Unit–3
Inheritances and Polymorphism

Que 1:Introduction of inheritance

- Inheritance is one of the key features of Object-oriented programming in C++.
- It allows user to create a new class (derived class) from an existing class(base class).
- The derived class inherits all the features from the base class and can have additional features of its own.
- The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class.
- The class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.
- Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

- **NOTE :** All members of a class except Private, are inherited

Que 2: Advantage of C++ Inheritance

- **Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.
- **Method Overriding** (Hence, Runtime Polymorphism.)
- Use of **Virtual** Keyword

Basic Syntax of Inheritance

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

Here, **subclass_name** is the name of the sub class.

- **base_class_name** is the name of the base class from which you want to inherit the sub class. Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

  Note: private member of the base class will never get inherited in the sub class.

  <u>Example</u>

```
#include<iostream.h>
#include<conio.h>
class p
{
public:
int a;
};
class b1:public p
{
public:
int b;
};
void main()
{
clrscr();
b1 c;
c.b=10;
c.a=11;
cout<<c.b<<endl;
cout<<c.a;


getch();
}
```

<u>OUTPUT</u>

```
10
11
```

In the above program the ‘b1’ class is publicly inherited from the ‘p’ class so the public data members of the class ‘p’ will also be inherited by the class ‘b1’.

Que 2: Inheritance using different access specifier

The access specifiers that are used are public, private and protected. When deriving a class from a base class, the base class may be inherited through public, private and protected inheritance.

1. Public Inheritance

When inheriting a class from a public parent class, public members of the parent class become public members of the child class and protected members of the parent class become protected members of the child class.

class Subclass : **public** Superclass

2. Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // **By default its private inheritance**

3. Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Table showing all the Visibility Modes

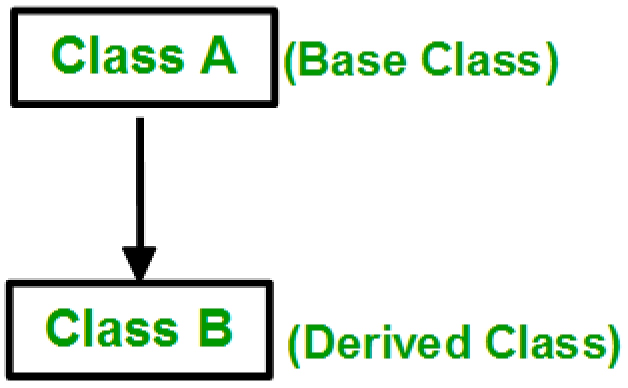
	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Que 3: Types of Inheritance

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual 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.



Example

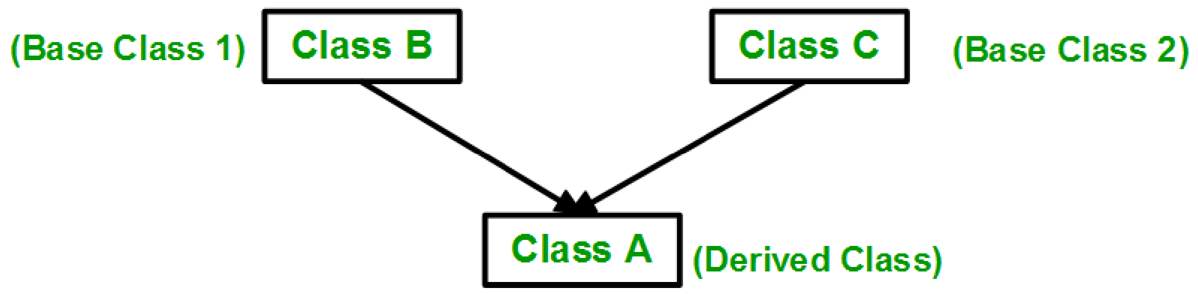
```
#include<iostream.h>
#include<conio.h>
class p
{
public:
int a;
};
class b1:public p
{
public:
int b;
};
void main()
{
clrscr();
b1 c;
c.b=10;
c.a=11;
cout<<c.b<<endl;
cout<<c.a;


getch();
}
```

OUTPUT

```
10
11
```

Multiple Inheritance

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
};
```

Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

Example

