OOP
(UNIT - II)

Unit II - Inheritance and Polymorphism
Base class, Derived class, public, private & protected keywords, Types of inheritance, Ambiguity in multiple inheritance, Classes within classes, Polymorphism concept,Types of polymorphism,function overloading, operator overloading, Unary operator overloading & Binary operator overloading

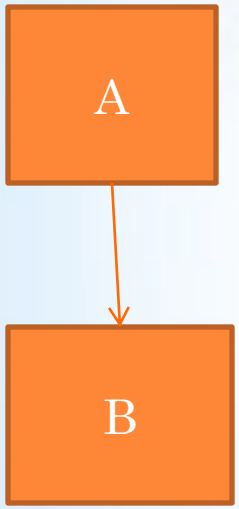
Contents

- Inheritance
  - ✓ Base Class and derived Class
  - ✓ protected members
  - ✓ relationship between base Class and derived Class
  - ✓ Constructor and destructor in Derived Class
  - ✓ Overriding Member Functions
  - ✓ Class Hierarchies, Inheritance
  - ✓ Public and Private Inheritance
  - ✓ Levels of Inheritance
  - ✓ Multiple Inheritance
  - ✓ Ambiguity in Multiple Inheritance
  - ✓ Aggregation
  - ✓ Classes Within Classes

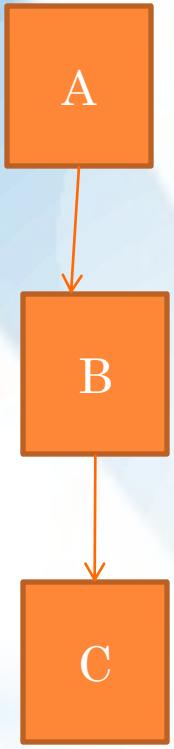
Need and concept

- Reusability is another important feature of OOP
- Using the features(data and/or functions) of one class into another class
- Mechanism of deriving a new class from an old one is called as inheritance(or derivation)
- The old class is referred as Base(super) class and new class is called as derived(Sub) class

