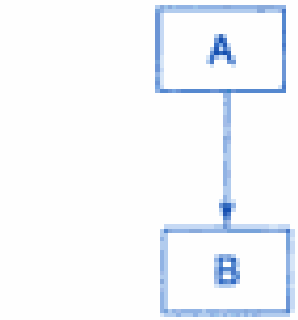
INHERITANCE

Introduction

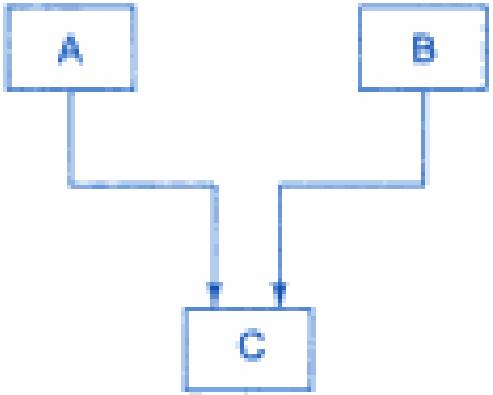
- Reusability--building new components by utilizing existing components- is yet another important aspect of OO paradigm.
- It is always good/ “productive” if we are able to reuse something that is already exists rather than creating the same all over again.
- This is achieve by creating new classes, reusing the properties of existing classes.
- It saves money , time etc.
- This mechanism of deriving a new class from existing/old class is called **“inheritance”**.
- The old class is known as **“base” class**, “super” class or “parent” class”.
- The new class is known as “sub” class **“derived” class**, or “child” class.

Types of Inheritance

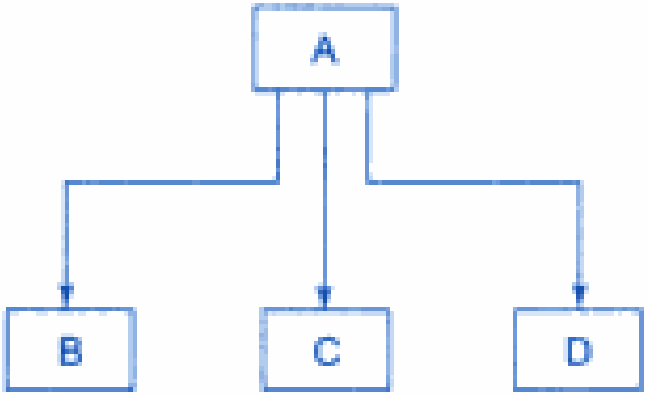
1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance



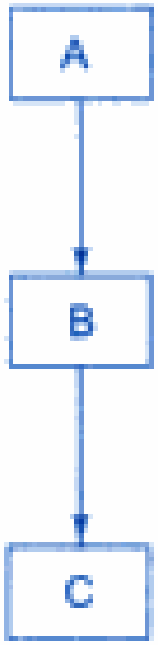
(a) Single inheritance



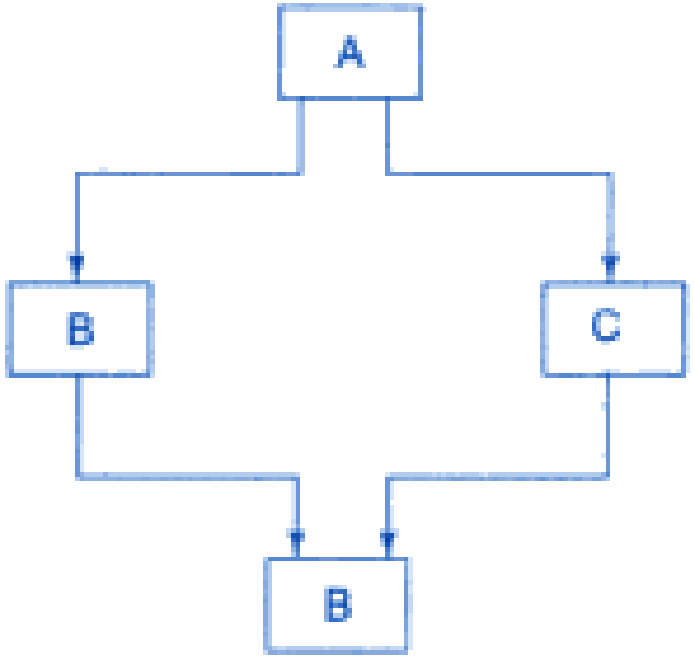
(b) Multiple inheritance



(c) Hierarchical inheritance



(d) Multilevel inheritance



(e) Hybrid inheritance

Defining Derived classes

- Syntax:

  class *DerivedClassName* : access-level *BaseClassName*

  where
  - access-level specifies the type of derivation
    - private by default, or
    - Public
    - Protected

