
classClassA
{
public:
int a;
};
classClassB : public ClassA
{
public:
int b;
};
```

```
Class ClassC : public ClassA
{
public:
int c;
};
classClassD : public ClassB, public ClassC
{
public:
int d;
};
void main()
{
ClassDobj;
//obj.a = 10;              //Statement 1, Error occur
//obj.a = 100;             //Statement 2, Error occur
obj.ClassB::a = 10;      //Statement 3
obj.ClassC::a = 100;     //Statement 4
obj.b = 20;
obj.c = 30;
obj.d = 40;
cout<< "\n A from ClassB : "<<obj.ClassB::a;
cout<< "\n A from ClassC : "<<obj.ClassC::a;
cout<< "\n B : "<<obj.b;
cout<< "\n C : "<<obj.c;
cout<< "\n D : "<<obj.d;
 }
Output:
 A from ClassB : 10
 A from ClassC : 100
B : 20
C : 30
D : 40
```

In the above example, both Class B & Class C inherit Class A, they both have single copy of Class A. However Class D inherit both Class B & Class C, therefore Class D have two copies of Class A, one from Class B and another from Class C.

If we need to access the data member a of Class A through the object of Class D, we must specify the path from which a will be accessed, whether it is from Class B or Class C, bco'z compiler can't differentiate between two copies of Class A in Class D. There are two ways to avoid c++ ambiguity.

- Using scope resolution operator
- Using virtual base class

Section B

Short Answer Questions

Attempt any eight questions. **[8*5=40]**

4. ***Explain the purpose of a namespace with suitable example.***

Answer: Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

When a program uses different classes and functions written by different programmers, there is a possibility that two programmers will use the same name for two different things. Namespaces are a way to deal with this problem. A namespace is a collection of name definitions, such as class definitions and variable declarations. A namespace can, in a sense, be turned on and off so that when some of its names would otherwise conflict with names in another namespace, it can be turned off.

Using namespace std (and why to avoid it)

It is a way to access identifiers inside a namespace is to use a using directive statement. Here's "Hello world" program with a using directive:

```
#include <iostream>
using namespace std; // this is a using directive telling the compiler to check the std namespace when resolving identifiers with no prefix
int main()
{
cout<< "Hello world!"; // cout has no prefix, so the compiler will check to see if //cout is defined locally or in namespace std
return 0;
}
```

A using directive tells the compiler to check a specified namespace when trying to resolve an identifier that has no namespace prefix. So in the above example, when the compiler goes to determine what identifier cout is, it will check both locally (where it is undefined) and in the std namespace (where it will match to std::cout).

Creating a Namespace

Creating a namespace is similar to creation of a class.

```
namespaceMySpâce
{
```

```
// declarations
}
int main()
{
// main function
}
```

5. ***What is the principle reason for passing arguments by reference? Explain with suitablecode.***

Answer: Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.For example, suppose that we called our function addition using the following code:

```
#include <iostream>
int addition (int&a, int&b)
{
int r;
r=a+b;
return (r);
}
int main ()
{
int z;
int x=5, y=3;
z = addition (x,y);
cout<< "The result is " << z;
return 0;
}
```

When the function addition is called, the value of its local variables a and b points to the same memory location respectively, therefore any modification to either a or b within the function addition will also have effect in the values of x and y outside it.

6. ***Why constructor is needed? Explain different types of constructors with example.***

Answer: A constructor is used to initialize members in objects. Any instantiated object may require some predefined values, so those values can be specified using the constructor. You may choose to initialize each object manually by function but the function needs to call each time after creating the object. If due to any reason initialization function is not called, you may end up with uninitialized objects. Constructor on other hand is called automatically after creating the object, so is a better way of initialization.

Characteristics constructor:

- Constructor name class name must be same.
- Constructor doesn't return value.
- Constructor is invoked automatically, when the object of class is created.

Types of Constructor

- Default Constructor
- Parameterize Constructor
- Copy Constructor

Default Constructor

Construct without parameter is called default constructor.

Example of C++ default constructor

```
#include<iostream.h>
#include<string.h>
class Student
{
int Roll;
char Name[25];
float Marks;
public:
Student()        //Default Constructor
{
Roll = 1;
strcpy(Name,"Kumar");
Marks = 78.42;
}
void Display()
{
cout<<"\n\tRoll : "<<Roll;
cout<<"\n\tName : "<<Name;
cout<<"\n\tMarks : "<<Marks;
```