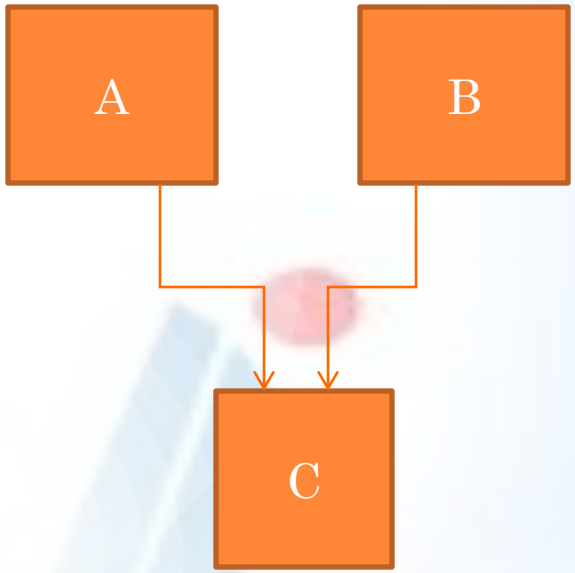
Types of inheritance



Single inheritance

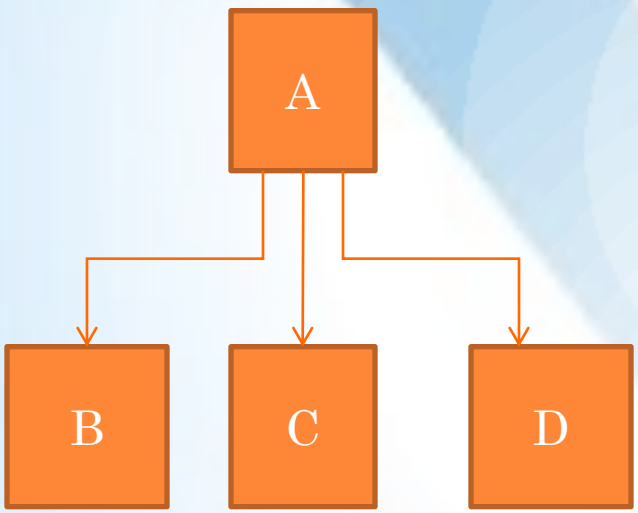


Multi level inheritance

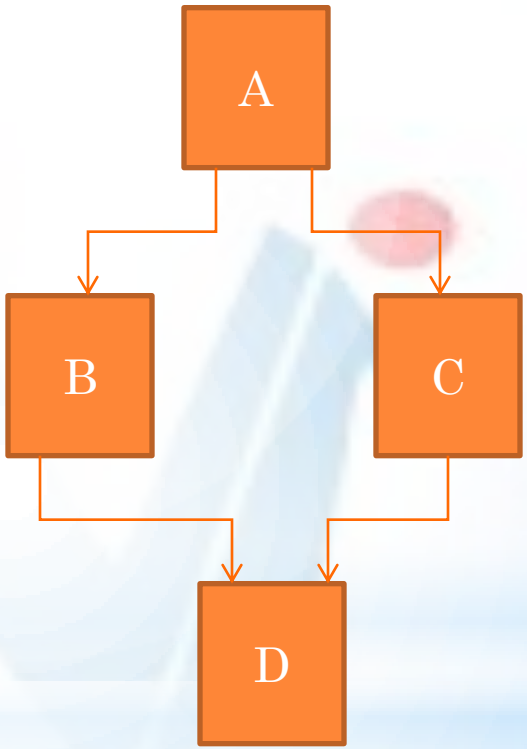


Multiple inheritance

Types of inheritance(cntd…)



Hierarchical inheritance



Hybrid inheritance

Derived class definition

- General form can written as :

```
class derived-class-name : visibility-mode base-class-name
{
   ……//
   ……//
   ……//
};
```

Colon indicates derivation, by default it is private

Example

```
class ABC : private XYZ  // private derivation
{
        // members of ABC
};

class ABC : public XYZ  // public derivation
{
        // members of ABC
};

class ABC :  XYZ  // private derivation by default
{
        // members of ABC
};
```

Private and public derivation

Private derivation

- Public members of base → private of derived
  - So public members of base(now private of derived) can be accessed by member functions of derived