- Any class can serve as a base class
  - Thus a derived class can also be a base class

Implementing Inheritance in C++ by Deriving Classes From the Base Class

- Syntax:

```
class <base_class>
{
…
};
class <derived_class> : <access-specifier><base_class>
{
...
};
```

Examples

```
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
};
```

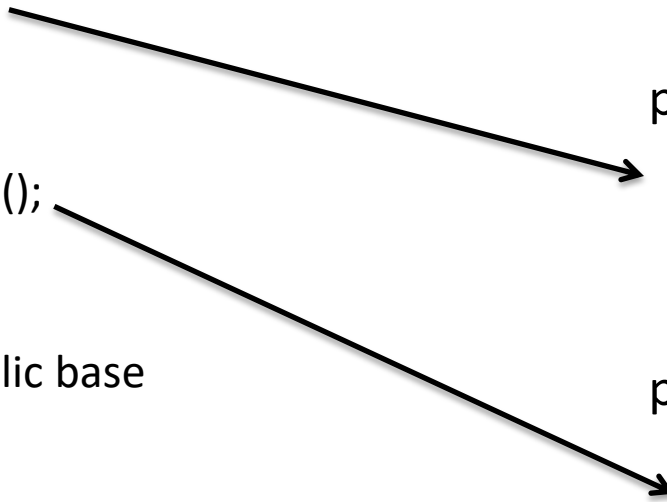
Public inheritance

- When a base class is publicly inherited, ‘public members’ of the base class become ‘public members’ of the derived class and therefore they are accessible to the objects of the derived class.
- **Note:** private members of the base class are not accessible in the derived class (to preserve encapsulation)

Public derivation

```
Class base
{    private:
       int base_a;
     protected:
       int base_b;
     public:
       void base_set();
};
Class derived: public base
{
     private:    int derived_a;
     protected:      int derived_b;
     public:     void derived_set();
};
```

```
Class derived: public base
{
     private:
       int derived_a;
     protected:
       int base_b;
       int derived_b;
     public:
       void base_set();
       void derived_set();
};
```



Private inheritance

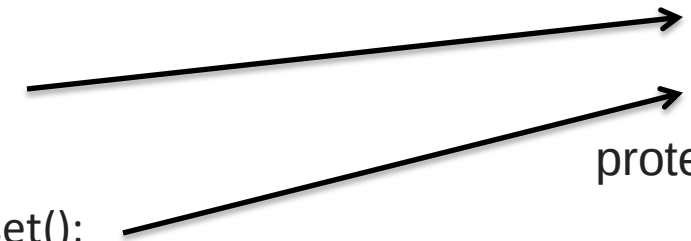
- Public members of base class become private members of derived class
- Public and protected members are only available to derived-class member functions - not to a derived object.
- They are inaccessible to the objects of the derived class.

Private derivation

```
Class base
{
    private:
        int base_a;
    protected:
        int base_b;
    public:
        void base_set();
};

Class derived: private base
{
    private:        int derived_a;
    protected:      int derived_b;
    public:         void derived_set();
};
```

```
Class derived: protected base
{
    private:
        int derived_a;
        int base_b;
        void base_set();
    protected:
        int derived_b;
    public:
        void derived_set();
};
```