```
}
};
void main()
{
Student S;        //Creating Object
S.Display();      //Displaying Student Details
}
```

Output:

```
Roll : 1
Name : Kumar
Marks : 78.42
```

C++ Parameterize Constructor

Construct with parameter is called parameterize constructor.

Example of C++ parameterize constructor

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
int Roll;
char Name[25];
float Marks;
public:
Student(intr,char nm[],float m)      //Parameterize Constructor
{
Roll = r;
strcpy(Name,nm);
Marks = m;
}
void Display()
{
cout<<"\n\tRoll : "<<Roll;
cout<<"\n\tName : "<<Name;
cout<<"\n\tMarks : "<<Marks;
}
};
void main()
{
Student S(2,"Sumit",89.63);
//Creating Object and passing values to Constructor
```

```
S.Display();
//Displaying Student Details
}
```

Output:

Roll: 2
Name: Sumit
Marks: 89.63

In parameterize constructor, we have to pass values to the constructor through object.

Copy Constructor

Initialization of an object through another object is called copy constructor. In other words, copying the values of one object into another object is called copy constructor.

Example of C++ copy constructor

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Student
{
int Roll;
char Name[25];
float Marks;
public:
Student(intr,char nm[],float m)
//Constructor 1 : Parameterize Constructor
{
Roll = r;
strcpy(Name,nm);
Marks = m;
}
Student(Student &S)   //Constructor 2 : Copy Constructor
{
Roll = S.Roll;
strcpy(Name,S.Name);
Marks = S.Marks;
}
void Display()
{
cout<<"\n\t Roll : "<<Roll;
cout<<"\n\t Name : "<<Name;
cout<<"\n\t Marks : "<<Marks;
```

```
}
};
void main()
{
Student S1(2,"Sumit", 89.63);
Student S2(S1);   //Statement 1
cout<<"\n\t Values in object S1";
S1.Display();
cout<<"\n\t Values in object S2";
S2.Display();
}
```

Output:

```
Values in object S1
Roll : 2
Name :Sumit
Marks : 89.63

Values in object S2
Roll : 2
Name :Sumit
Marks : 89.63
```

In the above example, Statement 1 is creating an object S2 and passing another object S1 as parameter to the constructor 2. Constructor 2 will take the reference of object S1 passed by the statement 1 and copy all the values of object S1 to data members associated to the object S2.

7. ***Write a program that illustrates the conversions between objects of different classeshaving conversion function in source object.***

Answer: This conversion is exactly like conversion of user-defined type to basic type i.e. overloading the cast operator is used.

E.g.

```
class A obj A;
class B obj B;
objA = objB;
```

Here, objB is the source of the class class B and obj A is the destination object of class class A. Conversion routine is specified as conversion (cast) operator overloading for the type of destination class.

Syntax:

```
//destination object class
Class classA
{
//body of classA
};
//source object class
Class classB
{
private:
//....
public:
operator classA () //cast operator destination types
{
//code for conversion from classB
//to classA
}
};
```

Example: Conversion from user defined to user defined. The conversion routine in source class

```
#include<iostream>
#include<cmath>
using namespace std;
class Cartesian
{
private:
floatxco, yco;
public:
Cartesian()
{
xco=0;
yco=0;
}
Cartesian(float x, float y)
{
xco=x;
yco=y;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")";
}
```

```
};
class Polar
{
private:
float radius, angle;
public:
Polar()
{
radius=0;
angle =0;
}
Polar(float rad, float ang)
{
radius =rad;
angle=ang;
}
operator Cartesian()
{
float x=static_cast<int>(radius * cos(angle));
float y=static_cast<int>(radius * sin(angle));
return Cartesian(x, y);
}
void display()
{
cout<<"("<<radius<<","<<angle<<")";
}
};
int main()
{
Polar pol(10.0, 0.78);
Cartesian cart;
cart=pol;
cout<<"\n Given Polar: ";
pol.display();
cout<<"\n Equivalent Cartesian: ";
cart.display();
return 0;
}
```

The output of the program is:

Give Polar: (10.0, 0.78)

Equivalent Cartesian: (7,7)

8. ***Explain the difference between private and public inheritance with suitable diagram.***

Answer: When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

- **Public Inheritance** – When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
- **Protected Inheritance** – When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
- **Private Inheritance** – When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Private Inheritance Example Program

In order to understand how to use private inheritance, let us look at the following example program that uses two classes: the parent and the child classes.