  - Private members are not inheritable

Public derivation

- public members of base → public of derived
  - Private members are not inheritable

Single inheritance : public derivation

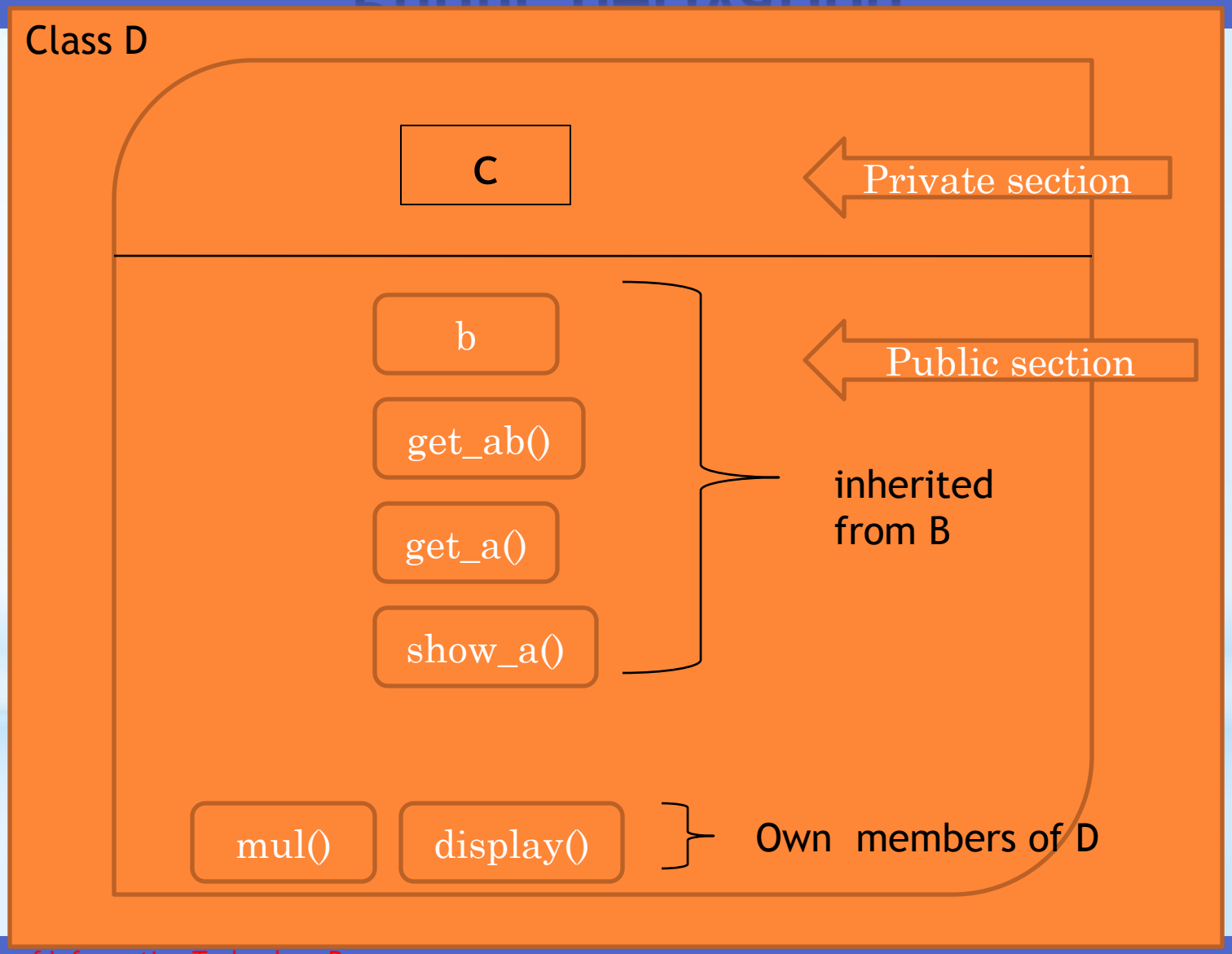
```
class  B
{
    int a; // private, not inheritable
public :   // public, ready for inheritance
    int b;
    void get_ab();
    int get_a(void);
    void show_a();
};
 class   D : public  B // public derivation

 {
      int c;
  public:
      void mul(void);
       void display(void);
 };
```

```
void  B :: get_ab(){
a= 5; b = 10;
}

int  B :: get_a(){
    return a;
}

Void  B :: show_a(){

Cout<<"a = "<<a<<"\n";
}

void  D :: mul(){

c = b * get_a();

}
```

```
void D :: display(){

    cout<<" a = "<<get_a()<<"\n";
    cout<< "b = "<<b<<"\n";
    cout<< "c = "<<c<<"\n";

}
```

```
int main() {

        D d;
        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b=20;
        d.mul();
        d.display();

        return 0;
}
```