Protected inheritance

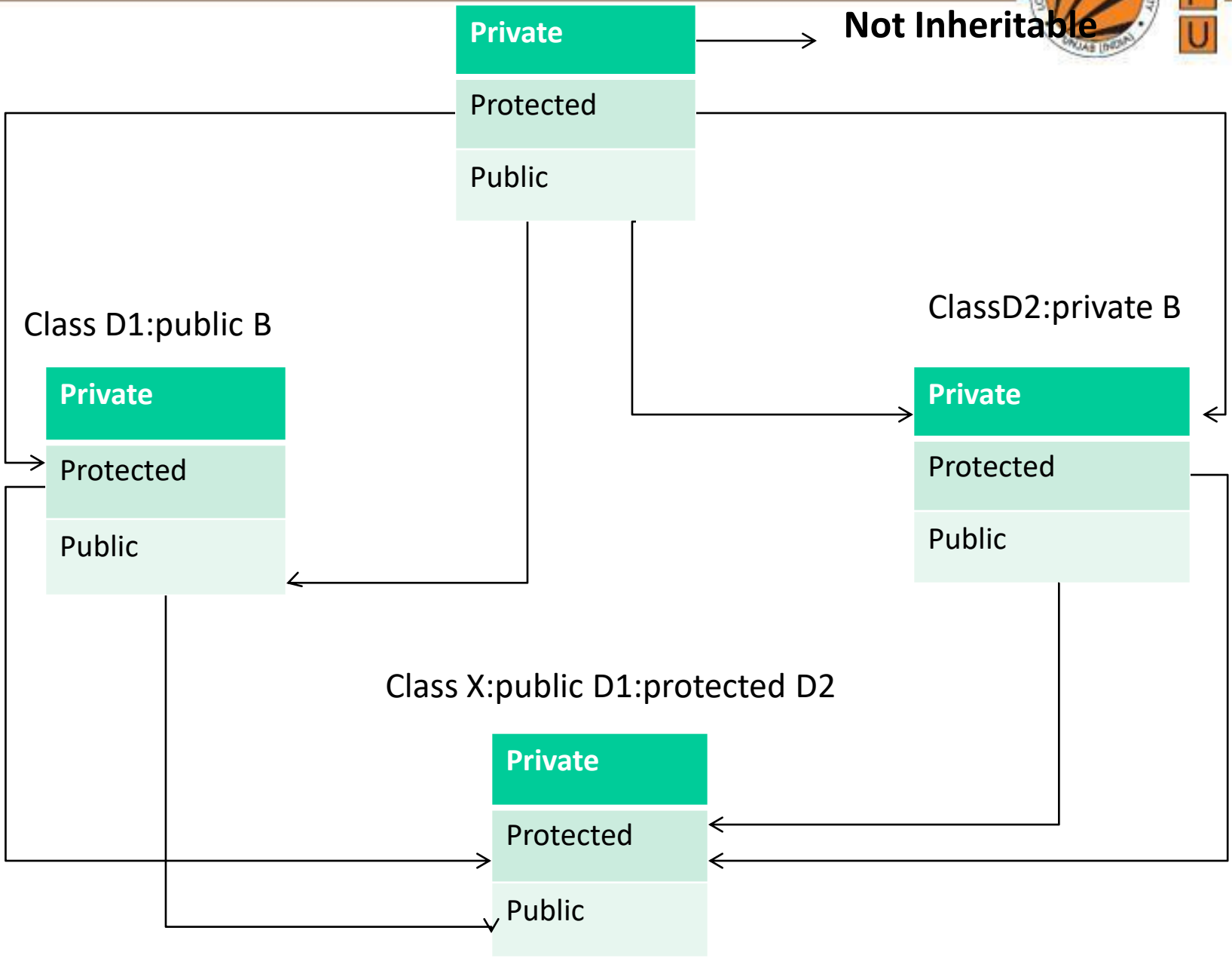
- A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- It cannot be accessed by the functions outside these two classes.
- It is possible to inherit a base class in protected mode. In this, Public and protected members of the base class become protected members of the derived class.

Protected derivation

```
Class base
{
    private:
        int base_a;
    protected:
        int base_b;
    public:
        void base_set();
};

Class derived: protected base
{
    private:        int derived_a;
    protected:      int derived_b;
    public:         void derived_set();
};
```

```
Class derived: protected base
{
    private:
        int derived_a;
    protected:
        int base_b;
        int derived_b;
        void base_set();
    public:
        void derived_set();
};
```


Private
Not Inheritable
Protected
Public
Class D1:public B
Private
Protected
Public
ClassD2:private B
Private
Protected
Public
Class X:public D1:protected D2
Private
Protected
Public

Access Rights of Derived Classes

Type of Inheritance

Access Control for Members	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

- The type of inheritance defines the access level for the members of derived class that are inherited from the base class