```
#include<iostream.h>
#include<conio.h>
class b1
{
public:
int x;
};

class b2
{
public:
int y;
};


class c:public b1,public b2
{
public:
int z;
};
```

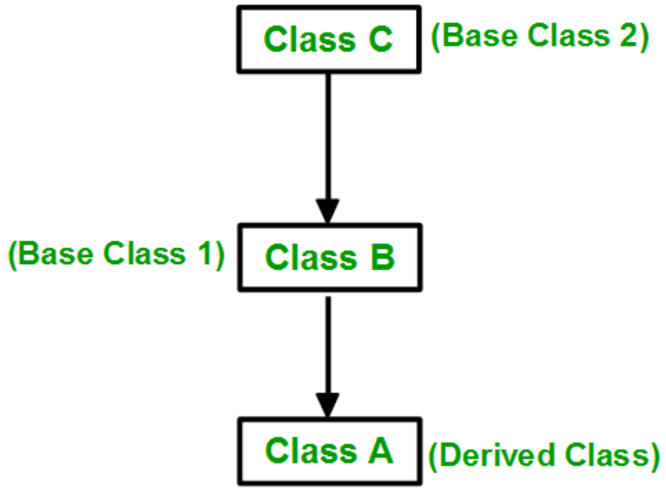
```
void main()
{
clrscr();
c obj;
obj.x=10;
obj.y=11;
obj.z=12;
cout<<obj.x<<endl;
cout<<obj.y<<endl;
cout<<obj.z;


getch();
}
```

<u>Output</u>

```
10
11
12
```

<u>Multilevel Inheritance</u>

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



Syntax:

```
class A

{

... .. ...

};

class B: public A

{

... .. ...

};

class C: public B

{

... ... ...

};
```

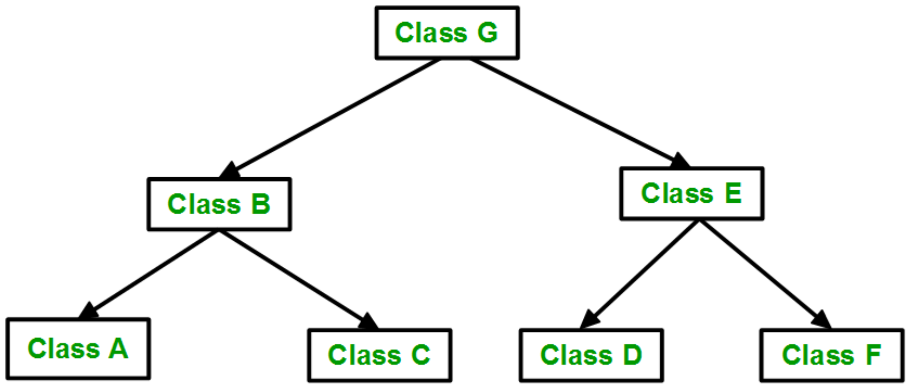
Example

```
#include<iostream.h>
#include<conio.h>
class b1
{
int a;
public:
int x;
     b1()
     {
     x=1000;
     a=100;
     cout<<x<<endl<<a<<endl;
     }

};
```

```
class d1:public b1
{
public:
int y;

};

class d2:public d1
{
public:
int z;
};

void main()
{
clrscr();
d2 obj;

obj.y=11;
obj.z=12;

cout<<obj.y<<endl;
cout<<obj.z;


