Object Oriented Programming

Unit 4: Inheritance

Inheritance

4.1. Advantage and Disadvantage of Inheritance, Derived class declaration

4.2. Accessibility of Base Class Members

4.3. Types of Inheritance

4.4. Virtual Base Class

4.5. Order of Calling of Constructor and Destructor

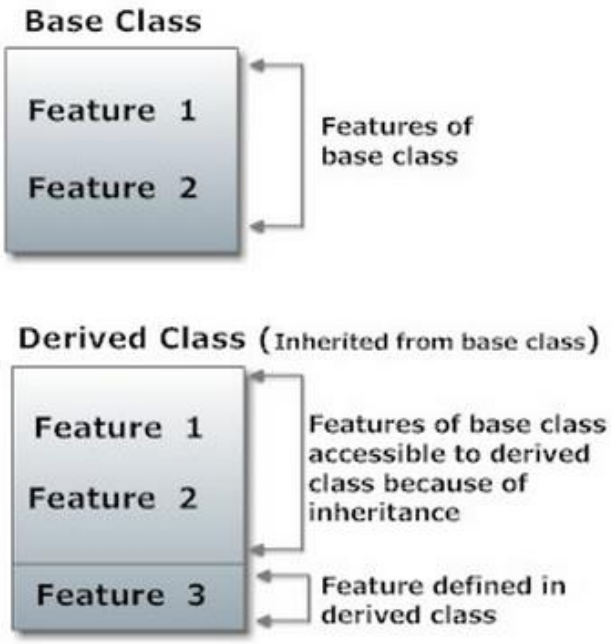
4.6. Function Overriding

4.7. Object Composition – Delegation

CE: 4.0. What is Inheritance?

- Inheritance is one of the key feature of object-oriented programming including C++ which allows user to create a new class (derived class) from a existing class (base class).

- The derived class inherits all the feature form a base class and it can have additional features of its own.



CE: 4.0. Inheritance (Conti…)

- Inheritance allows the **reusability** of existing code without writing the code again from the scratch.
- It would not only save time and memory, but also reduces the frustration and increases reliability (dependability).
- So by reusing the class that is already been tested and debugged, it also save much time and effort of developing the same thing again and again.
- Thus, C++ strongly supports the concept of reusability in several ways.

The mechanism of deriving a new class from an old one is called "inheritance".

CE: 4.0. Inheritance (Conti…)

- A class can also inherit properties from more than one base class or from more than one level.

CE: 4.1.1
Advantage and Disadvantage of Inheritance

CE: 4.1.1. Advantages of Inheritance

1. It helps in **reusability** of the code.
   [The codes are defined only once and can be used multiple times. In java we define the super class or base class in which we define the functionalities and any number of child classes can use the functionalities at any time.]

2. It also **saves lots of time and efforts**.

3. It **improves the program structure**, so that the program can be in **readable form**.

4. The **program structure will be short**, so that it will be more **reliable**.

5. The codes are **easy to debug**. Because it allows the program to capture the bugs easily.

CE: 4.1.1. Advantages of Inheritance (Conti…)

6. It makes the **application code more flexible to change**.

7. It **results code is better organized** into smaller, simpler and simpler compilation units.

8. The base class can decide to keep some data private, so that it cannot be altered by the derived class.

9. With inheritance, we will be able to override the methods of the base class, so that meaningful implementation of the base class method can be designed in the derived class.
   [Method Overriding/ Runtime Polymorphism]

CE: 4.1.1. Disadvantages of Inheritance

1. The two classes (i.e. base class and derive class) are tightly coupled, so that they are dependent on each other.

2. If the functionality of the base class is changed, then it is also needed to be change at child classes.

3. If the methods in the base or super class are deleted, then it is very difficult to maintain the functionality of the child class.
   [Because the functionality must be implementing the super class(s) methods.]

4. It also increases the time and efforts, that takes to jump through different levels of the inheritance.

International Certification Question

Q1.	**The method by which objects of one class get the properties of objects of another class is known as?**
A.	**Abstraction**
B.	**Encapsulation**
C.	**Inheritance**
D.	**Polymorphism**

Ans: C

International Certification Question

Q2.	**The major goal of inheritance in C++ is…..**
A.	**To facilitate the data type conversion**
B.	**To help modular programming**
C.	**To hide the detail of base class**
D.	**To extend the capabilities of base class**

Ans: D

International Certification Question

Q3.	**Advantages of inheritance includes….**
A.	**providing class growth through natural selection**
B.	**facilitating class libraries**
C.	**avoiding the rewriting of code**
D.	**None of above**

Ans: C

International Certification Question

Q4.	**Which of the following is/are false?**
A.	**Inheritance is deriving new class from existing class.**
B.	**In an inheritance, all data and function members of base class are derived by derived class.**
C.	**We can specify which data and function members of base class will be inherited by derived class.**
D.	**We can add new functions to derived class without recompiling the base class.**