Understanding Inheritance Restrictions

- The following are never inherited:
  - constructors
  - destructors
  - `friend` functions
  - overloaded `new` operators
  - overloaded = operators
- Class friendship is not inherited

Single Inheritance

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

Single Inheritance

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

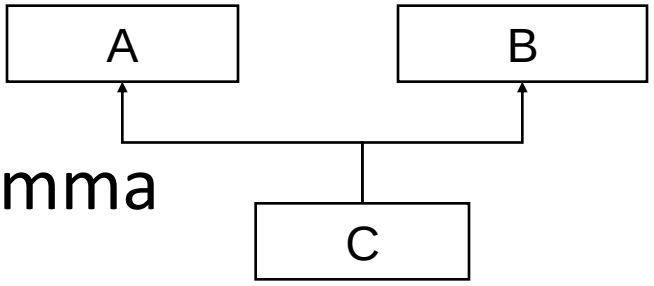
```
int B::get_a()
{
return a;
}
void B::show_a()
{
cout<<“a=“a<<“\n”;
}
void D::mul()
{
get_ab();
c= b*get_a();   //’a’ can’t be
                    used directly
}
void D::display()
{ show_a();      //output ‘a’
cout<<“b=“<<b<<“\n”;
cout<<“c=“<<c<<“\n”;
}
int main()
{
D d;
//d.get_ab();    //won’t work
d.mul();
//d.show_a();   //won’t work
d.display();
//d.b=20;      //won’t work
d.mul();
d.display();
return 0;
}
```

Multiple Inheritance

- Is the phenomenon where a class may inherit from two or more classes
- Syntax:

```
class derived : public base1, public base2
 {
 //Body of class
 };
```

- Base classes are separated by comma

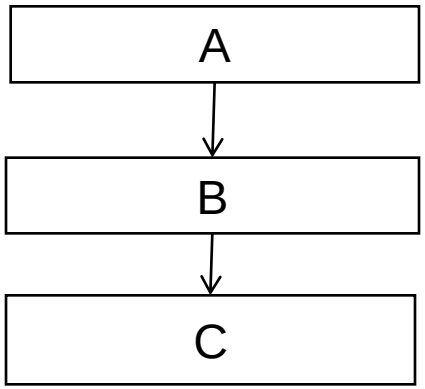


Multiple Inheritance

```
#include <iostream.h>
class M
{
protected:
    int m;
public:
    void get_m(int);
};
class N
{
protected:
    int n;
public:
    void get_n(int);
};
class P:public M, public N
{
public:
    void display(void);
};
void M::get_m(int x)
{ m=x;}
void N::get_n(int y)
{ n= y;}
void P::display(void)
{
cout<<“m=“<<m<<“\n”;
cout<<“n=“<<n<<“\n”;
cout<<“m*n=”<<m*n<<“\n”;
}
int main()
{
P p;
p.get_m(10);          //m=10
p.get_n(20);              //n=20
p.display();          //m*n = 200
return 0;
}
```

Multilevel Inheritance

- It is also possible to derive a class from an existing derived class.
- It is implemented by defining atleast three classes.
- Each derived class must have **a kind of** relationship with its immediate base class.



Multilevel Inheritance

```
#include <iostream.h>
class student
{
protected:
    int roll_no;
public:
    void get_no(int);
    void put_no(void);
};
void student::get_no(int a)
{ Roll_no=a;}
void student ::put_no()
{ cout<<"Roll number
    is"<<roll_no;
}
```

```
class test : public student   //first level
{                                derivation
protected:
     float sub1,sub2;
public:
    void get_marks(float,float);
    void put_marks(void);
};
void test::get_marks(float x,float y)
{ Sub1=x; sub2=y;}
void test::put_marks()
{ cout<<"Marks in sub1"<<sub1;
  cout<<"Marks in sub2"<<sub2;}
class result:public test    //second level
                               derivation
{ float total ;
  public:
    void display(void) ;}
```

```
Void result:: display(void)
{
Total= sub1+sub2;
Put_no();            //function of class student
Put_marks();         //function of class test
Cout<<“Total = “<<total;
}
Int main()
{
Result student1;
Student1.get_no(102);
Student1.get_marks(80.0,98.5);
Student1.display();
Return 0;
}
```