getch();
}
```
<u>Output</u>

```
1000
100
11
12
```

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.

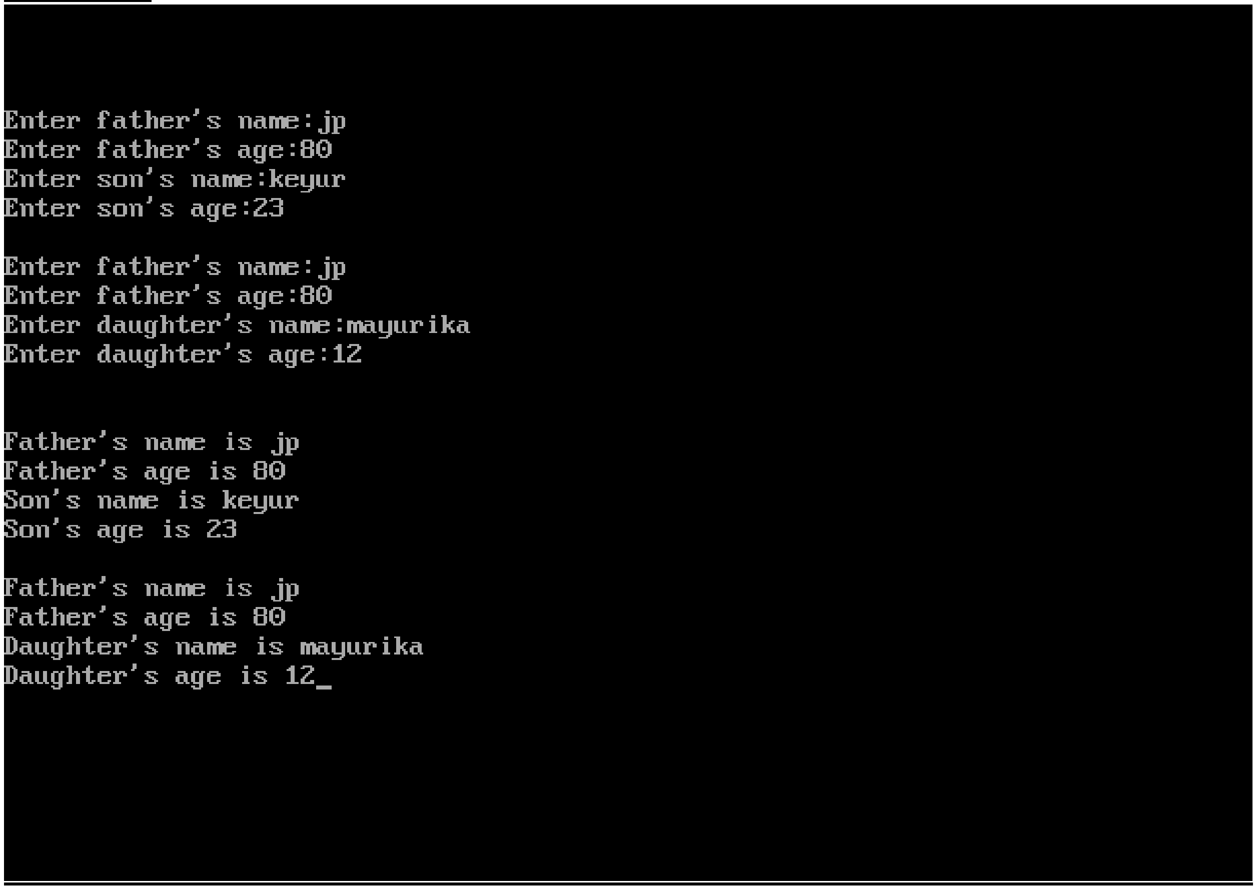


.

```
#include<iostream.h>
#include<Conio.h>
class father
// Base class derivation
{
  int age;
  char name [20];

  public:
    void get()
    {
      cout << "\nEnter father's name:";
      cin >> name;
      cout << "Enter father's age:";
      cin >> age;
    }

    void show()
    {
      cout << "\n\nFather's name is " << name;
      cout << "\nFather's age is " << age;
    }
};

class son : public father
```

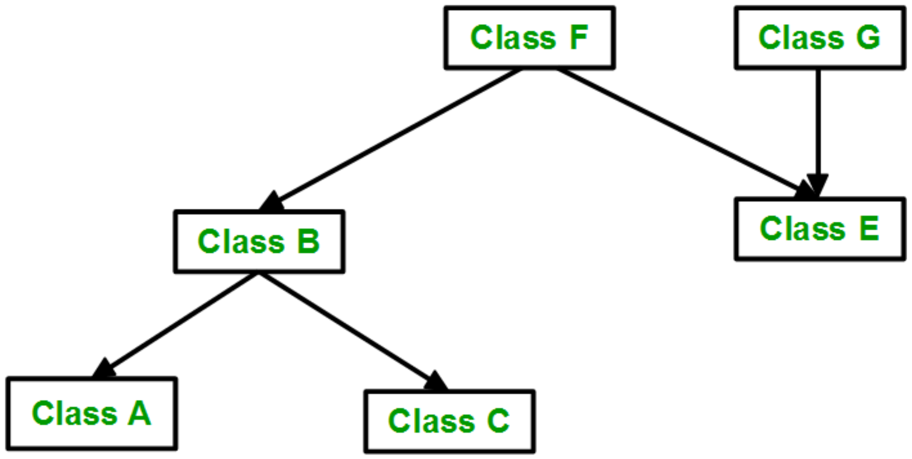
```
// First derived class derived from father class
{
  int age;
  char name [20];

  public:
    void get()
    {
      father :: get();
      cout << "Enter son's name:";
      cin >> name;
      cout << "Enter son's age:";
      cin >> age;
    }

    void show()
    {
      father::show();
      cout << "\nSon's name is " << name;
      cout << "\nSon's age is " << age;
    }
};

class daughter : public father
// Second derived class derived from the father class
{
  int age;
  char name [20];

  public:
    void get()
    {
      father :: get();
      cout << "Enter daughter's name:";
      cin >> name;
      cout << "Enter daughter's age:";
      cin >> age;
    }
    void show()
    {
      father::show();
      cout << "\nDaughter's name is " << name;
      cout << "\nDaughter's age is " << age;
    }
};

int main ()