```
#include <iostream>
using namespace std;
class Parent
{
public:
voidparentMethod( void )
{
cout<<"Inside parent method"<<endl;
}
};
class Child : private Parent
{
public:
voidchildMethod( void)
{
```

```
cout<<"Inside child method"<<endl;
parentMethod();
}
};
int main( void )
{
Child C;
C.childMethod();
return 0;
}
```

In the above example code:

- We created one object of the Child type with name "C"
- Then we applied childMethod(), which has some message and it will in-turn call the method parentMethod() that is placed in its body.
- If you try to call method parentMethod() on the object "C", you will get error message. From this we observe the most important property of private inheritance that it will disable the child object to accidental access some of the grandparent methods that would get inherited with public inheritance.

9. ***Why friend function is required? Discuss with example.***

Answer: If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function. The complier knows a given function is a friend function by the use of the keyword friend. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes. It works as bridge between classes.

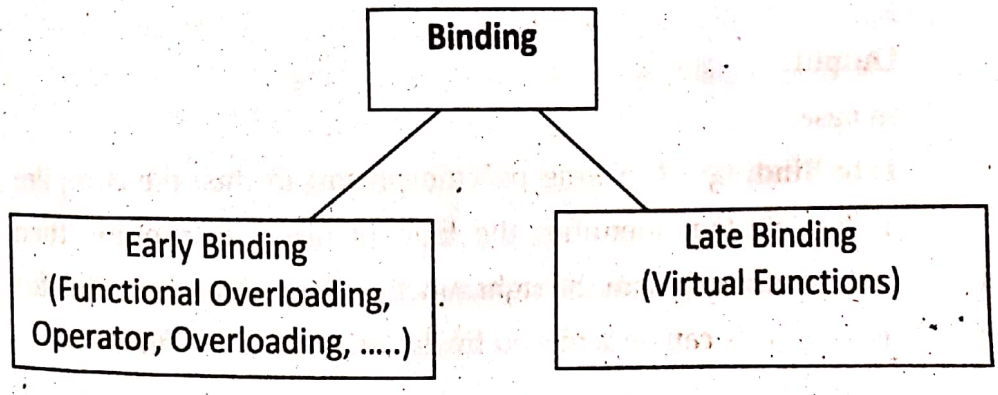
- Friend function must be declared with friend keyword.
- Friend function must be declare in all the classes from which we need to access private or protected members.
- Friend function will be defined outside the class without specifying the class name.
- Friend function will be invoked like normal function, without any object.

Example

```
#include <iostream>
using namespace std;
class Box
{
private:
int length;
public:
Box(): length(0)
{
}
friendintprintLength(Box); //friend function
};
intprintLength(Box b)
{
b.length += 10;
returnb.length;
}
int main()
{
Box b;
cout<<"Length of box: "<<printLength(b)<<endl;
return 0;
}
```

10. ***How late binding is different from early binding. Write a program that explains latebinding using virtual function.***

Answer: Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early Binding (compile-time time polymorphism) as the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function. By default early binding happens in C++. Late binding (discussed below) is achieved with the help of virtual keyword)

Example:

```
#include<iostream>
using namespace std;
class Base
{
public:
void show()
{
cout<<" In Base \n";
}
};
class Derived: public Base
{
public:
void show()
{
cout<<"In Derived \n";
}
};
int main(void)
{
Base *bp = new Derived;
/*The function call decided at compile time (compiler sees type of pointer and calls base class function. */
bp->show();
return 0;
}
```

Output:

In Base

Late Binding: (Run time polymorphism) in this, the compiler adds code that identifies the kind of object at runtime then matches the call with the right function definition (Refer this for details). This can be achieved by declaring a virtual function.

Example:

```
#include<iostream>
using namespace std;
class Base
{
public:
virtual void show()
{
cout<<" In Base \n";
}
};
class Derived: public Base
{
public:
void show()
{
cout<<"In Derived \n";
}
};
int main(void)
{
Base *bp = new Derived;
bp->show(); // RUN-TIME POLYMORPHISM
return 0;
}
```

Output:

In Derived

11. ***Why do we need exceptions? Explain "exceptions with arguments" with suitableprogram.***

Answer: Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions:

a. Synchronous,

b. Asynchronous (Ex: which are beyond the program's control, Disc failure etc.).

C++ provides following specialized keywords for this purpose

- **try**: represents a block of code that can throw an exception.
- **catch**: represents a block of code that is executed when a particular exception is thrown.
- **throw**: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

The following example shows handling of division by zero exception.