Ans: B

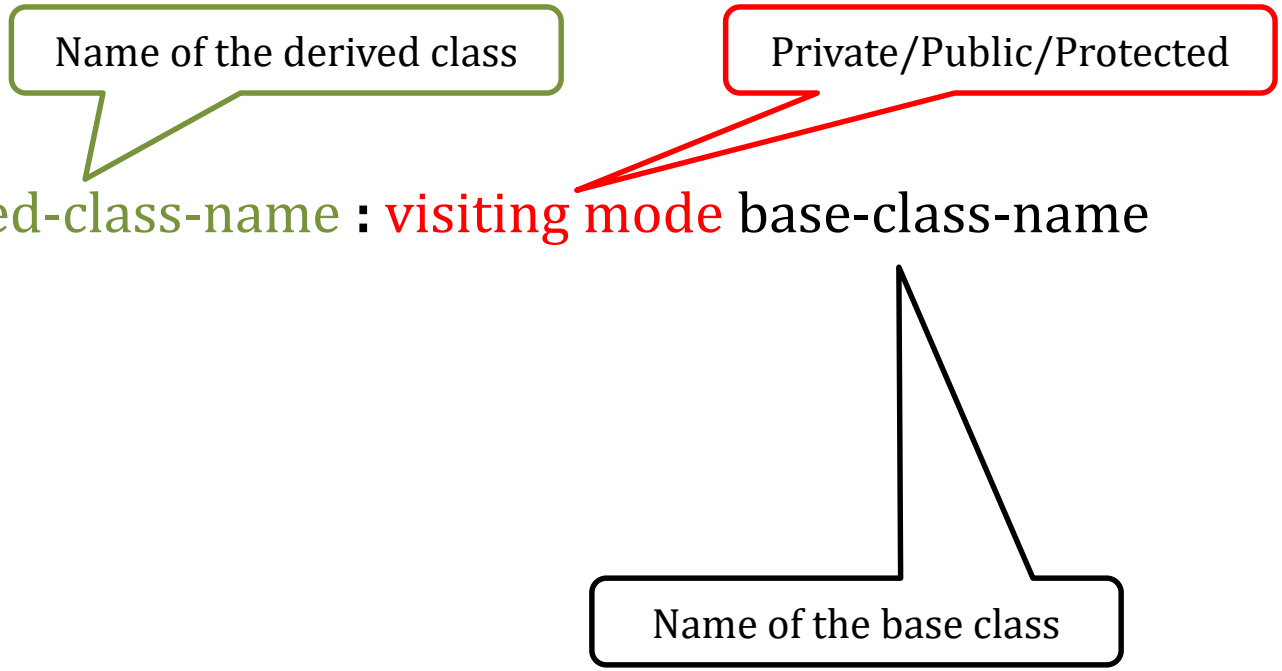
International Certification Question

Q5.	C++ strongly support the concept of …
A.	Reusability
B.	Availability
C.	Maintainability
D.	Readability

Ans: A

CE: 4.1.2. Derived class declaration

- Syntax:



CE: Visibility Modes

- The visibility mode (private / public / protected) in the definition of the derived class specifies, whether the features of the base class are *privately* derived or *publicly* derived or *protected* derived.

- The visibility modes basically control the access-specifier to be for inheritable members of base-class, in the derived class.

Visibility Mode is	Inheritable public member becomes (in derived class)	Inheritable protected member becomes (in derived class)	Private Member of base class are not directly accessible to derived class.
Public	Public	Protected	
Protected	Protected	Protected	
Private	Private	Private	

Role of visibility modes

CE: Possibilities of Derivation

Public Derivation

```
class D: public B
    {
        //members of D
    }
```

Private Derivation by default

by default Derivation???

```
class D: B
    {
        //members of D
    }
```

```
class D: private B
    {
        //members of D
    }
```

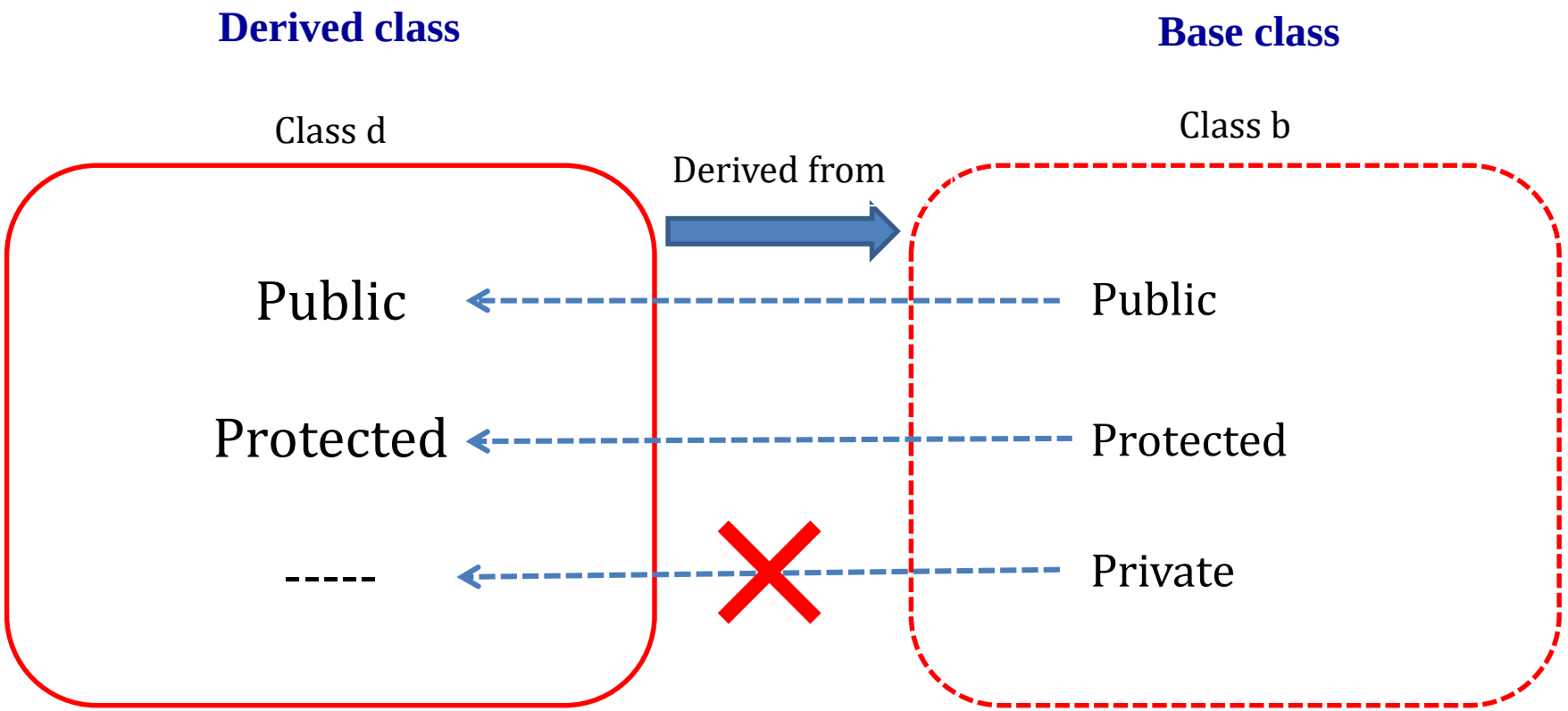
Private Derivation

```
class D: protected B
    {
        //members of d
    }
```

