4. Object Inheritance and Reusability

Introduction to Inheritance

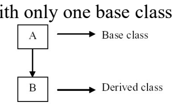
- Reusability is the important features of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save the time but also reduce frustration and increase reliability.
- OOP supports reusability which means same code can be used in different portion of the program. The concept of software reusability is provided by inheritance. Here, inheritance means reuse the properties of one class (existing class) by another class (new class). Due to inheritance, the length of code is reduced and program becomes more reliable. The new class can create which can access or reuse the properties of the base class or previously defined class. Here, old class is known as base class or super class and the new class is known as child class or derived class. The derived class inherits some or all the information from the base class. A class can also inherit properties from more than one class or from more than one level.

…

Types of Inheritance

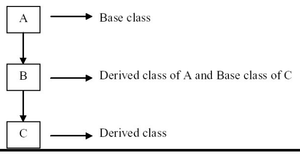
1. Single Inheritance

A derived class with only one base class is known as **single inheritance.**



2. Multilevel Inheritance

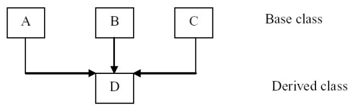
The mechanism of deriving a class from another derived class is known as **multilevel inheritance.**



…

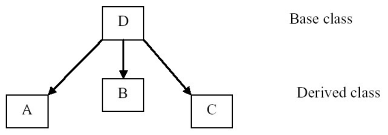
3. Multiple Inheritance

A derived class with several base classes is called **multiple inheritances.**



4. Hierarchical Inheritance

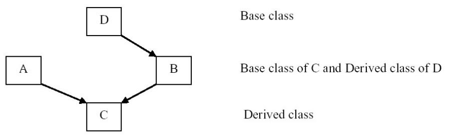
The traits of one base class may be inherited by more than one derived class. This process is known as **hierarchical inheritance.**



…

5. Hybrid Inheritance

The combination of different types of inheritance is known as **hybrid inheritance.**



Defining a Derived Class:

- A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived_class_name : visibility_mode base_class_name
{ .............
.............//members of derived classs };
```

…

- The colon indicates that the derived class name is derived from the base_class_name. The visibility is optional and if present may be either private or public. The default visibility mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

```
1. class ABC : private XYZ //private derivation
{
//members ABC };
2. class ABC : XYZ //private derivation
{ //members ABC };
3. class ABC : public XYZ //public derivation
{
//members ABC
};
```

…

- When the base class is privately inherited by a derived class, ‘public members’ of the base class becomes ‘private members’ of the derived class and therefore, the public members of the base class can only be accessed by the member function of the derived class. They are inaccessible to the objects of the derived class.
- When the base class in publicly inherited, ‘public members’ of the base class become ‘public members’ of the derived class and therefore, they are accessible to the objects of derived class.
- In both the cases, the private members are not inherited. So, the private members of the base class will never become the members of its derived class.

…

Single Inheritance

Syntax for single inheritance:

```
class A
{
//members of A
};
class B : public A
{
// members of B };
//Example single inheritance public
#include<iostream.h>
#include<conio.h>
class B
{public:
int a, b;//public ready for inheritance
void get_ab(int,int); };
class D: public B //public derivation
{int c;
public:
void mul();};
void B::get_ab(int x,int y)
{
a=x;b=y;}
void D :: mul()
{ c=a*b;
cout<<"Multiplication="<<c<<endl;
}
void main ()
{D d;
d.get_ab(5,10);
d.mul(); getch();}
```

…

```
//Example single inheritance private
#include<iostream.h>
#include<conio.h>
class B
{
public:
int a, b;//public ready for inheritance
void get_ab();};
class D: private B //private derivation
{int c;
public:
void mul();};
void B::get_ab()
{a=5;b=10; }
void D :: mul()
{get_ab();
c=a*b;
cout<<"Multiplication="<<c<<endl;
}
void main ()
{D d;
d.mul();
getch();}
```

Output:

```
Multiplication=50
```

…

```
//Example single inheritance
    public
#include<iostream.h>
#include<conio.h>
class B
{int a;//private not inheritable
public:
int b;//public inheritable
void get_ab();
int get_a();
void show_a();
};
class D: public B //public derivation
{
int c;
public:
void mul();
void display();
};
void B::get_ab()
{
a=5;b=10;}
```

…

```
int B :: get_a()
{
return a;
}
void B :: show_a()
{
cout<<"a="<<a<<endl;
}
void D :: mul()
{
c=b*get_a();
}
void D :: display()
{
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;}
void main ()
{D d;
d.get_ab();
d.mul();
d.show_a();
d.display();
d.b=20;
d.mul();
d.display();
getch();}
```

Output:

```
a=5
b=10
c=50
b=20
c=100
```

…

```
// Example single inheritance private
#include<iostream.h>
#include<conio.h>
class B
{int a;
public:
int b; void get_ab(void);
int get_a();
void show_a();};
class D: private B
{int c;
public:
void mul();
void display();};
void B::get_ab(void)
{cout<<"Enter a and b"<<endl;
cin>>a>>b;}
void D :: mul()
{get_ab();
c=b*get_a();}
int B :: get_a()
{return a;}
void B :: show_a()
{cout<<"a="<<a<<endl;}
void D :: display()
{show_a();
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;}
void main ()
{D d;
d.mul();
d.display();
getch();}
```

…

Making Private Member Inheritable:

- If the private data needs to be inherited by a derived class, then this can be accomplished by modifying the visibility limit of private member by making it public. This would make it accessible to all the other function of the program. C++ provides a third visibility modifier, protected, which stores a limited purpose in inheritance. A member declared as protected is accessible by the member function within its class and any class immediately derived from it. It cannot be accessed by the functions out of these two classes. A class using all 3 visibility modes is illustrated as below:

```
class alpha
{ private: //optional
.......... //visibility of the member function within the class
..........
protected:
.......... //visible to member functions of its own class and derived class
..........
public:
.......... //visible to all function in the program    .......... };
```

…

- When a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is accessible by the member function of the derived class. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member function of the derived class, it is not available for further inheritance (since private members cannot be inherited).
- It is also possible to inherit a base class in protected mode known as protected derivation. In protected derivation, both the public and protected members of the base class become protected to members of the derived class.
- The various functions that can have access to the private and protected member's function of a class are listed as below:

 A function that is friend of the class.

 A member functions of the class i.e. the friend of the class.

 A member functions of the derived class.

…

- While the friend functions and the member function of a class can have direct access to both the private and protected data, the member function of a derived class can directly access only the protected data. However, they can access the private data through the member functions of the base class.

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Table: Visibility of inherited members.

…

Multiple Inheritances

- Syntax:

```
class B1
{
…………
};
class B2
{
…………
};
class D : public B1, public B2,
    …….
{
………… };
```

```
//Example multiple inheritance
#include<iostream.h>
#include<conio.h>
class M
{protected:
int m;
public:
void get_m(int a)
{m=a;}};
class N
{protected:
int n;
public:
void get_n(int b)
{n=b;}};
```

…

```
class D : public M, public N
{
public:
void display()
{
cout<<"m="<<m<<endl;
cout<<"n="<<n<<endl;
cout<<"m*n="<<m*n<<endl;
}
};
void main()
{
D z;
z.get_m(10);
z.get_n(20);
z.display();
getch();}
```

Output:

```
m=10
n=20
m*n=200
```

Note: in multiple inheritance a problem may face due to **ambiguity** arises by the function with the same name appears in more than one base class. In such case function can be invoked by using scope resolution operator as:

o_name.c_name::f_name(arguments);

…

Hierarchical Inheritance

- Syntax:

```
class B
{ ………
};
class D1 : public B
{ ………
};
class D2 : public B
{ ………};
class D3 : public B
{ ………
};
```

```
//Example hierarchical inheritance
#include<iostream.h>
#include<conio.h>
class B
{
protected:
int x,y;
public:
void assign()
{x=10;y=20;}};
class D1 : public B
{protected:
int A;
public:
void add()
{A=x+y;
    cout<<"x+y="<<A<<endl;}};
```

…