```
#include<iostream>
using namespace std;
double division(int var1, int var2)
{
if (var2 == 0)
{
throw "Division by Zero.";
}
return (var1 / var2);
}
int main()
{
int a = 30;
int b = 0;
double d = 0;
try
{
d = division(a, b);
cout<< d <<endl;
}
catch (const char* error)
{
cout<< error <<endl;
}
return 0;
}
```

12. ***What are the advantages of using the stream classes for I/O? Write a program that writesobject to a file.***

Answer: The C++ I/O stream classes provide the same facilities for input and output as C stdio.h. The I/O stream classes in the Standard C++ Library have the following advantages:

- The input (>>) operator and output (<<) operator are type safe. These operators are easier to use than scanf() and printf().
- We can overload the input and output operators to define input and output for your own types and classes. This makes input and output across types, including your own, uniform.

Program part:

```
#include <iostream>
#include <fstream>
using namespace std;
class student
{
private:
char name[30];
int age;
public:
voidgetData(void)
{
cout<<"Enter name:"; cin.getline(name,30);
cout<<"Enter age:"; cin>>age;
}
voidshowData(void)
{
cout<<"Name:"<<name<<",Age:"<<age<<endl;
}
};
int main()
{
student s;
ofstream file;
file.open("aaa.txt", ios::out);
if(!file)
{
cout<<"Error in creating file.."<<endl;
return 0;
}
s.getData();  //read from user
file.write((char*)&s, sizeof(s));  //write into file
file.close();  //close the file
cout<<"\nFile saved and closed successfully"<<endl;
ifstream file1;  //re open file in input mode and read data
file1.open("aaa.txt", ios::in);  //again open file in read mode
if(!file1)
{
cout<<"Error in opening file..";
return 0;
}
file1.read((char*)&s, sizeof(s)); //read data from file
s.showData();  //display data on monitor
file1.close();
return 0;
}
```

Tribhuvan University
Institute of Science and Technology

Course Title: Object Oriented Programming — Full Marks: 60
Course No: CSC161 — Pass Marks: 24
Level: B. Sc. CSIT First Year/ Second Semester — Time: 3 Hrs.

TU QUESTIONS-ANSWERS 2075

Section A

Long Answer Questions

Attempt any two questions. **[2*10=20]**

1: *Explain the concept of user-defined to user-defined data conversion with the conversion routine located in the destination class.*

Answer: This conversion is exactly like conversion of basic to user defined data type i.e. one argument constructor is used.Conversion routine is defined as a one argument constructor in destination class and takes the argument of the source class type.

E.g.

```
classA objA;
classB objB;
objA=objB;
```

Here, objB is the source of the class classB and objA is the destination object of class classA. For this expression to work, the conversion routine should exist in class A (destination class type) as a one argument constructor as,

```
//source object class
Class classB
{
//body of classB
}
//destination object class
Class classA
{
private:
//....
public:
classA (class BobjB) //object of source class
{
//code for conversion from classB to classA
```

```
}
};
```

Complete Example in C++

```
class distance
{
int meter;
float centimeter;
public:
distance(int m, int c)
{
meter = m;
centimeter = c;
}
Int getmeter()
{
return meter;
}
Float getcentimeter()
{
return centimeters;
}
};
Class dist
{
int feet;
int inch;
public:
dist(int f, inti)
{
feet = f;
inch = i;
}
dist(distance d)
{
Int m,c;
m=d.getmeter();
c=d.getcentimeter();
feet= m*3.3;
inch= c*0.4;
feet=feet+inch/12;
inch= inch%12;
}
```