Protected Derivation

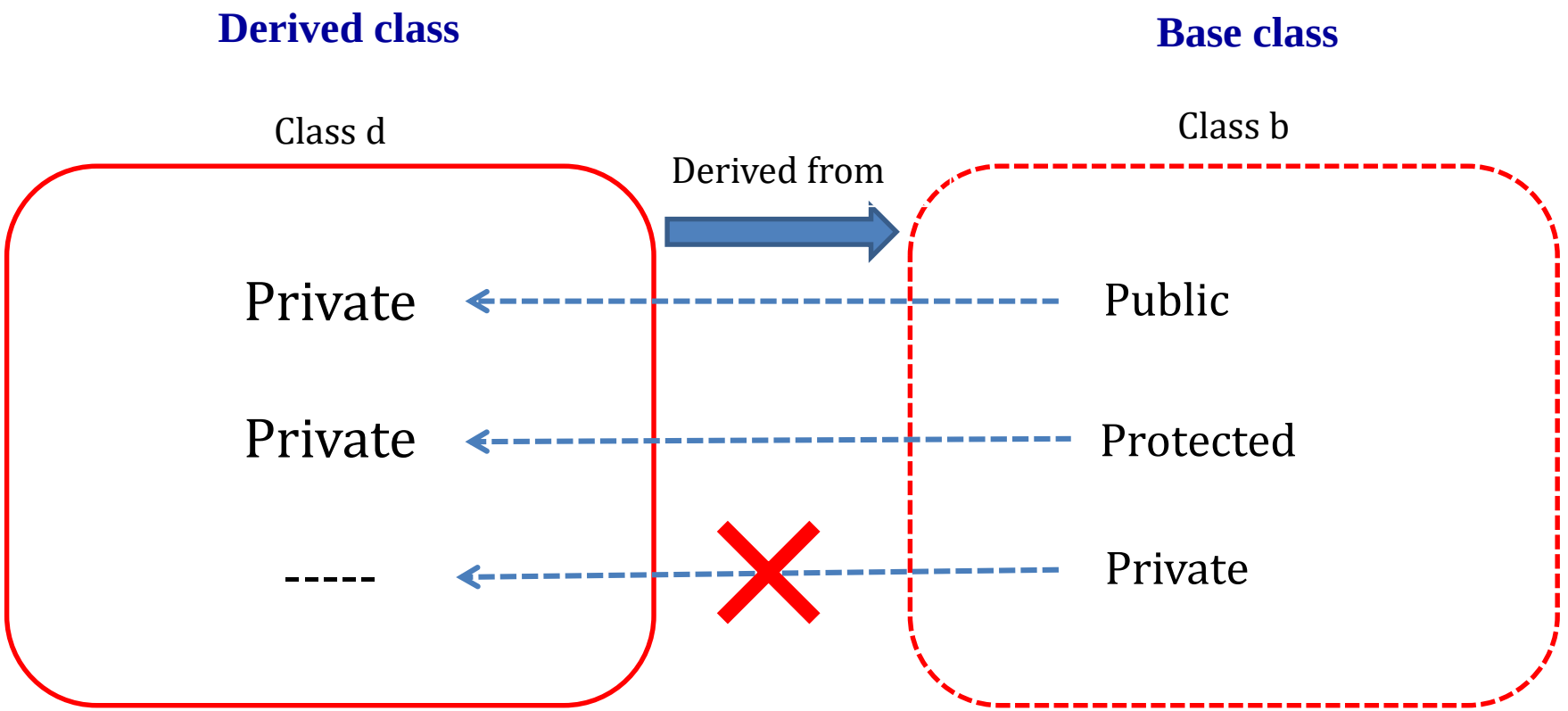
CE: Public Derivation

- class d : **public** b



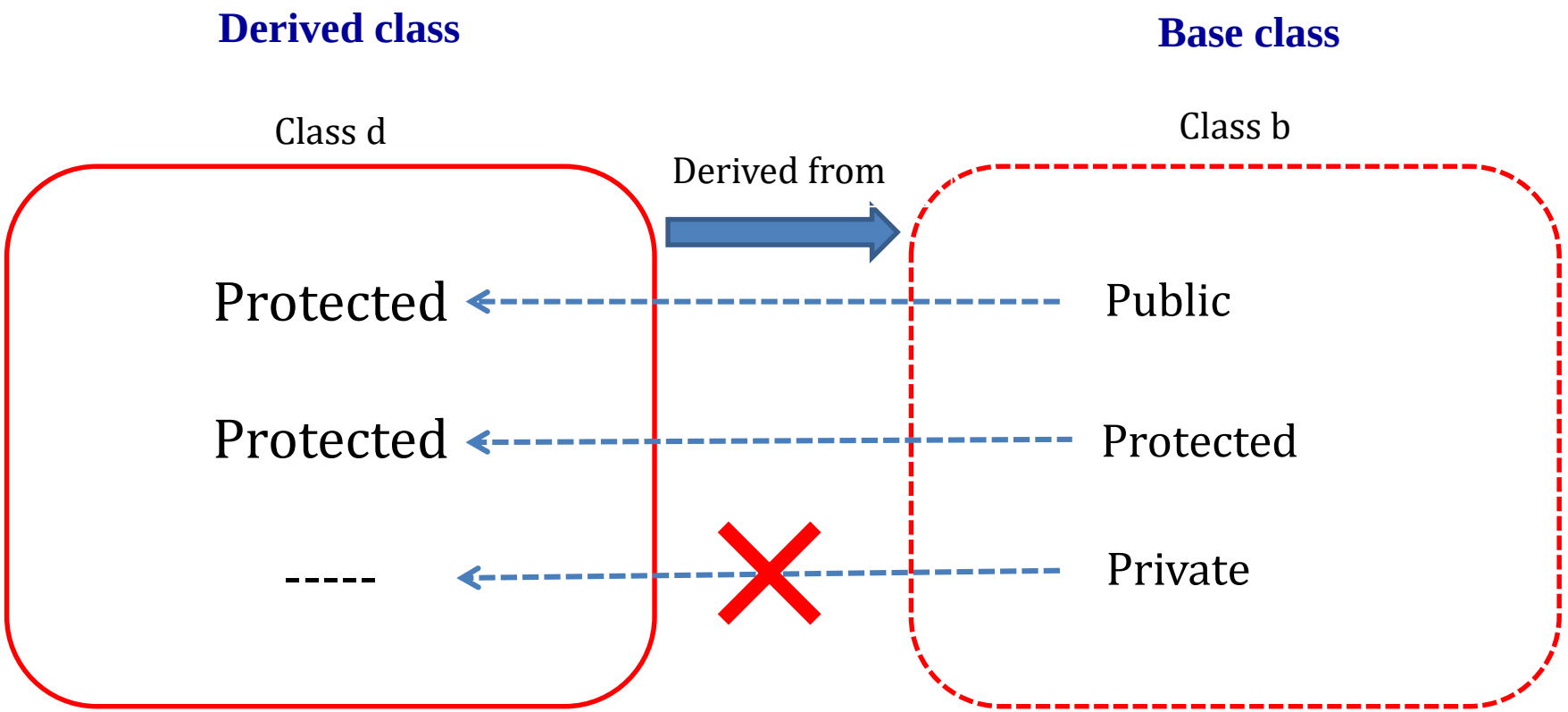
CE: Private Derivation

- class d : **private** b



CE: Protected Derivation

- class d : **protected** b



Demonstration: Visibility Modes

```
class base
{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
};

class publicDerived: public base
{
        // x is public
        // y is protected
        // z is not accessible from publicDerived
};
```

Demonstration: Visibility Modes (Conti…)

```
class protectedDerived: protected base
{
        // x is protected
        // y is protected
        // z is not accessible from protectedDerived
};

class privateDerived: private base
{
        // x is private
        // y is private
        // z is not accessible from privateDerived
}
```

CE: 4.2
Accessibility of Base Class Members

CE: 4.2. Accessibility of Base Class Members

- Since in an inheritance hierarchy, the derived classes inherit from base classes, the base class can be said to be in a position to play an important role as far as inheritance of its members is concerned.

- While defining a base class, following things should be kept in mind:
  1. The Data Members, Member Functions and even Non-Member Functions that are planned to be inherited, should be declared as **public** members in the base class.

  2. The Members which is to be inherited, but not proposed to be **public**, that should be declared as **protected**.

  3. The Members which is not to be inherited, that should be declared as **private**.

CE: 4.2. Accessibility of Base Class Members (Conti…)

Access Specifier	Accessible from own class	Accessible from derived class (Inheritable)	Accessible from objects outside class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
private	Yes	No	No

Accessibility of Base Class Members

- Therefore, the **private** and **protected** members of a class can only be accessed by the member functions (and friends) of the **same class.**

- However, **protected** members can be inherited, whereas, **private** members cannot be inherited.

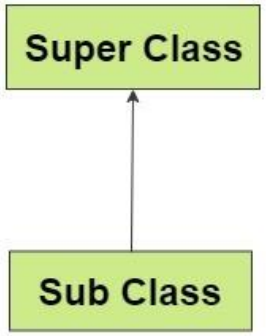
CE: 4.3
Types of Inheritance

CE: 4.3. Types of Inheritance

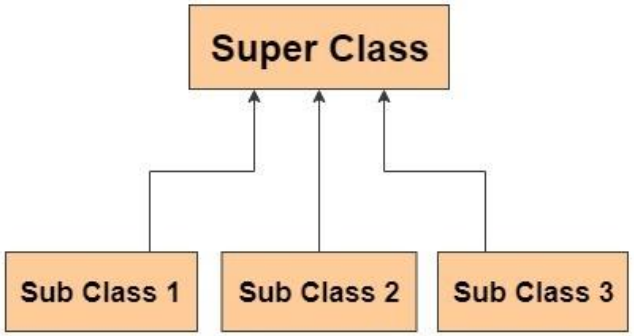
- Inheritance may take place in many forms:
  1. Single Inheritance
  2. Multiple Inheritance
  3. Hierarchical Inheritance
  4. Multilevel Inheritance
  5. Hybrid Inheritance