```
class D2 : public B
{protected:
int S;
public:
void sub()
{S=x-y;
cout<<"x-y="<<S<<endl;}};
class D3 : public B
{protected:
int M;
public:
void mul()
{M=x*y;
cout<<"x*y="<<M<<endl;}};
```

```
void main()
{
D1 d1;
D2 d2;
D3 d3;
d1.assign();
d2.assign();
d3.assign();
d1.add();
d2.sub();
d3.mul();
getch();
}
```

```
Output:
x+y=30
x-y=-10
x*y=200
```

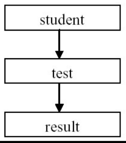
…

Multilevel Inheritance

Syntax:

```
class A
{
.........
};
class B : public A
{
...........
};
class C: public B
{
...........
};
```

Les us consider a simple example. Assume that the test results of a batch of students are stored in three different classes. Class student stores the rollno, class test stores the marks obtained in two subjects and class result contains the total obtained in the test. The class result can inherit the details of the marks obtained in the test and the rollno of the students through multilevel inheritance.



…

```
//Example multilevel inheritance
#include<iostream.h>
#include<conio.h>
class student
{protected:
int rollno;
public:
void get_number(int a)
{rollno=a;}
void put_number()
{cout<<"Roll No: "<<rollno<<endl;}};
class test: public student
{protected:
float sub1,sub2;
public:
void get_marks(float x, float y)
{sub1=x;sub2=y;}
void put_marks()
{cout<<"SUB1: "<<sub1<<endl;
cout<<"SUB2: "<<sub2<<endl;}};
class result : public test
{protected:
float totalmark;
public:
void display()
{put_number();
put_marks();
totalmark=sub1+sub2;
cout<<"Total is "<<totalmark<<endl;}};
void main()
{result s;
s.get_number(111);
s.get_marks(56.5,75.5);
s.display();getch();}
```

Output:

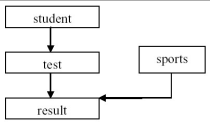
```
Roll No:111
SUB1:56.5
SUB2:75.5
Total is 132
```

…

Hybrid Inheritance

- Syntax:

```
class A
{
.......
.......
};
class C
{
.......
.......
};
class B : public A
{
.......
.......
};
class D : public B,public C
{
.......
.......
};
```



```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
int rollno;
public:
void get_number(int a)
{
rollno=a;
}
```

…

```
void put_number()
{cout<<"Roll No: "<<rollno<<endl;}};
class test: public student
{protected:
float sub1,sub2;
public:
void get_marks(float x, float y)
{sub1=x;sub2=y;}
void put_marks()
{cout<<"SUB1: "<<sub1<<endl;
cout<<"SUB2: "<<sub2<<endl;}};
class sports
{protected:
float score;
public:
void get_score(float m)
{score=m;}
void put_score()
{cout<<"score: "<<score<<endl;}};
class result : public test,public sports
{protected:
float totalmark;
public:
void display()
{put_number();
put_marks();
put_score();
totalmark=sub1+sub2;
cout<<"Total is: "<<totalmark<<endl;}};
void main()
{result s; s.get_number(111);
s.get_marks(56.5,75.5);
s.get_score(47); s.display();
getch();}
```

Output:

```
Roll No:111
SUB1:56.5
SUB2:75.5
Score:47
Total is: 132
```

…

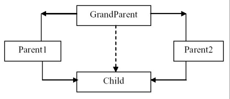
Virtual Base Class:

- Lets us consider an example of hybrid inheritance. Form the figure the child class has two direct base classes Parent1 and Parent2 which themselves have common class GrandParent. The child inherits the character of GrandParent via two separate paths. It can also be inherited directly as shown by the broken line. In the above diagram, all the public and protected members of GrandParent are inherited in the child twice. First via Parent1 and second via Parent2. This means child would have duplicate set of members inherited from GrandParent. This introduces **ambiguity** and should be avoided. The duplication of inherited members due to their multi paths can be avoided by making common base class as **virtual base class.**

…

- Syntax:

```
class GrandParent
{
.......
....... };
class Parent1 : virtual public GrandParent
{
.......
.......
};
class Parent2 : virtual public GrandParent
{
.......
.......
};
class child : public Parent1, public
    Parent2
{
.......
......./*only one copy of
    grandparent will be inherited*/
};
```



…

Abstract Classes

- An abstract class is that class which is not used to create objects. The abstract class is always the base class whose members are derived and used by the object of other child classes. Due to abstract base class no more copy of object should create in program. (a class can only be considered as an abstract class if it has at least one pure virtual function, *about a pure virtual function we study later*)

Constructor in Derived Classes

- The constructor plays an important role in initializing objects. One important thing to note here is that, as long as no base class constructor takes any argument, the derived class need not have a constructor functions. However, if any base 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. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

…

- Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when the derived class object is declared.
- The general form of defining the derived class constructor is:

```
constructor (arglist) : initialization-section
{
assignment-section
}
```

- The assignment-section is nothing but the body of the constructor function and is used to assign initial values to its data members. The part immediately follow the colon is known as the initialization-section. We can use this section to provide initial values to the base constructors and also to initialize its own class members.

…

```
//Example virtual base class &
    constructor
#include<iostream.h>
#include<conio.h>
class student
{protected:
int roll_number;
public:
student(int a)
{roll_number=a;}
void put_number()
{
cout<<"Roll Number
    :"<<roll_number<<endl;}};
class test: virtual public student
{
protected:
float sub1,sub2;
public:
test(int a,float x, float b):student(a)
{
sub1=x;sub2=b;
}
void put_marks()
{
cout<<"Sub1"<<sub1<<endl;
cout<<"Sub2"<<sub2<<endl;}};
```

…

```
class sports:public virtual student
{protected:
float score;
public:
sports(int a,float z):student(a)
{score=z;}
void put_score()
{
cout<<"Score
    Weight"<<score<<endl;}};
class result:public test,public sports
{protected:
float total;
public:
result(int a,float b, float c, float
    d):student(a),test(a,b,c),sports(a,d)
    {}
void display()
{
put_number();
put_marks();
put_score();
total=sub1+sub2+score;
cout<<"Total is "<<total<<endl;
}
};
void main()
{
clrscr();
result s(111,23.4,34.8,7.5);
s.display();
getch();
}
```

Output:

```
Roll Number :111
Sub1 23.4
Sub2 34.8
Score Weight 7.5
Total is 65.7
```

…

Subclass, Subtype & Substitutability

- A class which inherits the features of super class is known as subclass. Subclassing provides a way of constructing new components using existing component and subtyping means subclass is special case of super class.
- type A is said to be a subtype of type B if an instance of type A can be substituted for an instance of B with no observable effect. Subtype is defined in terms of behavior not in terms of structure. If we have two class A & B, A is super class and B is subclass, then the behavior of A can be shared by subclass or child class B. The relation of subclass and subtype is clearly defined from the relationship of data types associated with the parent class to the data types associated with the derived class.

…

- Some relation of subclass and subtypes is given below:

1. Instance of subclass must influence all data areas associated with the parent class
2. Instance of subclass must implement through inheritance at least all functionality defined for parent class and they also can define new functionality.
3. An instance of child class can act as the behavior of the parent class and should indistinguishable from an instance of parent class if substituted in similar situation.

- **Substitutability** means the features of a program in which certain things can substitute in other section or part of a program. The principle of substitutability says that if we have two classes A and B such that class B is subclass of A, then it should be possible to substitute instance of class B for instance of class A in any situation with no observable effect.

…

Forms of Inheritance

Inheritance is used in variety of ways:

1. Subclassing for Specialization (Subtyping)

- The most common use of inheritance and sub classing is for specialization. Here new class is specialized from the parent class but child class should satisfy all the specification given by the parent class. This form of inheritance supports principle of substitutability where the instance can substitute inside a child class. In subclassing for specialization, the subclass can inherits all the behavior of the parent class.

2. Subclassing for Specification

- In this form, the parent class does not implements behavior but only defines the behavior which is implemented by the child class. This class maintains common interface of some methods. The child class can implement the parent class behavior but no interface changes.

…

3. Subclassing for Construction

- The child class can be constructed from parent class by implementing the behavior of parent class. The desired behavior can be collected and can construct the child class. Some arguments and its way of use can also be including in construction of child class.

4. Subclassing for Generalization

- Here, the child class modifies or overrides some of the methods of the parent class. A subclass extends the behavior of the parent class to create a more general kind of object. This forms occurs when the overall design is based on data values primarily and behavior secondarily. Subclassing for generalization is the opposite of subclassing for specification.

…

5. Subclassing for Extension

- The subclassing for extension adds new functionality in the child class from the base class. In new child class, at least one method is inherited and the functionality is tied to that parent class. Extension simply adds new method to those of the parent class and the functionality is tied to the existing methods to the parent class.

6. Subclassing for Limitation

- Subclassing for limitation occurs when the behavior of subclass is smaller or more restricted than the behavior of parent class. It occurs when child class cannot or should not change the behavior of parent class.

7. Subclassing for Variance

- This form is same as hierarchical inheritance in which more child class can inherit varieties of behavior and different child class can inherit the behavior and implement in their own way.

8. Subclassing for Combination

- Here, features can combine from more than one class can inherits features from more than one base class.

…

Advantages of Inheritance

1. Software Reusability

- When behavior is inherited from another class, the code that provides that behavior does not have to be written. Many programmers spend much of their time rewriting code, they have written many times before. With object oriented techniques, these functions can be written once and reused. Other benefits and reusable code include increased reliability.

2. Code Sharing

- Code sharing can occur on several levels with object oriented techniques. On one level, many users or projects can use the same class. Another form of sharing occurs when two or more classes are developed by single programmer as a part of project inherits from a single parent class. When this happen two or more types of objects will share the code that they appear.

…

3. Consistency of Inheritance

- When two or more class inherits from the same super class, we are assured that the behavior they inherit will be the same in all cases. Thus, it is easier to guarantee that interfaces to similar objects are in fact similar and that the user is not presented with a confusing collection of objects that are almost the same but behave and are interacted with very difficulty.

4. Software Component

- Inheritance provides programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel application that nevertheless requires little or no actual coding. Already, several such libraries are commercially available and we can expect many more specialized systems to appear in time.

…

5. Rapid Prototyping

- When a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system. Thus, software system can be generated more quickly and easily, leading to a style of programming known as rapid prototyping. A prototype system is developed, users experiment with it, a second system is produced that is based on experience with the first, and further experimentation takes place and so on for several iteration. Such programming is particularly useful in situations where the goal and requirements of the system are only vaguely understood when the project begins.

…

6. Polymorphism & framework

- Software produced conventionally is generally written from the bottom up, although it may be designed from top down. That is, the lower-level routines are written and on top of these slightly higher abstractions are produced and on top of these even more abstract elements are generated.
- Normally, code portability decreases as one move up the levels of abstraction. That is, the lower-level routines may be used in different projects and perhaps even the next level of abstraction may be reused but the higher level routines are intimately tied to a particular application. The lower level pieces can be carried to a new system.
- Polymorphism is programming language permits the programmer to generate high level reusable components that can be tailored to fit different applications by changes in their low level parts.

112

…

7. Information Hiding

- A programmer who reuses software component needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques to implement the components. Thus, the interconnectedness between software systems is reduced.

113

…

Disadvantage of Inheritance:

1. Execution Speed

- It is seldom possible for general purpose software tools to be as fast as carefully hand-crafted systems. Thus, inherited methods which must deal arbitrary sub-classes are often slower than specialized code.

2. Program Size

- The use of any software library frequently imposes a size penalty not imposed by systems constructed for a specific project. Although this expense may be substantial, as memory costs decrease, the size of programs becomes less important. Containing development costs and producing high quality and error free code rapidly are now more important than limiting the size of program.

114

…

3. Message Passing Overhead

- Must have been made of the fact that message passing is by nature, a more costly operation than simple procedure invocation. As with overall execution speed, however, over concern about the cost of message passing is frequently penny-wise and pound foolish. Timing figures varies from language to language. The overhead of the message passing will be much higher in dynamically bound languages, and statistically bound languages. The increased cost like other must be weighed against the many benefits of the object oriented technique.
- A few languages make a number of options available to the programmer that can reduce message passing overhead. These include eliminating the polymorphism from message passing and expanding inline procedure. Dynamic methods are inherently slower but require less space.

115

…

4. Program Complexity

- Although object oriented programming is often touted as a solution to software complexity, in fact, over use of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. This is known as yo-yo problem.

Inheritance and Substitutability

- The principle of substitutability referred to the relation between a variable declared as one class and a value derived from another class. We can assign the value of variable if class of variable is same as subclass of variable and the concept of substitutability is provided form inheritance.

116

…