```
void display()
{
cout<<"Feet = "<<feet<<endl<<"Inches = "<<inches;
},
};
void main( )
{
clrscr();
distance d1(6,40);
dist d2;
d2=d1;
d2.display();
getche();
}
```

2. ***Depict the difference between private and public derivation. Explain derived class constructor with suitable example.***

Answer:

- **Private Modifier:** The scope of private members are restricted to its own class. Private members can't be accessed by the derived class or in main() function.
- **Public Modifier:** Public members can be accessed by its own class, derived class and in main() function.

	Own class	Derived class	main()
Private	√	×	×
Protected	√	√	×
Public	√	√	√

A constructor plays a vital role in initializing an object. An important note, while using constructors during inheritance, is that, as long as a base class constructor does not take any arguments, the derived class need not have a constructor function. However, if a base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor. Remember, while applying inheritance, we usually create objects using derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base class contains constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base class is constructed in the same order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructor will be executed in the order of inheritance.

The derived class takes the responsibility of supplying the initial values to its base class. The constructor of the derived class receives the entire list of required values as its argument and passes them on to the base constructor in the order in which they are declared in the derived class. A base class constructor is called and executed before executing the statements in the body of the derived class.

```
// program to show how constructors are invoked in derived
class
#include <iostream.h>
class alpha
{
private:
int x;
public:
alpha(inti)
{
x = i;
cout<< "\n alpha initialized \n";
}
Void show_x()
{
cout<< "\n x = "<<x;
}
};
class beta
{
private:
float y;
public:
beta(float j)
{
y = j;
cout<< "\n beta initialized \n";
}
Void show_y()
{
cout<< "\n y = "<<y;
```

```
}
};
class gamma : public beta, public alpha
{
private:
int n, m;
public:
gamma(int a, float b, int c, int d):: alpha(a), beta(b)
{
m = c;
n = d;
cout<< "\n gamma initialized \n";
}
Void show_mn()
{
cout<< "\n m = "<<m;
cout<< "\n n = "<<n;
}
};
void main()
{
gamma g(5, 7.65, 30, 100);
cout<< "\n";
g.show_x();
g.show_y();
g.show_mn();
}
```

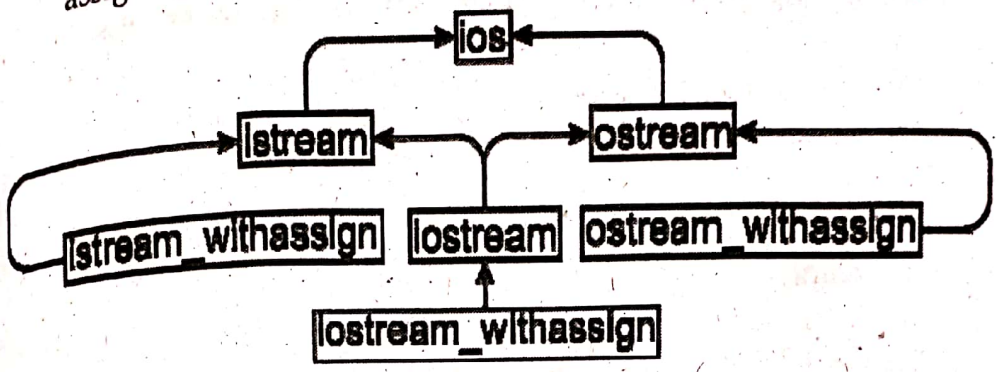
Output:

```
beta initialized
alpha initialized
gamma initialized
x = 5
y = 7.65
m = 30
n = 100
```

3. ***Briefly explain the hierarchy of stream classes. Write a program that overloads extraction and insertion operators.***

Answer: All the stream classes are derived from the base class ios, which stores the state of the stream and handles error. The ios class has an associated streambuf object that acts as the buffer for the stream. The istream and ostream classes, devided from ios, are meant for input and output, respectively. The iostream

class uses multiple inheritance to acquire the capabilities of both istream and ostream class and there for support both input and out. The classes iostream_withassign, ostream_withassign and istream_withassign are derived from istream, ostream and iostream, respectively, by adding the definition of the assignment operator (=) so that we can redirected I/O by assigning one stream to another.



C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object. Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object. Following example explains how extraction operator >> and insertion operator <<.