CE: 4.3. Types of Inheritance

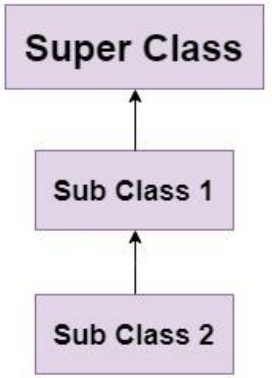
Single Inheritance



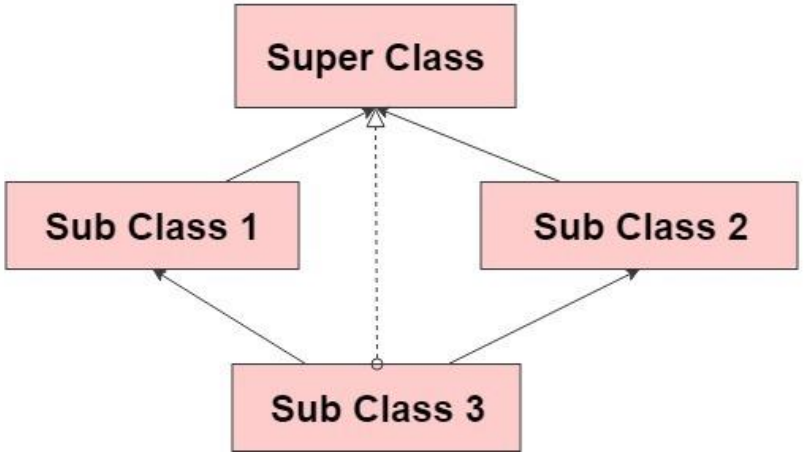
Hierarchial Inheritance



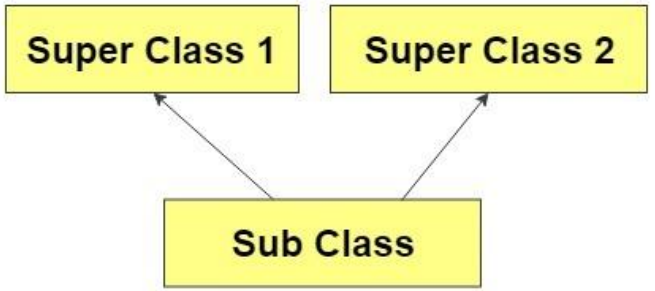
MultiLevel Inheritance



Hybrid Inheritance



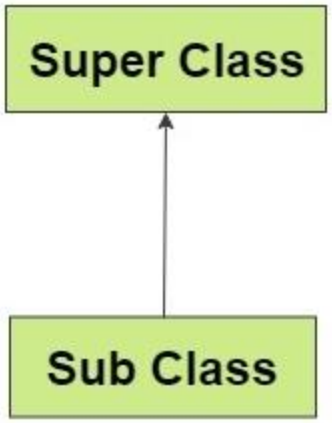
Multiple Inhertance



CE: 4.3. Types of Inheritance (Conti…)

1. Single Inheritance:

- When a sub-class inherits only from one base class, it is known as single inheritance.



CE: Single Inheritance

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

class person
  {
            protected:
                    int id;
                    char name[30];
            public:
                   void getinfo()
                       {
                              cout<<"Enter your ID: ";
                              cin>>id;
                              cout<<"Enter your name: ";
                              cin>>name;
                       }
  };
```

CE: Single Inheritance (Conti…)