{
   clrscr();
  son s1;
  daughter d1;
  s1.get();
  d1.get();
  s1.show();
  d1.show();
                getch();
getch();
  return 0;
}
```

OUTPUT



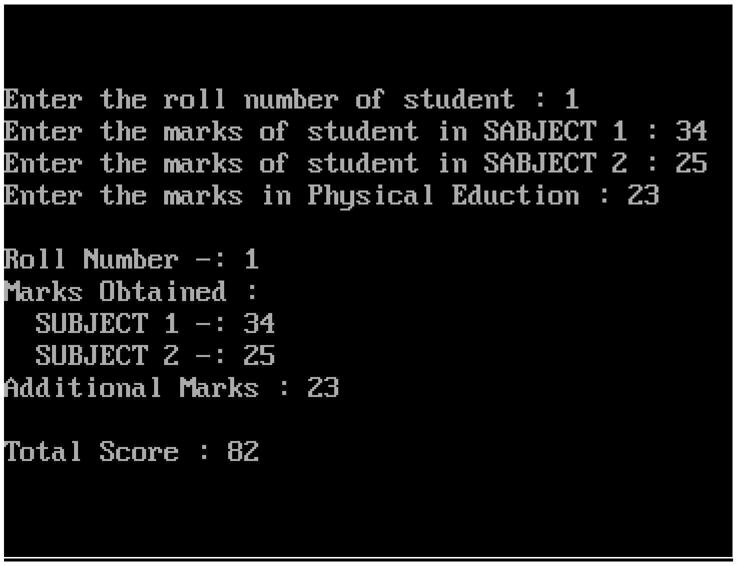
```
Enter father's name:jp
Enter father's age:80
Enter son's name:keyur
Enter son's age:23

Enter father's name:jp
Enter father's age:80
Enter daughter's name:mayurika
Enter daughter's age:12


Father's name is jp
Father's age is 80
Son's name is keyur
Son's age is 23

Father's name is jp
Father's age is 80
Daughter's name is mayurika
Daughter's age is 12_
```

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:



```
#include<iostream.h>
#include<conio.h>

class student
//base class derivation
{
 protected:
   int r_no;

 public:
   void getRollno()
   {
     cout << "Enter the roll number of student : ";
     cin >> r_no;
   }

   void putRollno()
   {
     cout << "\nRoll Number -: " << r_no << "\n";
   }
};

class test : public student
//intermediate base class
{
 protected:
   int part1, part2;

 public:
   void getMarks()
   {
     cout << "Enter the marks of student in SAB 1 : ";
     cin >> part1;
     cout << "Enter the marks of student in SAB 2 : ";
     cin >> part2;
   }

   void putMarks()
   {
     cout << "Marks Obtained : " << "\n";
           cout << "  Part 1 -: " << part1;
     cout << "\n  Part 2 -: " << part2 << "\n";
   }
};

class sports
{
 protected:
   int score;

 public:
   void getSportsMarks()
   {
     cout << "Enter the marks in Physical Eduction : ";
     cin >> score;
   }

   void putSportsMarks()
   {
     cout << "Additional Marks : " << score << "\n \n";
   }
};

class result : public test, public sports    //derived from test and sports
{
 int total;

 public:
   void display ()
   {
     total = part1 + part2 + score;
     putRollno();
     putMarks();
     putSportsMarks();