```
#include <iostream>
using namespace std;
class Distance
{
private:
int feet;          // 0 to infinite
int inches;        // 0 to 12
public:
// required constructors
Distance()
{
 feet = 0;
 inches = 0;
}
Distance(int f, int i)
{
 feet = f;
 inches = i;
```

```
}
friend ostream &operator<<( ostream &output, const Distance
&D )
 {
output << "F : " << D.feet << " I : " << D.inches;
return output;
}
friend istream &operator>>( istream  &input, Distance &D )
{
input >> D.feet >> D.inches;
return input;
}
};
int main()
{
Distance D1(11, 10), D2(5, 11), D3;
cout << "Enter the value of object : " << endl;
cin >> D3;
cout << "First Distance : " << D1 << endl;
cout << "Second Distance :" << D2 << endl;
cout << "Third Distance :" << D3 << endl;
return 0;
}
```

Short answer questions

Group B

Attempt any eight questions

4. *Write a member function called reverseit() that reverses a string (an array of characters). Use a for loop that swaps the first and last character, then the second and next-to last characters and so on. The string should be passed to reverseit() as argument*

Answer:

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
#include<string.h>
void reverseit(char s[100])
{
int i, l;
char temp;
l=strlen(s);
int j=l-1;
```

```
for(i=0;i<l/2;i++)
{
temp=s[j];
s[j]=s[i];
s[i]=temp;
j--;
}
s[i]='\0';
cout<<s;
}
void main()
{
char st[100];
cout<<"Enter any string";
cin>>st;
reverseit(st);
getch();
}
```

5. ***What is the principle reason for using default arguments in the function? Explain how missing arguments and default arguments are handled by the function simultaneously?***

Answer: In C++, a function can be called without specifying all its arguments. But it does not work on any general function. The function declaration must provide default values for those arguments that are not specified. When the arguments are missing from function call, default value will be used for calculation.

```
#include<iostream.h>
float interest(int p, int t = 5, float r = 5.0);
void main()
{
float rate, i1,i2,i3;
int pr , yr;
cout<<"Enter principal, rate and year";
cin>>pr>>rate>>yr;
i1= interest(pr ,yr ,rate);
i2=interest(pr , yr);
i3=interest(pr);
cout<<i1<<i2<<i3;
return(0);
}
```

```
float interest(int p, int t, float r)
{
return((p*t*r)/100);
}
```

In the above program, t and r has default arguments. If we give, as input, values for pr, rate and yr as 5000, 10 and 2, the output will be

1000 500 1250

6. ***"An overloaded function appears to perform different activities depending the kind of data send to it". Justify the statement with appropriate example.***

Answer: Function that share the same name are said to be overloaded functions and the process is referred to as function overloading. I.e. function overloading is the process of using the same name for two or more functions. Each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters. The number, type or sequence of parameters for a function is called the function signature. When we call the function, appropriate function is called based on the parameter passed. Two functions differing only in their return type cannot be overloaded. For e.g.-

int add(int , int) and float add(int, int)

A function call first matches the declaration having the same number and type of arguments and then calls the appropriate function for execution.

```
#include<iostream.h>
float perimeter(float);
int perimeter(int,int);
int perimeter(int,int,int);
void main()
{
cout<<"Perimeter of a circle; "<<perimeter(2.0)<<endl;
cout<<"Perimeter of a rectangle: "<<perimeter(10,10)<<endl;
cout<<"Perimeter of a triangle: "<<perimeter(5,10,15);
return (0);
}
//function definitions
float perimeter(float r)
```

```
{
return(2*3.14*r);
}
int perimeter(int l,int b)
{
return(2*(l+b));
}
int perimeter(int a,int b,int c)
{
return(a+b+c);
}
```

In the above program, a function "perimeter" has been overloaded. The output will be as follows:

Perimeter of a circle 12.56
Perimeter of a rectangle 40
Perimeter of a triangle 30

7. ***Explain the default action of the copy constructor. Write a suitable program that demonstrates the technique of overloading the copy constructor.***

Answer: The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to –

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here;

```
Class name (const classname &obj)
{
// body of constructor
}
```

Example:

```
#include <iostream>
using namespace std;
class A
{
int x, y;
public:
A()
```

```
{
cout << "Default constructor called!";
}
A(int a, int b)
{
cout << "Parameterized Constructor called!\n";
x = a;
y = b;
}
A(const A &old)
{
// old is the old object being passed
x = old.x; //This object's x to old object's x
y = old.y;
cout << "Copy Constructor called!\n";
}
void print()
{
cout << x << " " << y << "\n";
}
};
int main()
{
// Sample Code to show default constructor
A obj(10, 20); // making a object of class A  -- >Implicit
A obj2(obj);  // Copy Constructor called old object 'obj' is passed
obj2.print();
return 0;
}
```