Ambiguity in Multiple Inheritance

- Can arise when two base classes contain a function of the same name

Example:

```
#include<iostream>
        class base1
        {
                public:
                void disp()
                {
                cout << "Base1" <<endl;
                }
        };
```

```
class base2
{
    public:
    void disp()
    {
    cout<< "Base2"<<endl;
    }
};
class derived : public base1,public
    base2
{
    //Empty class
};
```

```
int main()
{
        derived Dvar;
//Ambiguous function call
        Dvar.disp();
        return 0;
         }
```

- Can be resolved
  - By overriding the function in the derived class

```
Void disp()
{
base1::disp();
base2::disp();
}
```

- **Ambiguity in Single Inheritance**

```
class A
{
    public:
     void display()
     {
      cout<<"A\n";
      }
};
class B: public A
{
public:
    void display()
      {
       cout<<"B\n";
       }
};
```

```
int main()
{
B b;
b.display();   // invokes
                    display()  in B
return 0;
}
```

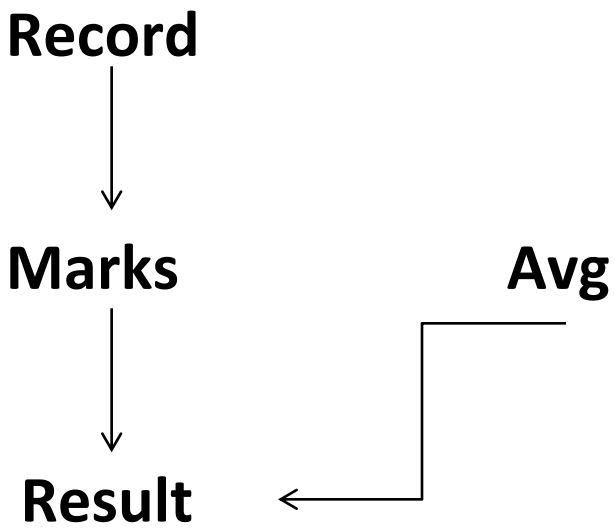
In this case, the function in the derived class overrides the inherited function.

Can be resolved:

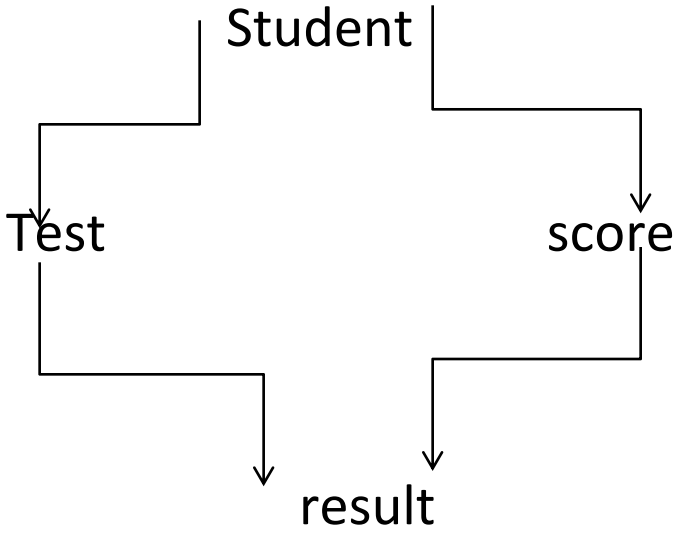
- By using the scope resolution operator

```
int main()
{
B b;
b.A::display();
b.B::display();
}
```

Hybrid Inheritance

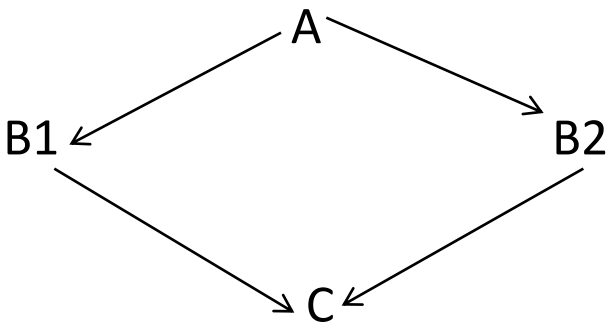


Virtual base class



//Multiple copies of the variables of student class are generated

Virtual base class




```
class A
    {…….
    ……
    };
class B1:virtual public A
    {…….
    …….
    };
class B2:public virtual A
    {…….
    ……..
    };
class C:public B1,public B2
    {……
    ……
    };    //only one copy of A
    will be inherited
```