Public derivation

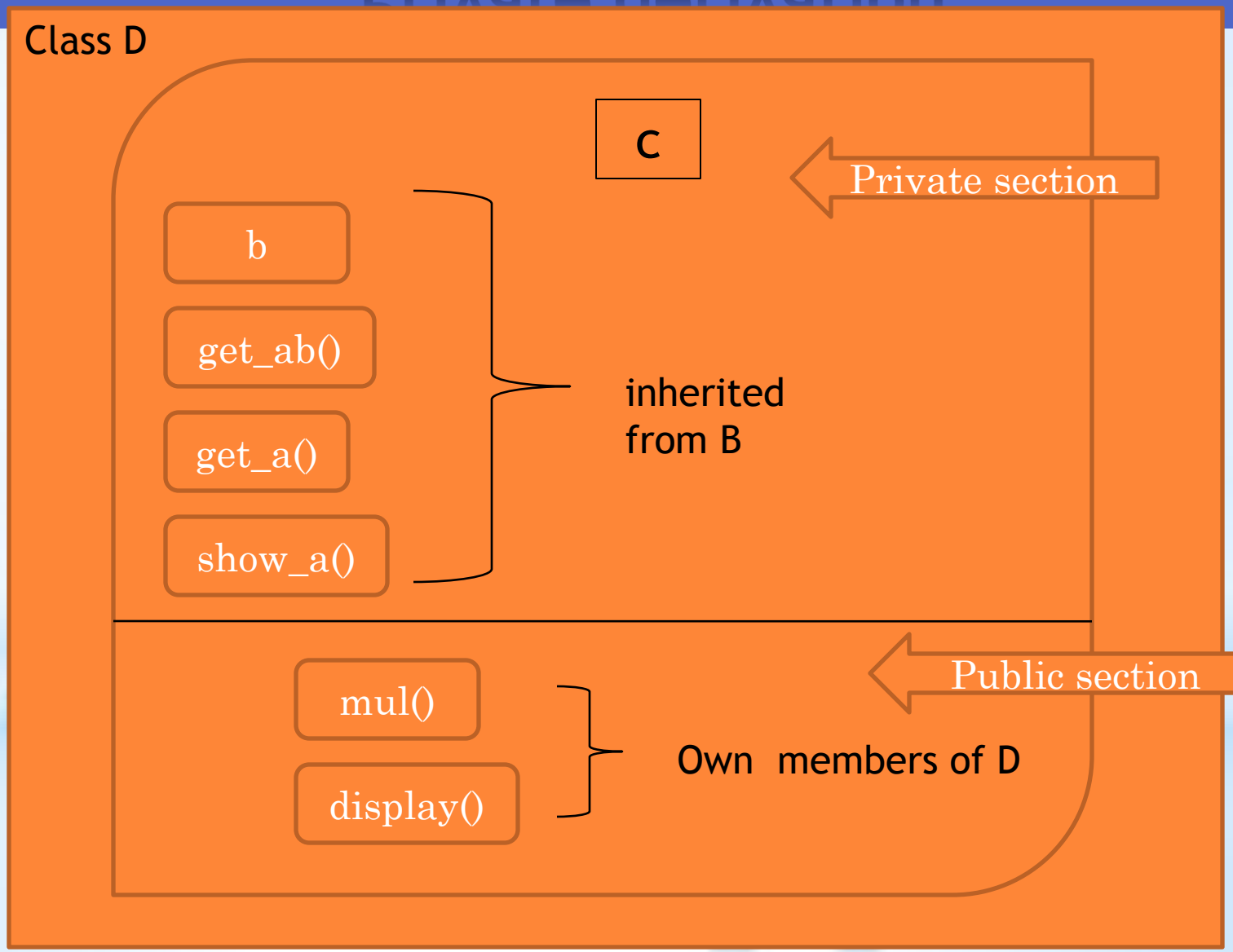


Single inheritance : private derivation

```
class  B
{
    int a;  // private, not inheritable
public :   // public, ready for inheritance
    int b;
    void get_ab();
    int get_a(void);
    void show_a();
};
class  D : private  B  // private derivation
{
     int c;
  public:
     void mul(void);
     void display(void);
};
```

```
void B :: get_ab(){
cout<< "enter the values of a & b";
cin>>a>>b;
}

int B :: get_a(){
    return a;
}

void  B :: show_a(){

Cout<<"a = "<<a<<"\n";
}

void D :: mul(){

get_ab();
c = b * get_a();
// 'a' can not be directly used
}
```

Private derivation



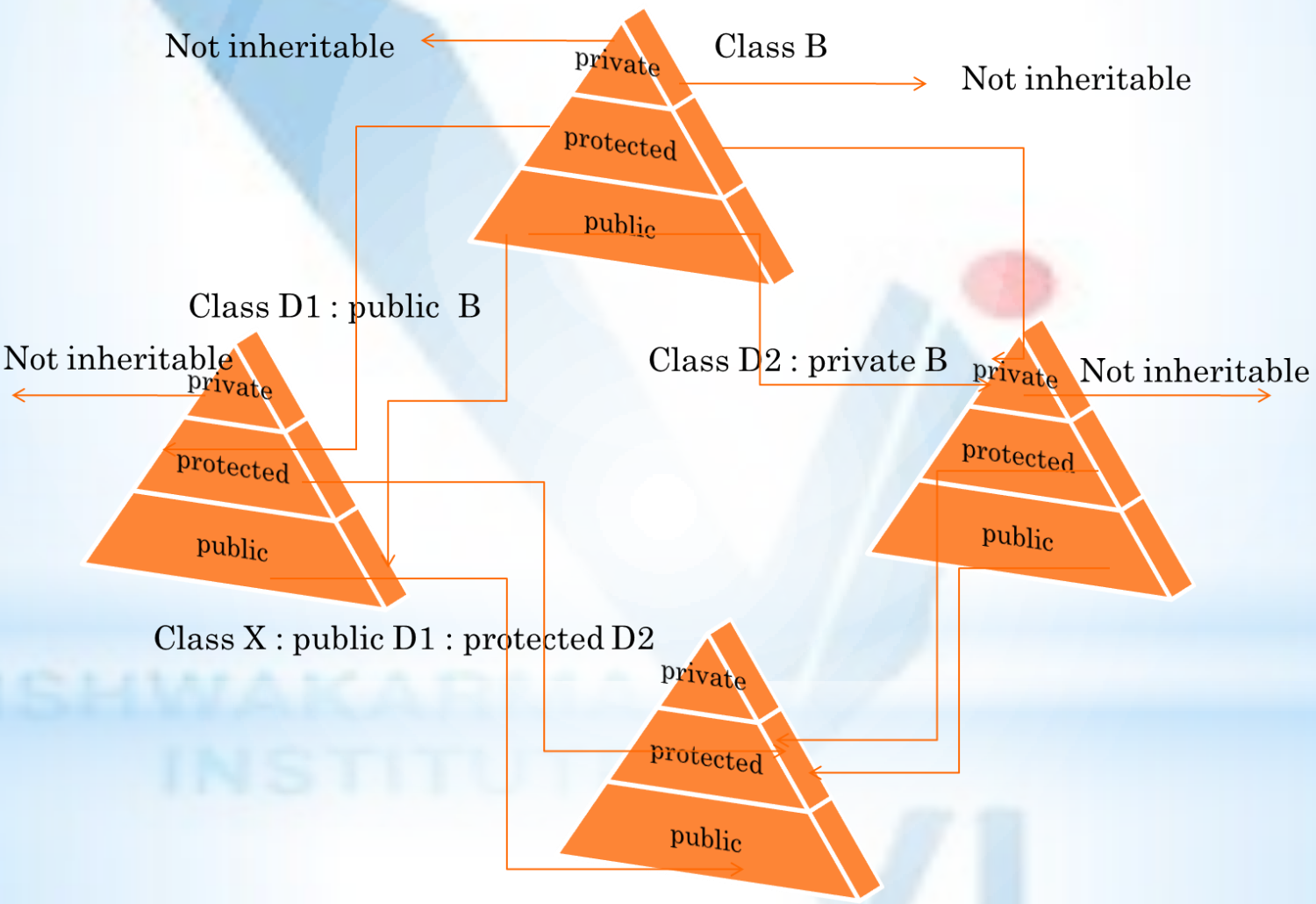
```
void D :: display(){

    show_a();     // outputs value of ' a '
    cout<< "b = "<<b<<"\n";
    cout<< "c = "<<c<<"\n";

}
```

```
int main() {

        D d;
        // d.get_ab();   // WON'T WORK
        d.mul();
        // d.show_a();  // WON'T WORK
        d.display();

        // d.b=20;   // WON'T WORK,
// b has become private
        d.mul();
        d.display();

        return 0;
}
```