     cout << "Total Score : " << total ;
   }
};
```

```
int main ()
{
clrscr();
  result s1;
  s1.getRollno();
  s1.getMarks();
  s1.getSportsMarks();
  s1.display();
            getch();
  return 0;
}
```



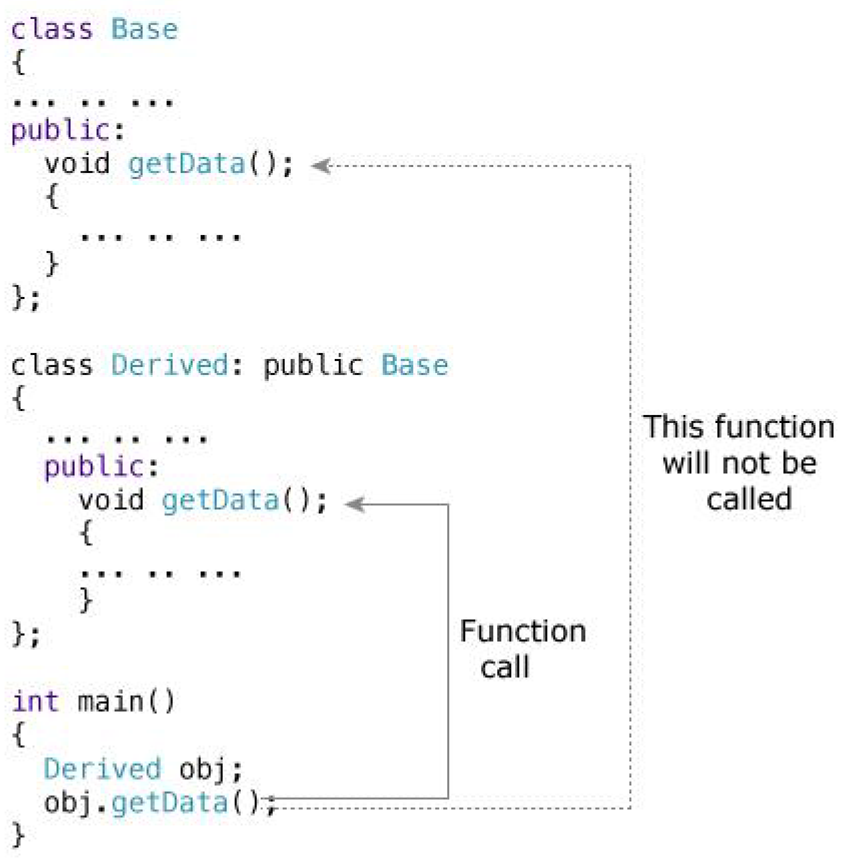
```
Enter the roll number of student : 1
Enter the marks of student in SABJECT 1 : 34
Enter the marks of student in SABJECT 2 : 25
Enter the marks in Physical Eduction : 23

Roll Number -: 1
Marks Obtained :
  SUBJECT 1 -: 34
  SUBJECT 2 -: 25
Additional Marks : 23

Total Score : 82
```

Que 4: Function Overriding in C++

- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.
- A child class inherits the data members and member functions of parent class, but when you want to override functionality in the child class then you can use function overriding.

- It is like creating a new version of an old function, in the child class.

- In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.



```
class Base
{
... .. ...
public:
  void getData();
  {
    ... .. ...
  }
};

class Derived: public Base
{
  ... .. ...
  public:
    void getData();
    {
    ... .. ...
    }
};

int main()
{
  Derived obj;
  obj.getData();
}
```

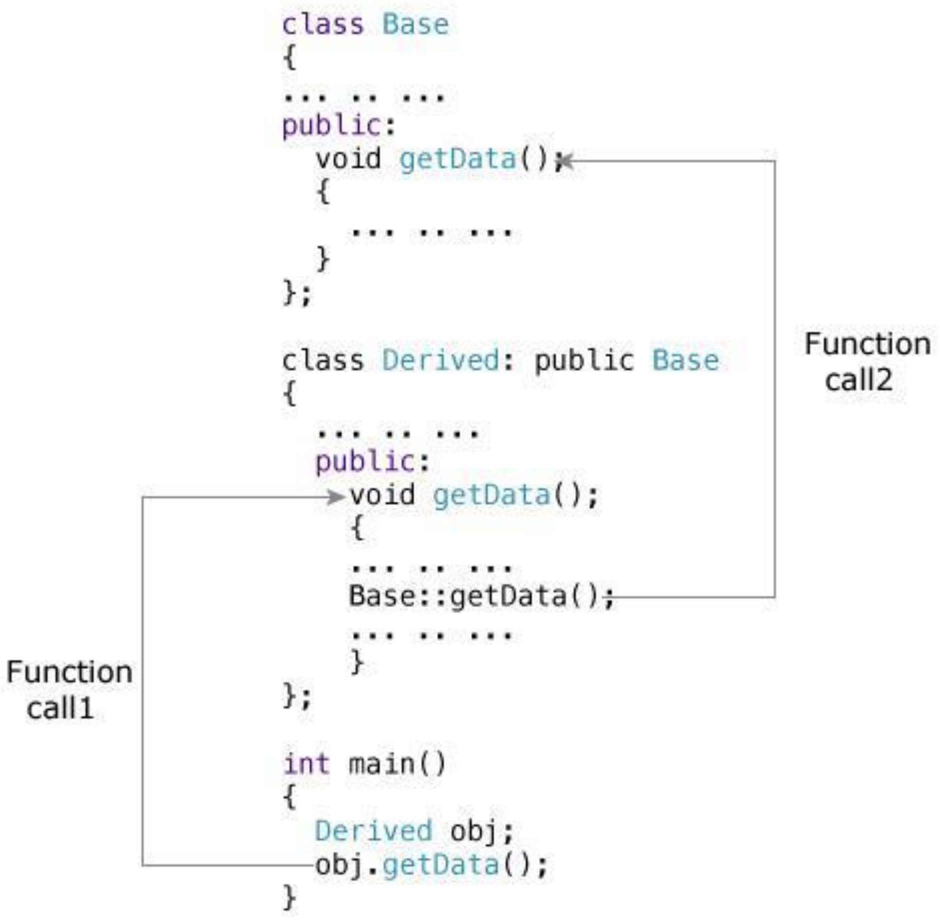
Example