8. ***Briefly explain types of inheritance need in object oriented programming.***

Answer: Inheritance (or derivation) is the process of creating new classes, called derived classes, from existing classes, called base classes. The derived class inherits all the properties from the base class and can add its own properties as well. The inherited properties may be hidden (if private in the base class) or visible (if public or protected in the base class) in the derived class. Inheritance uses the concept of code reusability. Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance. Reusing existing code saves time and money and

increases program's reliability. An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Types of Inheritance in C++

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

Syntax:

```
class subclass_name : access_mode base_class
{
//body of subclass
};
```

2. **Multiple Inheritances:**

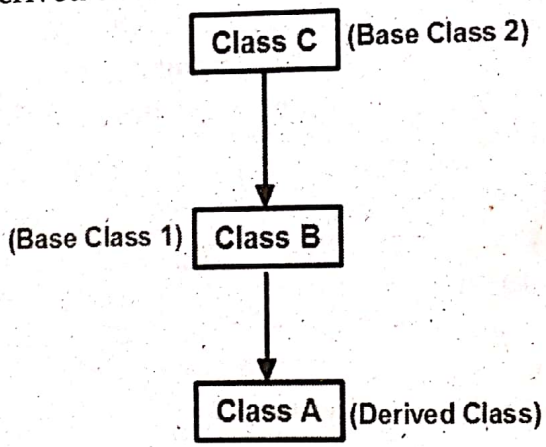
Multiple inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
//body of subclass
};
```

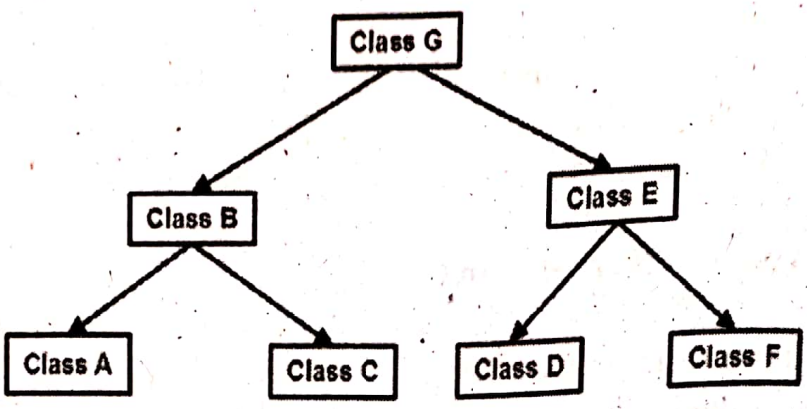
3. **Multilevel Inheritance**

In this type of inheritance, a derived class is created from another derived class.



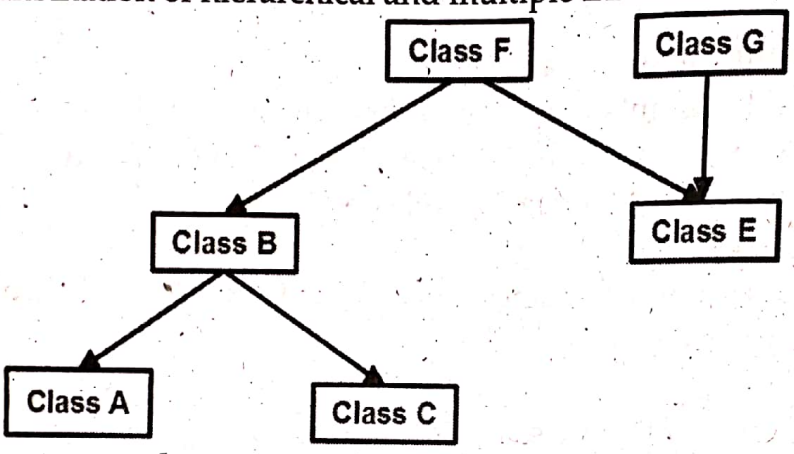
5. **Hierarchical Inheritance**

In this type of inheritance, more than one sub class is inherited from a single base class. I.e. more than one derived class is created from a single base class



Hybrid (Virtual) Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



9. *Create a real scenario where static data members are useful. Explain with suitable example.*

Answer: If a data member in a class is defined as static, then only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created. Hence, these data members are normally used to maintain values common to the entire class and are also called class variables.

Example:

```
class rectangle
{
private:
int length;
int breadth;
static int count; //static data member
public:
void setdata(int l, int b)
{
length = l;
```

```
breadth = b;
count++;
}
void displaycount()
{
cout<<count<<endl;
}
};
int rectangle :: count;
void main()
{
clrscr();
rectangle r1, r2, r3;
r1.displaycount();
r2.displaycount();
r3.displaycount();
r1.setdata(15, 6);
r2.setdata(9, 8);
r3.setdata(12, 9);
r1.displaycount();
r2.displaycount();
r3.displaycount();
getche();
}
```

In this program, the static variable count is initialized to zero when the objects are created. The count is incremented whenever data is supplied to an object. Since the data is supplied three times, the variable count is incremented three times. Because there is only one copy of the count shared by all the three objects, all the three calls to the display count member function cause the value 3 to be displayed.

10. ***Create a function called swaps() that interchanges the values of the two arguments count to it (pass these arguments by reference). Make the function into a template. No it can used with all numerical data types (char, int, float and so on). Write a main () program to execute the function with several types.***

Answer: Templates help in defining generic classes and functions and hence allow generic programming. Generic programming is an approach where generic data types are used as parameters and the same piece of code work for various data types. Function templates are used to create family of functions with different argument types. The format of a function template is shown below:

```
template<class T>
return_type function_name (arguments of type T)
{
   ....
   ....
}
```

I have written a program below which will swap two numbers using function templates.

```
#include<iostream>
using namespace std;
template <class T>
void swap(T&a,T&b)    //Function Template
{
T temp=a;
a=b;
b=temp;
}
int main()
{
 int x1=4,y1=7;
 float x2=4.5,y2=7.5;
cout<<"Before Swap:";
cout<<"nx1="<<x1<<"ty1="<<y1;
cout<<"nx2="<<x2<<"ty2="<<y2;
swap(x1,y1);
swap(x2,y2);
cout<<"nnAfter Swap:";
cout<<"nx1="<<x1<<"ty1="<<y1;
cout<<"nx2="<<x2<<"ty2="<<y2;
return 0;
}
```

11. ***Explain how exceptions are used for handling C++ errors in a symmetric and OOP oriented uses with the design that includes multiple exceptions.***

Answer: Exception handling mechanism in C++ is basically built upon three keywords: try, throw, and catch. The keyword try is used to surround a block of statements, which may generate exceptions. This block of statements is known as try block. When an exception is detected, it is thrown using the throw statement situated either in the try block or in functions that are invoked from within the try block. This is called throwing an exception and the point at which the throw is executed is called the throw point.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- **throw** – A program throws an exception when a problem shows up. This is done using a throw keyword.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **try** – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows –

```
try
{
// protected code
} catch( ExceptionName e1 )
{
 // catch block
} catch( ExceptionName e2 )
{
// catch block
} catch( ExceptionName eN )
{
// catch block
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
try
{
if(x == 0) throw x;
if(x == 1) throw 1.0;
}
catch(int m)
{
cout<<"Caught an integer\n";
}
catch(double d)
```

```
{
cout<<"Caught a double";
}
}
void main()
{
clrscr();
test(0);
test(1);
test(2);
getche();
}
```

Output:

Caught an integer
Caught a double

12. ***Use the character I/O differs from binary I/O? Explain with examples.***

Answer: The C++ language supports two types of files:

- Text files
- Binary files

The basic difference between text files and binary files is that in text files various character translations are performed such as "\r+\f" is converted into "\n", whereas in binary files no such translations are performed. By default, C++ opens the files in text mode.

Files Text	Binary Files
For writing data	
ofstream out ("myfile.txt"); or ofstream out; out.open("myfile.txt");	ofstream out ("myfile.txt",ios::binary); or ofstream out; out.open("myfile.txt", ios::binary);
For Appending (adding text at the end of the existing file)	

ofstream out("myfile.txt",ios::app);
or
ofstream out;
out.open("myfile.txt", ios::app);
	ofstream out
	("myfile.txt",ios::app | ios::binary);
	or
	ofstream out;
	out.open("myfile.txt", ios::app | ios::binary);

For reading data	
ifstream in ("myfile.txt"); or ifstream in ; in.open("myfile.txt");	ifstream in ("myfile.txt", ios::binary); or ifstream in ; in.open("myfile.txt", ios::binary);