Virtual base class

```
class student
{
protected: int roll_no;
public:
        void get_no(int a)
        {
        roll_no=a;
        }
        void put_no(void)
        {
        cout<<“roll no. :” <<
roll_no;
        }
 };
```

```
class test : virtual public
student
{
protected: float part1,part2;
public:
        void get_marks(float x,
float y)
        { part1=x; part2=y;}
        void put_marks()
        {    cout<<“marks
obtained”<<“part1”<<part1<<“
part2”<<part2;}
};
class sports: public virtual
student
{
```

```
protected :
     float score;
public:
          void get_score(float s)
          {  score=s;  }

void put_score()
{
cout<<“sports wt:”<<score;
}
};
class result : public test ,
public sports
 {
          float total;
          public:
          void display(void);
};
```

```
void result::display(void)
{
total= part1+part2+score;
          put_no();
          put_marks();
          put_score();
cout<<“total score: “<<total;
}
int main()
{
result student1;
student1.get_no(100);
student1.get_marks(50.5,65.2
);
student1.get_score(10.5);
student1.display();
return 0;
}
```

Constructors and Inheritance

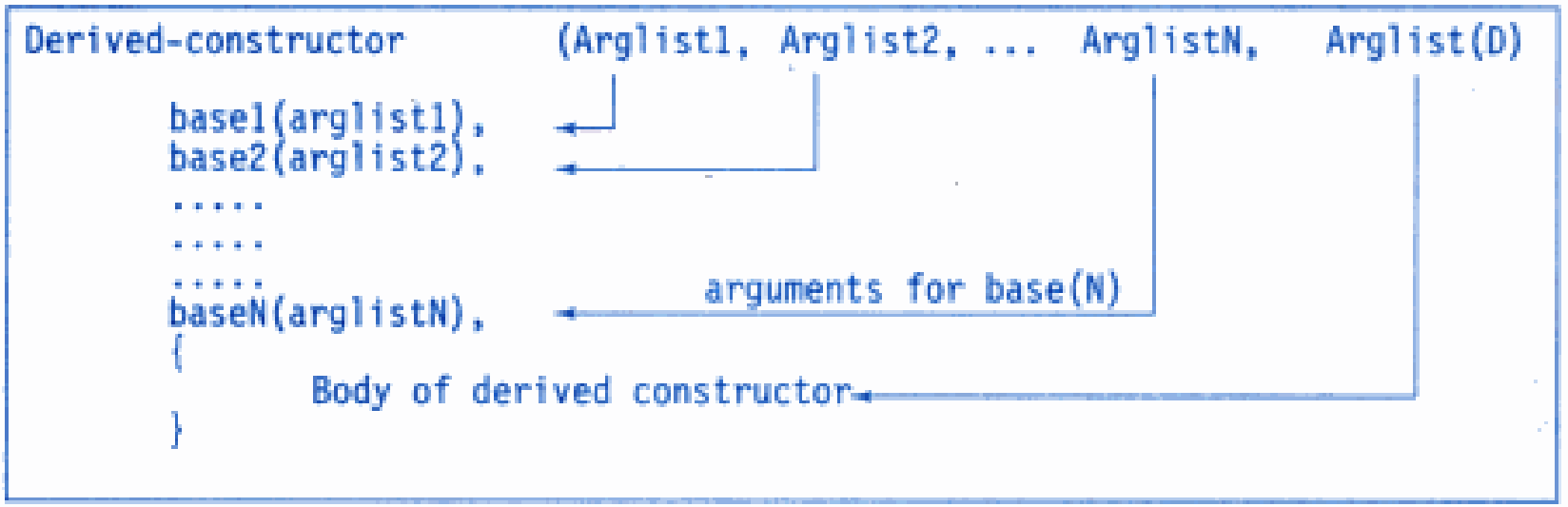
- If a base class constructor takes no arguments, the derived class need not to have constructor function.
- If any base class contain constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor.
- When both the derived and base classes contain constructor, the base class constructor is executed first and then the constructor in the derived class is executed.