Protected Members

- Protected members from base class can be accessed by own class and its all subclasses.
- Protected members are not accessible other than own class and derived class
- They created solely for inheritance
- They are combination of private and public access control.
- They are private to own class and public to derived class.

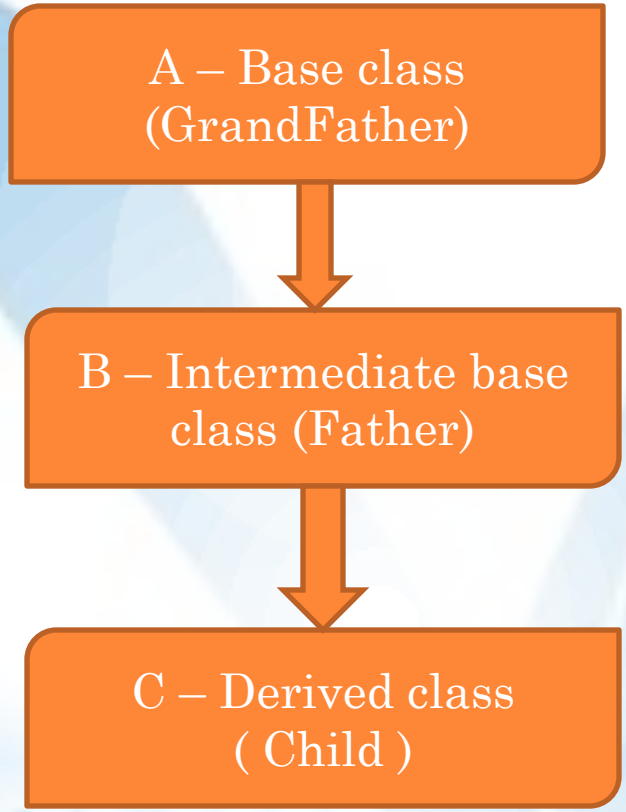
Effect of inheritance on visibility of members



Visibility of inherited members

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

Multi-level inheritance




```
Class A {……..};                    // Base class A
Class B : public A{……};             // B derived from A
Class C : public B{……};             // C derived from B
```

```
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};
```

```
// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base
classes
    Car obj;
    return 0;
}
```

*

output:

This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels

Multiple inheritance

- A class inheriting attributes of two or more classes is called multiple inheritance

Syntax :

Class D : visibility B-1, visibility B-2, ...