```
class student:public person
  {          private:
                  int  marks[5];
                  int sum;
                  float  per;
             public:
             void getresult()
                 {
                       int i;
                       for(i=0; i<=2;i++)
                           {
                                 cout<<"Mark["<<i+1<<"]: ";
                                 cin>> marks[i];
                           }
                       for(i=0; i<=2;i++)
                           {     sum += marks[i];     }
                       per = sum/3;
                  }
```

CE: Single Inheritance (Conti…)

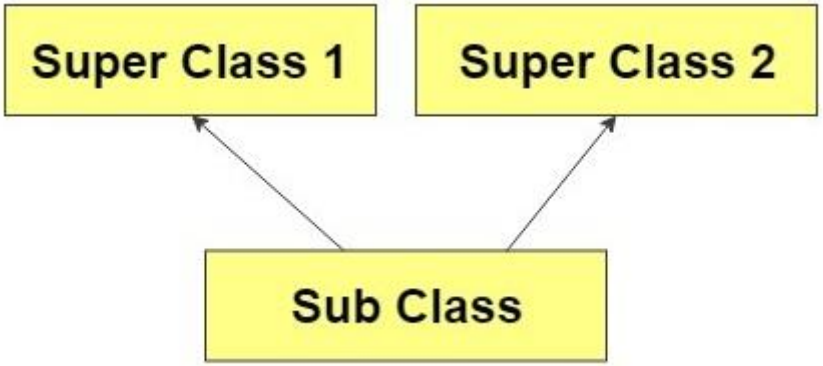
```
            void displayinfo()
                {
                        cout<<"\nID: "<<id;
                        cout<<"\nName: "<<name;
                        cout<<"\nPercentage = "<<per;
                }
  };
int main()
  {
        student p;    //create object of derived class
        cout<<"\n************** Enter Details ******************\n";
        p.getinfo();
        p.getresult();
        cout<<"\n************** Output ******************";
        p.displayinfo();
        cout<<"\n*************************************************";
        return 0;
  }
```

simplein.cpp

CE: 4.3. Types of Inheritance (Conti…)

2. Multiple Inheritance:

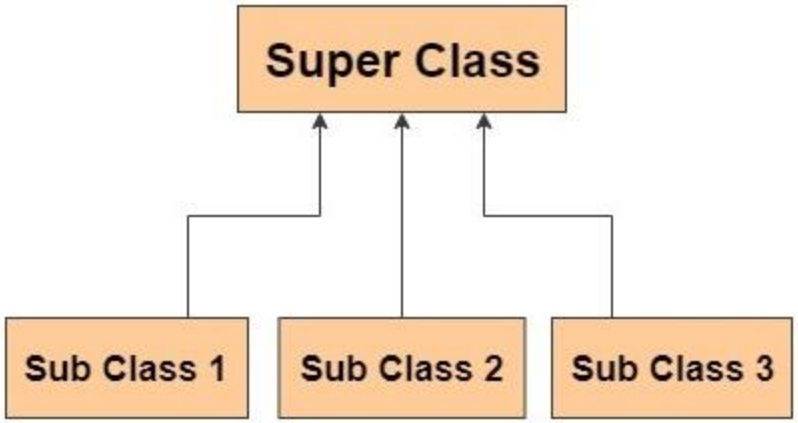
- When a sub-class inherits only from multiple base classes, it is known as multiple inheritance.



CE: 4.3. Types of Inheritance (Conti…)

3. Hierarchical Inheritance:

- When many sub-classes inherits from a single base class, it is known as hierarchical inheritance.

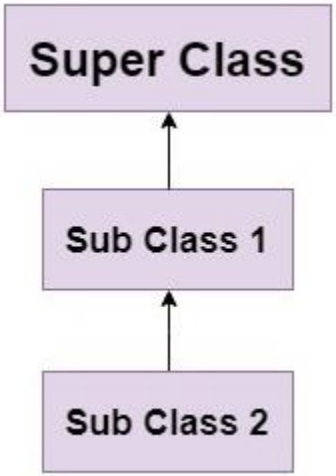


CE: 4.3. Types of Inheritance (Conti…)

4. Multilevel Inheritance:

- The transitive nature of inheritance is reflected by this form of inheritance.

- When a sub-class inherits from a class that itself inherits from another class, it is known as multilevel inheritance.

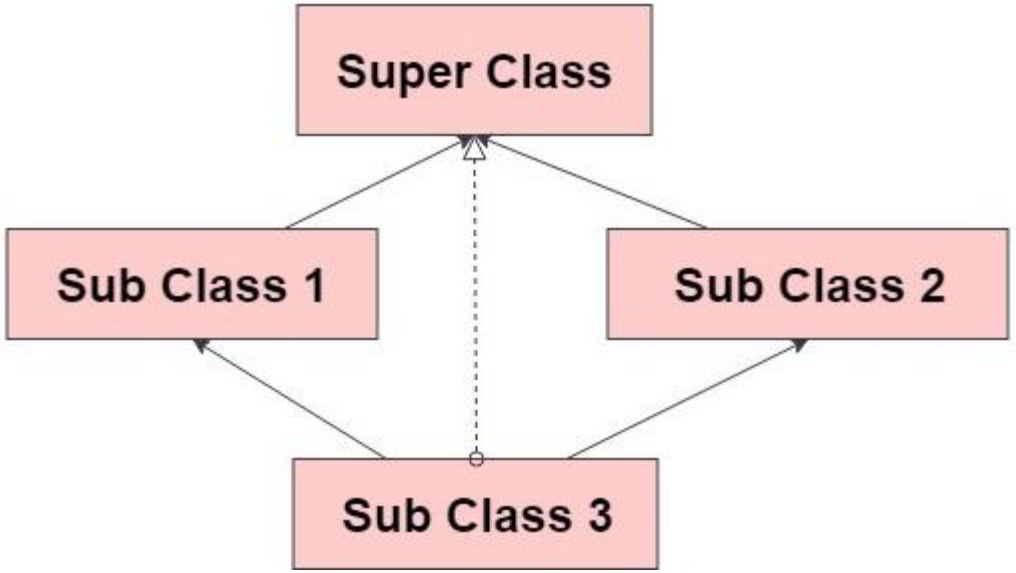