```
#include<iostream.h>
#include<conio.h>
class base
{
   public:
         void show()
         {
         cout<<"It is base class";
         }
};
class derived: public base
{
  public:
          void show()
         {
     cout<<"It is Derived class";
     }

//   base::eat();}
};
void main()
{
         clrscr();
  derived d;
  d.show();
getch();

}
```

How to access the overridden function in the base class from the derived class?

- To access the overridden function of the base class from the derived class, scope resolution operator :: is used.
- **For example**, If you want to access getData() function of the base class, you can use the following statement in the derived class.

Base::getData();


```
class Base
{
... .. ...
public:
  void getData();
  {
    ... .. ...
  }
};

class Derived: public Base
{
  ... .. ...
  public:
    void getData();
    {
    ... .. ...
    Base::getData();
    ... .. ...
    }
};

int main()
{
  Derived obj;
  obj.getData();
}
```

```
#include<iostream.h>
#include<conio.h>
class base
{
   public:
         void show()
         {
         cout<<"It is base class";
         }
};
class derived: public base
{
  public:
          void show()
         {
```

```
    cout<<"It is Derived class"<<endl;
       base::show();
    }


};
void main()
{
       clrscr();
  derived d;
  d.show();
getch();

}
```

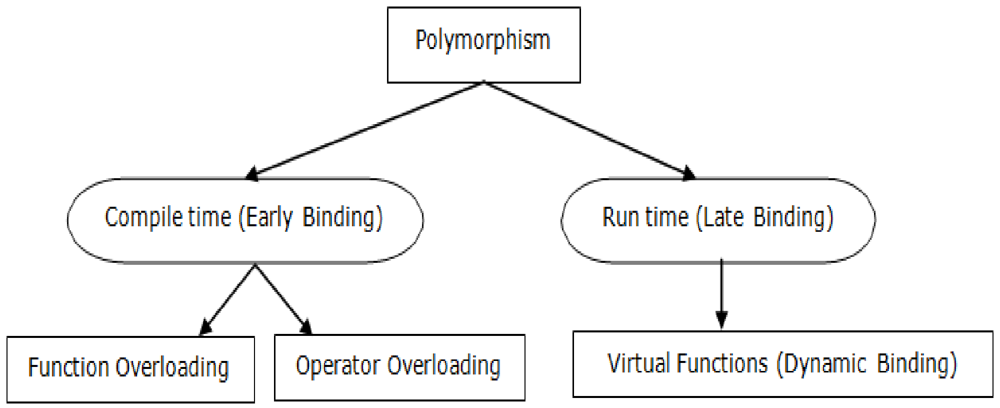
Que 5: C++ Polymorphism

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A real-life example of polymorphism, a person at the same time can have different characteristics.
- Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

Different types of polymorphism are



Compile time polymorphism (Early binding):

- When perform Early Binding, an object is assigned to a variable declared to be of a specific object type.
- Early binding objects are basically a strong type objects or static type objects.
- While Early Binding, methods, functions and properties which are detected and checked during compile time and perform other optimizations before an application executes.
- The biggest advantage of using early binding is for performance and ease of development.
- This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading: Function overloading is an example of static polymorphism. More than one function with same name, with different signature in a class or in a same scope is called function overloading.

Operator Overloading: Another example of static polymorphism is Operator overloading. Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.

Dynamic Binding (Late Binding)

- By contrast, in Late binding functions, methods, variables and properties are detected and checked only at the run-time.

- It implies that the compiler does not know what kind of object or actual type of an object or which methods or properties an object contains until run time.
- The biggest advantages of Late binding is that the Objects of this type can hold references to any object, but lack many of the advantages of early-bound objectsAt run-time, the code matching the object under current reference will be called.

Virtual function:

Virtual function is an example of dynamic polymorphism. Virtual function is used in situation, when we need to invoke derived class function using base class pointer. Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function. This is how we can achieve "Runtime Polymorphism".

Example of polymorphism

```
#include <iostream.h>
#include <conio.h>
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n" ;
    }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n" ;
    }
};