{

.....// members of D

}

The base classes are separated by comma

```
// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};
```

```
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

* Output:
  This is a Vehicle
  This is a 4 wheeler Vehicle

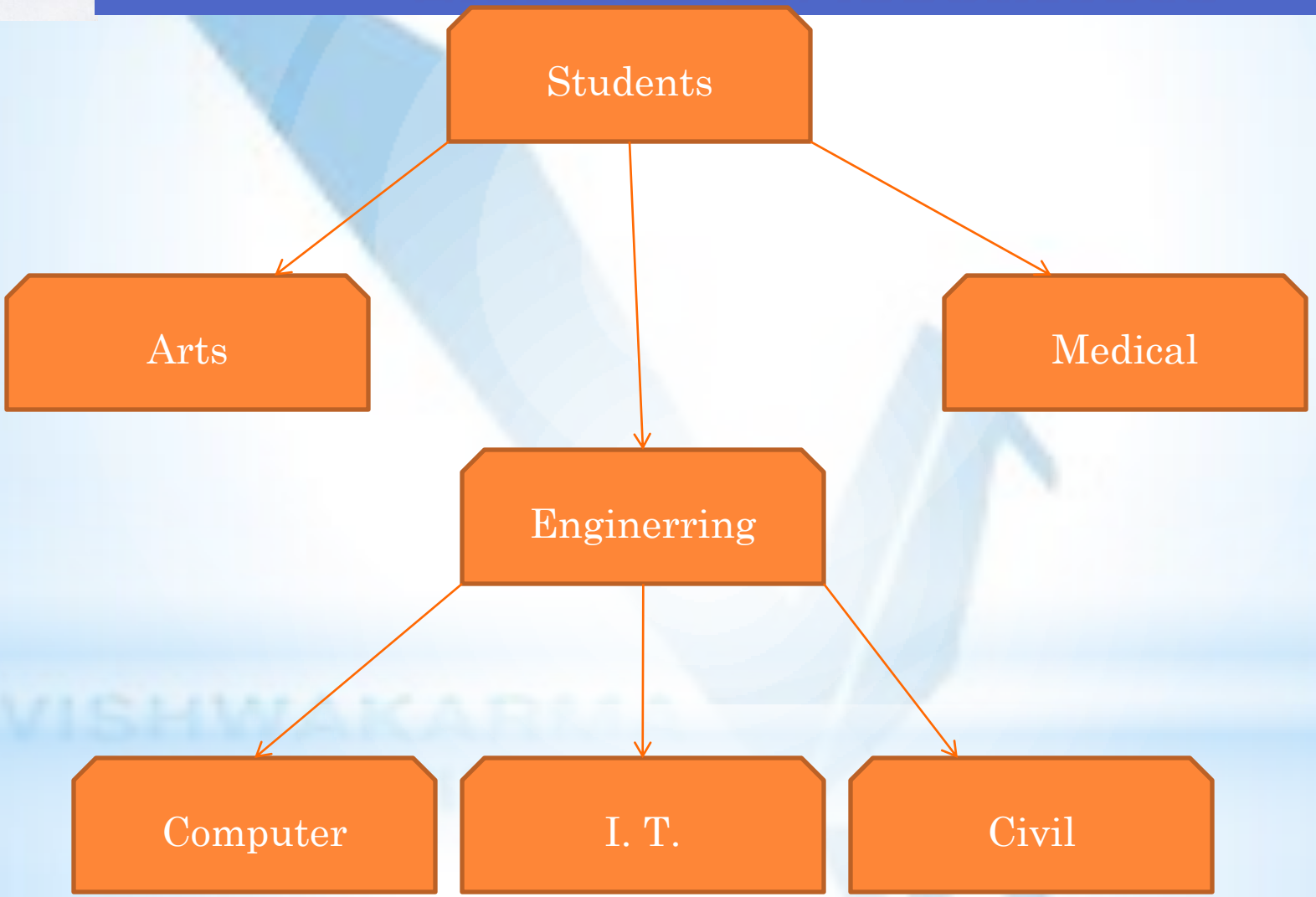
Ambiguity in Multiple Inheritance

- Multiple inheritance faces a problem of ambiguity, when multiple base class have members with same name
- Due to this derived class faces a ambiguity as which class version of the member they should use
- At this condition compiler get confused
- Can be resolved by scope resolution operator

Objectname.Classname::function_name()