CE: 4.3. Types of Inheritance (Conti…)

5. Hybrid Inheritance:

- When a sub-class inherits from multiple base classes and all of its base classes inherits from a single base class, this form of inheritance is known as hybrid inheritance.

- Hybrid inheritance combines two or more forms of inheritance.



Class Work

Write a C++ program by creating a class CRICKET which has data members name and total_match. Derive a BATSMAN class from CRICKET which has data members runs, average run and best performance. Create appropriate methods for reading and displaying the cricketer's details, and for calculating average run.

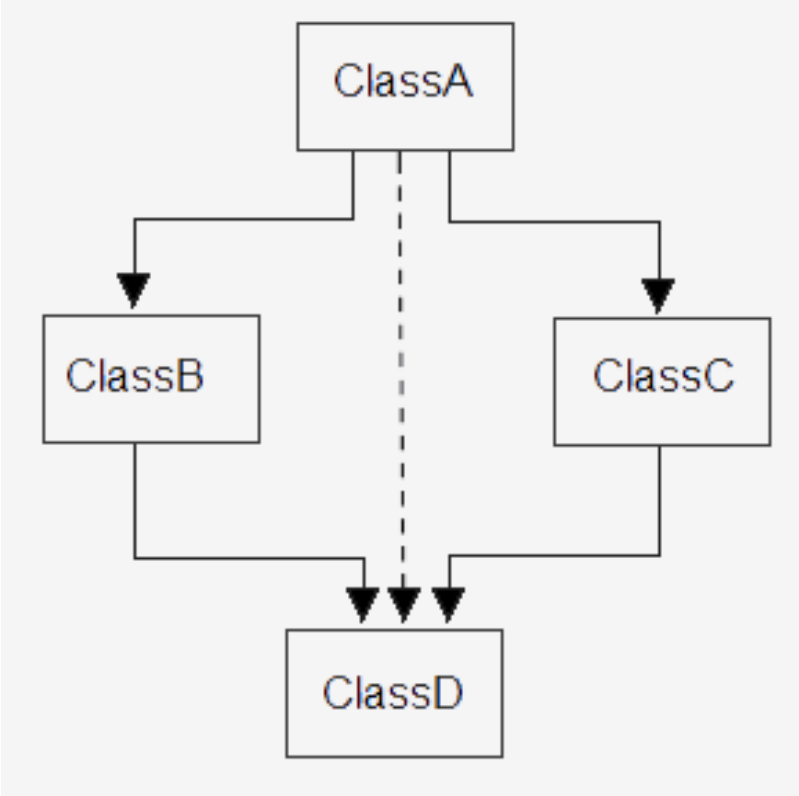
Practical Work

Create a class EMPLOYEE having data members EmpNo(type int), Name(type char) and Designation(type char) and member function getdetail() which will ask for the employee details.
Derive a class SALARY from EMPLOYEE having visibility mode “public”, data members basic_pay and PF; member function get() to ask for the basic_pay and PF, calculate function to give the net_pay.(NP=BP-PF) and display function to display all the details of Employee.

CE: 4.4
Virtual Base Class

CE: 4.4. Virtual Base Class

- Virtual base class is used in situation where a derived have multiple copies of base class.



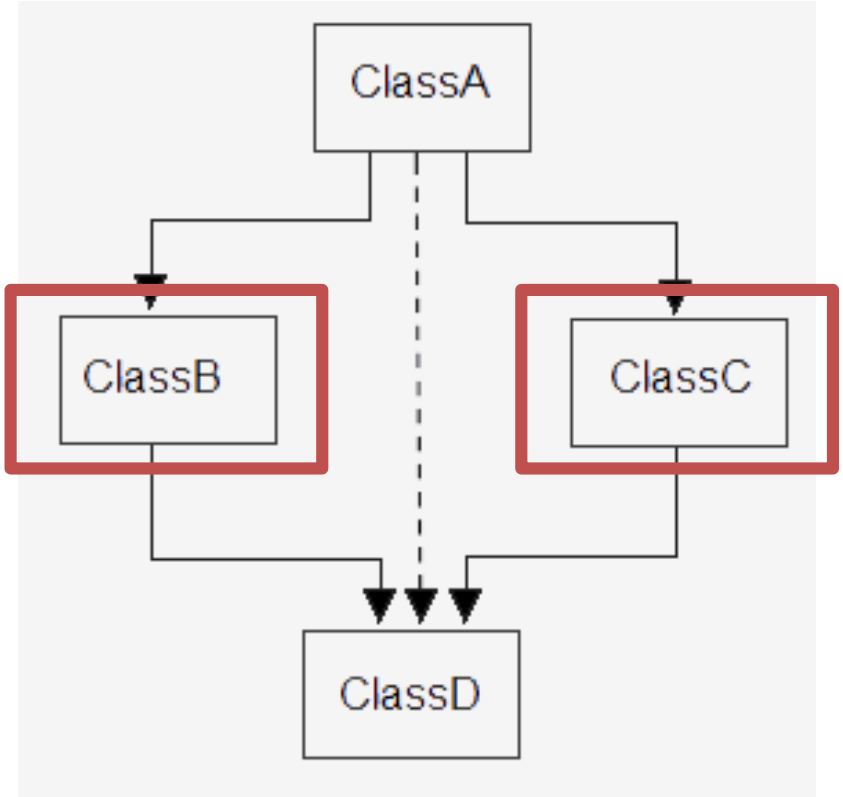
CE: 4.4. Virtual Base Class (Conti…)

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

class ClassA
  {
        public:
              int a;
  };

class ClassB: public ClassA
  {
        public:
              int b;
  };
class ClassC: public ClassA
  {
        public:
              int c;
  };
class ClassD: public ClassB, public ClassC
  {
        public:
              int d;
  };
int main()
  {
        ClassD d;
        d.a = 100;      //Error will occur
        d.a = 200;      //Error will occur

        d.b = 300;
        d.c = 400;
        d.d = 500;

        cout<< "\n A : "<< d.a;
        cout<< "\n B : "<< d.b;
        cout<< "\n C : "<< d.c;
        cout<< "\n D : "<< d.d;
        return 0;       }
```