// Derived class
```

```
class Dog : public Animal {
 public:
   void animalSound() {
     cout << "The dog says: bow wow \n" ;
   }
};

int main() {
 Animal myAnimal;
 Pig myPig;
 Dog myDog;

 myAnimal.animalSound();
 myPig.animalSound();
 myDog.animalSound();
 return 0;
}
```

OUTPUT

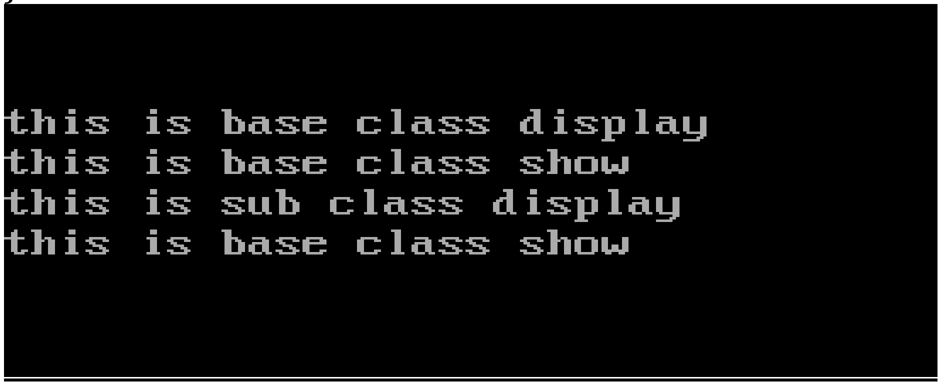
```
The animal makes a sound
The pig says : wee wee
The dog says : bow bow
```

Que 6 :Explain Virtual Function with example

- It is a run time polymorphism.
- Base class and derived class have same function name and base class pointer is assigned address of derived class object then also pointer will execute base class function.
- To execute function of derived class, we have to declare function of base class as virtual.
- To declare virtual function just uses keyword virtual preceding its normal function declaration.
- After making virtual function, the compiler will determine which function to execute at run time on the basis of assigned address to pointer of base class.

Example

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
  virtual void disp()
        {

        cout<<"this is base class display"<<endl;
        }
        void show()
        {
        cout<<"this is base class show"<<endl;
        }

};

class deri:public base
{
        void disp()
        {
        cout<<"this is sub class display"<<endl;
        }
        void show()
        {
        cout<<"this is sub class show"<<endl;
        }
};
void main()
{
clrscr();
base *b;
base objb;
deri objd;
b=&objb;
b->disp();
b->show();
b=&objd;
b->disp();
b->show();

getch();
}
```



```
this is base class display
this is base class show
this is sub class display
this is base class show
```

Rules for virtual function

- The virtual functions must be member of any class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- We cannot have virtual constructors, but we can have virtual destructors.
- The derived class pointer cannot point to the object of base class.
- When a base pointer points to a derived class, then also it is incremented or decremented only relative to its base type.
- Therefore we should not use this method to move the pointer to the next object.
- If a virtual function is defined in base class, it need not be necessarily redefined in the derived class. In such cases, call will invoke the base class.

Que 6: Explain Pointers to derived class with example.

- We can use pointers not only to the base class objects but also to the objects of derived classes.
- A single pointer variable can be made to point to objects belonging to different classes.

For example:

```
 B *ptr  //pointer to class B type variable

 B b; //base object

D d; // derived object

ptr = &b; // ptr points to object b
```

- In above example **B** is base class and **D** isa derived class from **B,** then a pointer declared as a pointer to **B** and point to the object b.
- We can make **ptr** to point to the object **d** as follow

    ptr = &d;
- We can access those members of derived class which are inherited from base class by base class pointer.
- But we cannot access original member of derived class which are not inherited by base class pointer.
- We can access original member of derived class which are not inherited by using pointer of derived class.

Example

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

    void show()
    {
      cout<<"This is base class"<<endl;
    }
};
class derive : public base
{
    public:

    void show()
    {
      cout<<"This is derived class"<<endl;
    }
};
int main()
{

   clrscr();
   base b;
   base *bptr;

   bptr=&b;

   bptr->show();
   derive d;

   bptr=&d;