```
class x
{
public:
void display()
{
cout<<"\nThis is base class x";
}
};
class y
{
public:
void display()
{
cout<<"\n This is second base class y";
}
};
class z:public x,public y
{
public:
void display_dev()
{
cout<<"\nThis is derived class inherited form x & Y";
}
};
int main()
{
cout<<"\n Demostration of Handling Ambiguity in Multiple inheritance";
z ob;
ob.x::display();//Access display function of class x
ob.y::display();//Access display function of class y
ob.display_dev();//Access Display function of
return 0;// derived class
}
```

Hierarchical inheritance

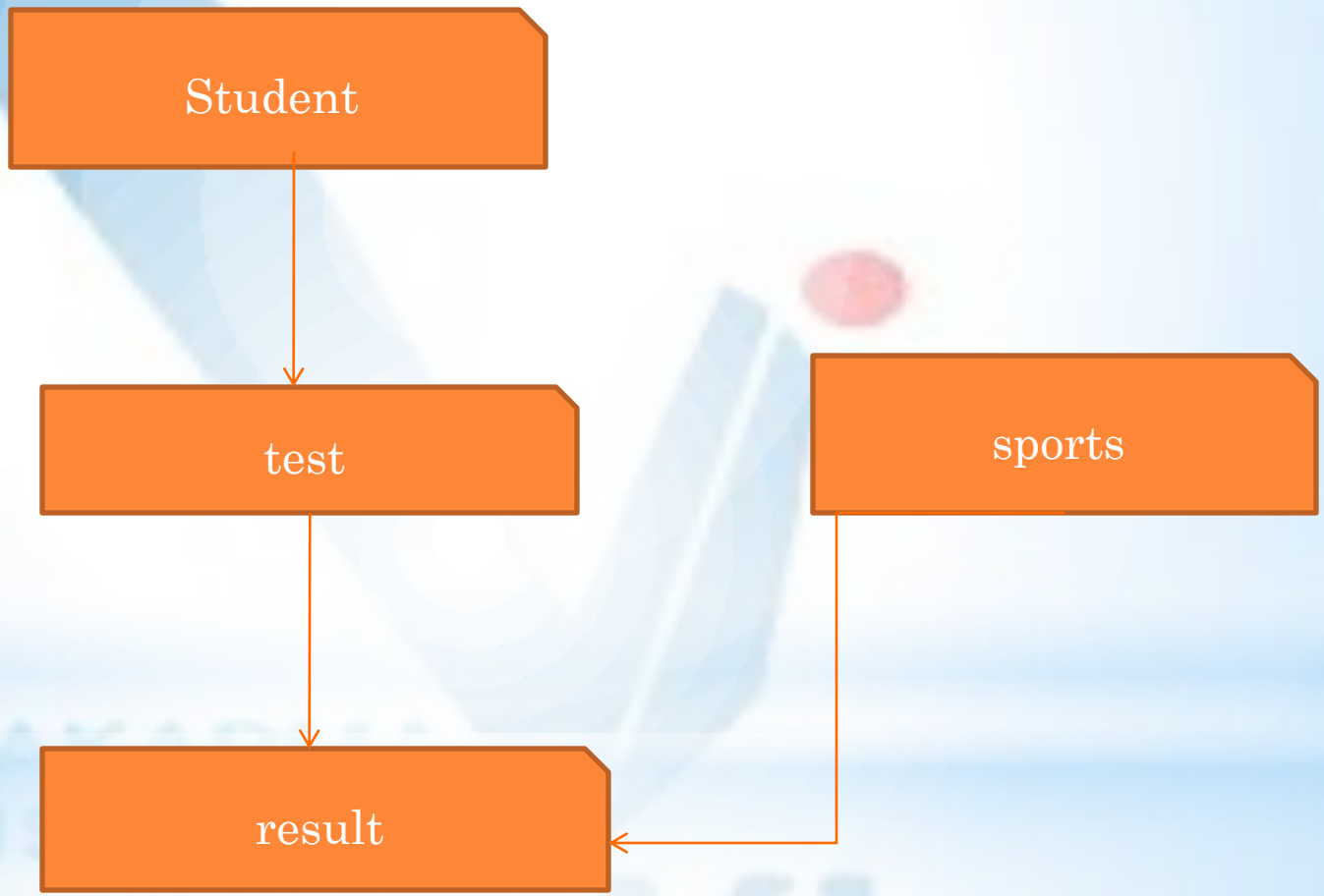


```
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
  // first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};
```

```
// main function
int main()
{
    // creating object of
sub class will
    // invoke the
constructor of base
class
    Car obj1;
    Bus obj2;
    return 0;
}
```

* Output:
  This is a Vehicle
  This is a Vehicle

Hybrid inheritance



```
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
 //base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
```

```
// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle, public Fare
{

};
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

* Output:

  This is a Vehicle

  Fare of Vehicle

Constructor and Destructor in Derived Class

- Base and derived class have their own constructor
- Constructor is used to construct the object and destructor is called destroy the object
- Order of Execution:

  First base constructor is called

  Next derived constructor is called

  Next destructor of derived class is called

  Next destructor of base class is called

Constructor and Destructor in Derived Class(cont…)

```
class base
{
public:
base()
{
cout<<“Base class constructor is called”;
}
~base()
{
cout<<“Base class destructor is called”;
}};
```

```
class child:public base
{
public:
child()
{
cout<<“child class constructor is called”;
}
~child()
{
cout<<“child class destructor is called”;
}};
```

Constructor and Destructor in Derived Class(cont…)

```
int main()
{
child d;
return 0;
}
```

- If base class constructor is not taking argument, then there is no need to define constructor in derived class
- If base class is having a constructor with argument then we need to define constructor in derived class and pass argument to base class constructor

Constructor and Destructor in Derived Class(cont…)

- In multiple inheritance, base classes are constructed in the order in which they appear in the declaration of derived class.
- In multilevel inheritance also constructors are executed in order of inheritance.
- Child class or derived class take care of arguments for base class. We need to provide necessary which are required by all base classes.

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

class A
{
public:
  A()  { cout << "A's constructor called" << endl; }
};

class B
{
public:
  B()  { cout << "B's constructor called" << endl; }
};