```
//Example of substitutability
#include<iostream>
#include<conio.h>
class A
{
protected:
int a;
public:
void get_a()
{
a=10;
cout<<"a="<<a<<endl;
}
};
class B : public A
{
};
void main()
{
A a;
B b;
a.get_a();
b.get_a();
getch();}
```

Output:

```
a=50
a=50
```

117

…

IS a Rule and HAS a Rule

- There are two relationships between the components of the system. By knowing the relationship we can easily understand and apply object oriented software reuse technique.
- IS a relationship says that the first component is a specialized instance of second component or concept. The data and behavior with more specific idea forms a subset of a behavior in more abstract idea. The example of IS a relationship are; Dog is a mammal, Florist is a shopkeeper, Ram is Boy, etc. The relation derives its name from a simple rule that test the relationship between two concepts. In real life application if two components or objects have relation then we can apply it in programming.
- HAS a relationship occurs when second component is a component of first. For example, Car has a engine, Dog has a tail, etc. By knowing HAS a relationship, we can further extend the components.

…

Composition and Inheritance

- We know that object is simply an encapsulation of data and behavior. When the concept of composition is applied to reuse the data values in the development of new data types, the portion of state of new data structure is simply the instance of existing structure. The composition clearly indicates what operations can be performed on a particular data structures.
- The technique of including user defined objects type as a part newly defined objects is called composition.
- In such case super class object can be designed into derived class directly and called respective members using syntax:

class A {} class B {visibility mode: A obj;};

```
//Example of composition
#include<iostream>
#include<conio.h>
class A{int l,b;
public:
void getarea(int x, int y)
{
l=x; b=y;
cout<<"Area="<<l*b<<endl;}};
```

…

```
class B
{
private:
A a;
public:
void displayarea();
};
void B::displayarea()
{
a.getarea(10,20);
}
```

```
void main()
{
B b;
b.displayarea();
getch();
}
```

Output:

```
Area=200
```

…

Comparison Between Composition and Inheritance

i) Composition is simple than inheritance, composition indicates the operation but in inheritance the operation is the superset of existing operation when the object is created.

ii) In inheritance, we directly reuse the code and functionality provided by the parent class. But in composition the code becomes shorter but provides greater functionality.

iii) Inheritance does not prevent user from manipulating new structures using methods from parent class. But composition prevents for new data structure.

iv) Data structure using inheritance reduces in very small extent over construction with composition, since additional functionalities are avoid to use.

v) In inheritance, the programmer needs to know the upper class but it is not necessary to understand the parent class. In composition since it allows to use only usable components from the parent class.

Assignments

11. What does Inheritance mean in C++? Explain the disadvantages of inheritance.
12. What are the forms of inheritance? Describe them briefly.
13. What do you mean by virtual base class? At which condition it has to be implemented? Explain it with suitable example.
14. Explain the principle of substitutability. How does inheritance provide the concept of reusability? How do you define reusability of components?
15. Develop a complete program for an institution which wishes to maintain a database of its staff. Declare a base class STAFF which include staff_id and name. Now develop records for the following staffs with the given information below:

    Lecturer (subject, department)

    Administrative staff (post, department)

    Each staff members should inherit staff_id and name from base class.

Assignments…

16. Explain on different derivation mode, that a sub class can used to inherit base class in C++.
17. Compare and contrast IS-A rule and HAS-A rule.
18. Write short notes on:

    a) subtype b) abstract class c) composition

    d) ambiguity arises in multiple inheritance e) constructor in inheritance