 ((derive *)   bptr)->show();
   getch();
   return 0;
}
```

OUTPUT

```
This is base class
This is derived class
```

Que 7:Explain Pure virtual function

- A pure virtual function means ‘do nothing’ function.
- We can say empty function. A pure virtual function has no definition relative to the base class.
- Programmers have to redefine pure virtual function in derived class, because it has no definition in base class.

- A class containing pure virtual function cannot be used to create any direct objects of its own. This type of class is also called as abstract class.

Syntax:

virtual void display() = 0;

OR

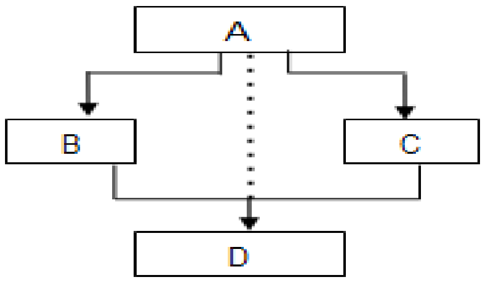
virtual void display()

{

}

Que 8: Explain virtual base class with example.

- It is used to prevent the duplication/ambiguity.
- In hybrid inheritance child class has two direct parents which themselves have a common base class.
- So, the child class inherits the grandparent via two separate paths. it is also called as indirect parent class.
- All the public and protected member of grandparent are inherited twice into child.



- We can stop this duplication by making base class virtual.

example:

#include<iostream.h>

#include<conio.h>

class A

```
{
public: int i;
};

class B : virtual public A
{
public: int j;
};
class C: virtual public A
{
public: int k;
};

class D: public B, public C
{
public: int sum;
};
void main()
{
clrscr();
D obj;
obj.j=10;
obj.k=20;
obj.i=30;
obj.sum=obj.i+obj.j+obj.k;
cout<<"sum="<<obj.sum;
getch();
}
```

Que 8: Explain Abstract class with example.

- Abstract Class is a class which contains at least one Pure Virtual function in it.
- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class.
- For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move.

  .

  Characteristics of Abstract Class

- We cannot create objects of abstract classes
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

<u>Example</u>

```
//Abstract base class

class Base

{

   public:

   virtual void show()
{
}   // Pure Virtual Function
};

class Derived:public Base
{
   public:
   void show()
   {
      cout << "Implementation of Virtual Function in Derived class\n";
   }
};
int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Output

```
Implementation of Virtual Function in Derived class
```

Que 9: Explain Virtual Destructor with example

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include<iostream.h>
#include<conio.h>


class base
 {
  public:
    base()
    {
cout<<"Constructing base \n";
 }
    ~base()
    {
cout<<"Destructing base \n";
 }
};

class derived: public base
{
  public:
    derived()
    {
 cout<<"Constructing derived \n"; }
    ~derived()
    {
 cout<<"Destructing derived \n"; }
};

Void main()
{
 Clrscr();
   derived *d = new derived();
   base *b = d;
   delete b;
   getch();
   return 0;
 }
```

Output

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
// A program with virtual destructor
#include<iostream.h>
#include<conio.h>


class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

void main()
```

```
{
 derived *d = new derived();
 base *b = d;
 delete b;
 getchar();
 return 0;
}
```

Output:

```
Constructing base

Constructing derived

Destructing derived

Destructing base
```

Que 10 : Explain Early Binding VS Late Binding

- In early binding, the compiler matches the function call with the correct function definition at compile time.
- It is also known as Static Binding or Compile-time Binding. By default, the compiler goes to the function definition which has been called during compile time.
- So, all the function calls you have studied till now are due to early binding.
- function overriding in which the base and derived classes have functions with the same name, parameters and return type.
- In that case also, early binding takes place.
- In function overriding, we called the function with the objects of the classes.
- Now let's try to write the same example but this time calling the functions with the pointer to the base class i.e., reference to the base class' object.

Example

```
#include <iostream.h>

#include <conio.h>


class Animals
{
        public:
                void sound()
                {
                        cout << "This is parent class" << endl;
                }
};

class Dogs : public Animals
{
        public:
                void sound()
                {
                        cout << "Dogs bark" << endl;
                }
};

Void main()
{
clrscr();
        Animals *a;
        Dogs d;
        a= &d;
        a -> sound();   //  early binding
        getch();
}
```

- Now in this example, we created a pointer a to the parent class Animals. Then by writing a= &d , the pointer 'a' started referring to the object d of the class Dogs.

- a -> sound(); - On calling the function sound() which is present in both the classes by the pointer 'a', the function of the parent class got called, even if the pointer is referring to the object of the class Dogs.
- This is due to Early Binding. We know that a is a pointer of the parent class referring to the object of the child class.
- Since early binding takes place at compile-time, therefore when the compiler saw that a is a pointer of the parent class, it matched the call with the 'sound()' function of the parent class without considering which object the pointer is referring to.

<u>Late Binding</u>

- In the case of late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as Dynamic Binding or Runtime Binding.
- In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.
- By default, early binding takes place. So if by any means we tell the compiler to perform late binding, then the problem in the previous example can be solved.
- This can be achieved by declaring a virtual function.

Note: Virtual function Example of late binding so in exam write example