class C: public B, public A  // Note the order
{
public:
  C()  { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
    return 0;
}
```

Output:

B's constructor called
A's constructor called
C's constructor called

Aggregation Classes within Classes

- If class B is derived from A then we say B Kind of A
- B has all data members and member function of parent class A
- This relation is know as ***aggregation***
- InOOP, aggregation is occurs when one object is attribute of another class.

Classes within Classes

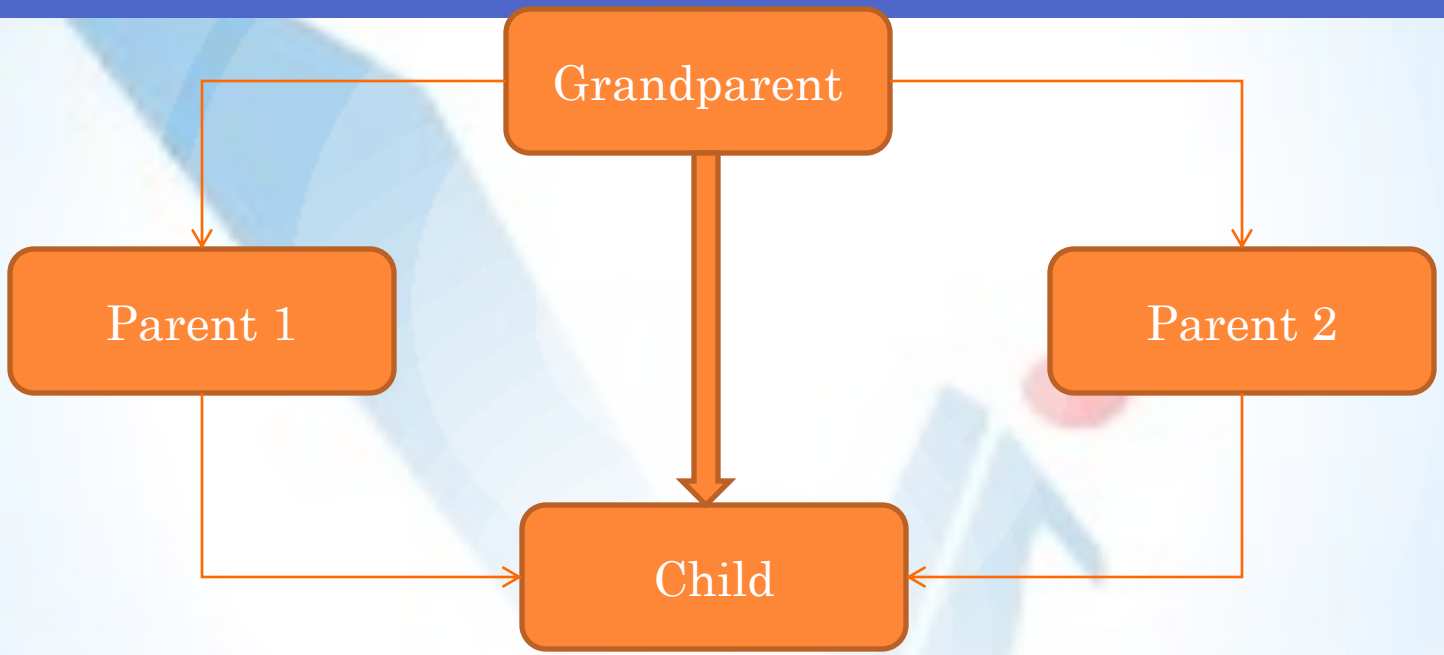
- class declared inside another class
- Nested class is member of outer class so it has access right like other members of a class
- Member of outer class has no special access to member of nested class.

Classes within Classes

```
class outerclass
{
public:
        class innerclass
        {
        private:
        int sum:
        public:
        void addition(int x, int y)
        {
        sum=x+y;
        }
        void disp()
        {
        Cout<<"Addition is"<<sum;
        }
        };
        };
        int main()
        {
        outerclass::innerclass ob;
        ob.addition(25,15);
        Ob.disp();
        }
```

//Object Creation

Virtual base class



- In above case all the public as well as protected members of grandparent are inherited ‘twice’ , so child would have duplicate copies.
- Can be avoided by making common base class(grandparent in this case) as virtual base class
- This problem is called as Diamond problem

Example VBC

```
Class A {        // grandparent
};

Class B1 : virtual public A // Parent 1
{
};

Class B2 :  virtual public A // parent 2
{
}
Class C : public B1, public B2  // child
{                    // only one copy of A will be inherited
}
```

```
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;
 };
```

```
 void main()
{
 ClassD obj;
obj.a = 10; //Statement 1, Error occur
obj.a = 100; //Statement 2, Error occur
obj.b = 20;
obj.c = 30;
obj.d = 40;
cout<< "\n A : "<< obj.a;
cout<< "\n B : "<< obj.b;
cout<< "\n C : "<< obj.c;
cout<< "\n D : "<< obj.d;
}
```

*error

```
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;
};
```

```
void main()
{
 ClassD obj;
obj.a = 10; //Statement 1
obj.a = 100; //Statement 2
obj.b = 20;
obj.c = 30;
obj.d = 40;
cout<< "\n A : "<< obj.a;
cout<< "\n B : "<< obj.b;
cout<< "\n C : "<< obj.c;
cout<< "\n D : "<< obj.d;
}
```

* Output :

```
A : 100
B : 20
C : 30
D : 40
```