Order of Constructors and destructors in derived classes

- Derived-class constructor
  - Calls the constructor for its base class first to initialize its base-class members
  - If the derived-class constructor is omitted, its default constructor calls the base-class' default constructor
- Destructors are called in the reverse order of constructor calls.
  - Derived-class destructor is called before its base-class destructor

Constructors in Derived Class

The general form of defining a derived constructor is:



- The general form of defining a derived constructor is :

```
Der_constructor(par_list) : base1(par_list1), base2(par_list2)
{
  // implementation of derived class constructor
}
```

Here Der_constructor is the name of derived class constructor and par_list specifies the list of parameters that the derived class constructor will receive, out of which some may be used to initialize its own data members whereas remaining ones may be passed to its base class constructors.

Initialization list starts with colon(:) and consists of calls to base class constructors where each call is separated by comma.

Example

- Derived class object created as:

```
D objD(5, 12, 2.5, 7.54, 30);
```

- Definition of constructor called:

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),          /* call to constructor A */
B(b1, b2)          /* call to constructor B */
{
     d = d1;      // executes its own body
}
```

Order of Constructors in derived classes

```
Class B:public A
{               //A() base constructor
};              //B() derived constructor


Class A:public B, public C
{               //B() base first
};              //C() base second
                //A() derived

Class A:public B, virtual C
{               //C() virtual base
};              //B() ordinary base
                //A() derived
```

Constructors in derived classes

```
#include <iostream>
class alpha
{
    int x;
    public: alpha(int i)
    {x=i;
    cout<<“alpha initialized”;}
    void show_x(void)
    { cout<<“x=“<<x;}
};
class beta
{   float y;
    public: beta(float j)
    {y=j;
    cout<<“beta initialized”;}
    void show_y(void)
    {
    Cout<<“y=“<<y;
    }
};
```

```
class gamma:public beta, public alpha
{
    int m,n;
    public:
    gamma(int a,float b,int c, int
    d):alpha(a), beta(b)
    {
     m=c;
     n=d;
    cout<<“gamma initialized”;
    }
    void show_mn(void)
    {
    cout<<“m=“<<m;
    cout<<“n=“<<n;
    }
};
```

Constructors in derived classes

```
int main()
{
gamma g(5,12.34,50,20);
g.show_x();
g.show_y();
g.show_mn();
return 0;
}
output:
beta initialized
alpha initialized
gamma initialized
```

Output

```
x=5
y=12.34
m=50
n=20
```

Initialization list

- C++ supports another method of initializing the class objects.
- This method uses what is known as initialization list in the constructor function

```
Constructor(par_list) : initialization section
{
// body of constructor
}
```

- The part immediately following the colon is called initialization section.
- We use this section to provide initial values to the base constructors and also to initialize its own class members
- The initialization section basically contains a list of initializations separated by commas, known as initialization list.

Example

```
Class XYZ
{
 int a;
int b ;
Public :
XYZ(int I, int j) : a(i), b(2*j)  {   }
};
int main()
{       XYZ  x(2,3);   getch() return 0;   }
```

This program will initialize a to 2 and b to 6

Example

```
class alpha
{
 int x;
 public:
 alpha (int i)
 {
 x=i;
 cout<<“alpha constructed”;
 }
void show_alpha(void)
{
 cout<<“x=“<<x;
} };
class beta
{
 float p,q;
 public:
  beta ( float a, float b) : p(a), q(b+p)
  {
   cout<<“ beta constructed”;
 }
void show_beta(void)
 {
  cout<<“=p”<<p;
  cout<<“q=“<<q;
 }};
 class gamma : public beta, public alpha
  {
  int u,v;
  public:
gamma (int a, int b, float c): alpha (a*2),
beta(c,c), u(a)
 {
 v=b;
  cout<<“gamma constructed”;
  }
  void show_gamma(void)
 {
  cout<<“u=“<<u;
  cout<<“v=“<<v;
  }
```

```
main()
{
gamma g(2,4,2.5);
cout<<“ display number:”;
g. show_alpha();
g. show_beta();
g.show_gamma();
}
```

OUTPUT:

```
Beta constructed
Alpha constructed
Gamma constructed
Display Numbers:
x=4
p=2.5
q=5
u=2
v=4
```