CE: 4.4. Virtual Base Class (Conti…)

- Here, both **ClassB** & **ClassC** inherit **ClassA**, they both have single copy of **ClassA**.
- However, **ClassD** inherit both **ClassB** & **ClassC**, so **ClassD** have two copies of **ClassA**, one from **ClassB** and another from **ClassC**.
- The statements **d.a = 10;** and **d.a = 100;** are *inherently ambiguous* and will generate error, because the compiler can't differentiate between two copies of **ClassA** in **ClassD**.
- To remove multiple copies of **ClassA** from **ClassD**, we must inherit **ClassA** in **ClassB** and **ClassC** as **virtual class**.

CE: 4.4. Virtual Base Class (Conti…)



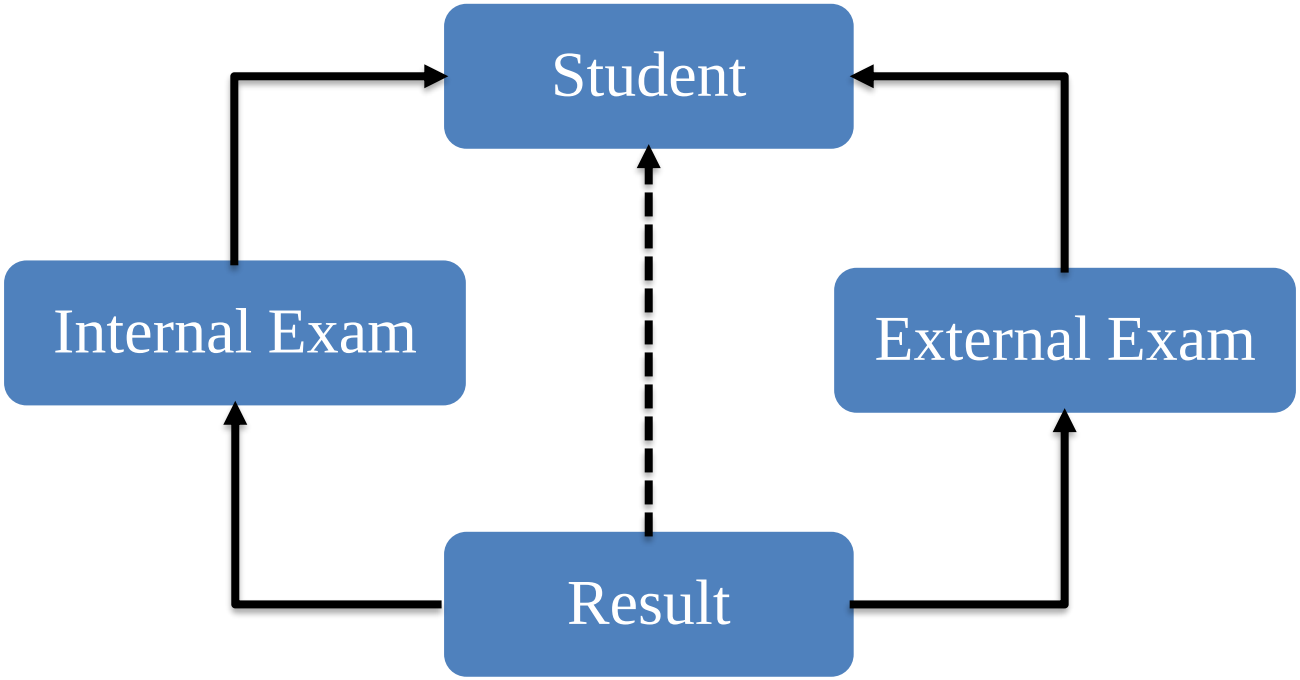
CE: 4.4. Virtual Base Class (Conti…)

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

class ClassA
  {
        public:
              int a;
  };

class ClassB: virtual public ClassA
  {
        public:
              int b;
  };
class ClassC: virtual public ClassA
  {
        public:
              int c;
  };
class ClassD: public ClassB, public ClassC
  {
        public:
              int d;
  };
int main()
  {
        ClassD d;
        d.a = 100;
        d.a = 200;

        d.b = 300;
        d.c = 400;
        d.d = 500;

        cout<< "\n A : "<< d.a;
        cout<< "\n B : "<< d.b;
        cout<< "\n C : "<< d.c;
        cout<< "\n D : "<< d.d;
        return 0;       }
```

virtualbaseclass.cpp

Home Work

Write a C++ program to perform the following:



Consider data members for STUDENT as rollno, branch, for INTERNAL and EXTERNAL exam take five subjects marks and for RESULT take total. Take appropriate methods and complete the task.

CE: 4.5
Order of Calling of Constructor and Destructor

CE: 4.5. Order of Calling of Constructor and Destructor

- The constructors play an important role in initialing objects.
- One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class also need not to have a constructor function.
- If a base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have constructor and pass the arguments to the base class constructors.
- This is because constructors can’t be inherited, but though a derived class it can be explicitly call the constructor and destructor of the base class.

CE: 4.5. Order of Calling of Constructor and Destructor (Conti…)

- The derived class takes the responsibility of supplying initial values to its base classes, so all the initial values that are required by all the classes together are passed to the derived class constructor.

- The constructor of the base class receives the entire list of values as its arguments' and passes them on to the base constructors in the order in which they are declared in the derived class.

- The base class constructors are called and executed before executing the statements in the body of the derived constructor.

CE: 4.5.1. Constructor in multiple inheritance

- The general form of defining a constructor of derived class:

```
Derived-constructor (arg_list1, arg_list2, …., arg_listN, arg_listD)
                : base1(arg_list1), base2(arg_list2), …., baseN(arg_listN)
    {
                //body of derived constructor arg_listD is used
    }
```

  - **arg_listD** is for derived constructor

- **In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.**

CE: 4.5.1. Constructor in multiple inheritance (Conti…)

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

class Base1
  {
        protected:
            int a;
        public:
            Base1(int x)
              {
                    a = x;
                    cout<<"\nBase1 constructor called!";
               }
             ~Base1()
              {
                    cout<<"\nBase1 destructor called!";
               }

  };
```

CE: 4.5.1. Constructor in multiple inheritance (Conti…)

```
class Base2
  {
        protected:
            int b;
        public:
            Base2(int y)
              {
                    b = y;
                    cout<<"\nBase2 constructor called!";
               }
             ~Base2()
              {
                    cout<<"\nBase2 destructor called!";
               }

  };
```

CE: 4.5.1. Constructor in multiple inheritance (Conti…)

```
class Derived: public Base2, public Base1
  {
        int c;
      public:
        Derived(int i, int j, int k): Base1(i), Base2(j) // Explicit call
          {
              c = k;
              cout<<"\nDerived constructor called!";
          }
        ~Derived()
          {
              cout<<"\nDerived destructor called!";
          }
        void display()
          {
              cout<<"\n\nFirst: "<<a;
              cout<<" Second: "<<b;
              cout<<" Third: "<<c<<"\n";
          }
  };
```

```
int main()
  {
        Derived d(10,20,30);
        d.display();
        return 0;
  }
```

multiplecd.cpp

To be continue….