CSE 1203

Object Oriented Programming[C++]

Chapter 3:

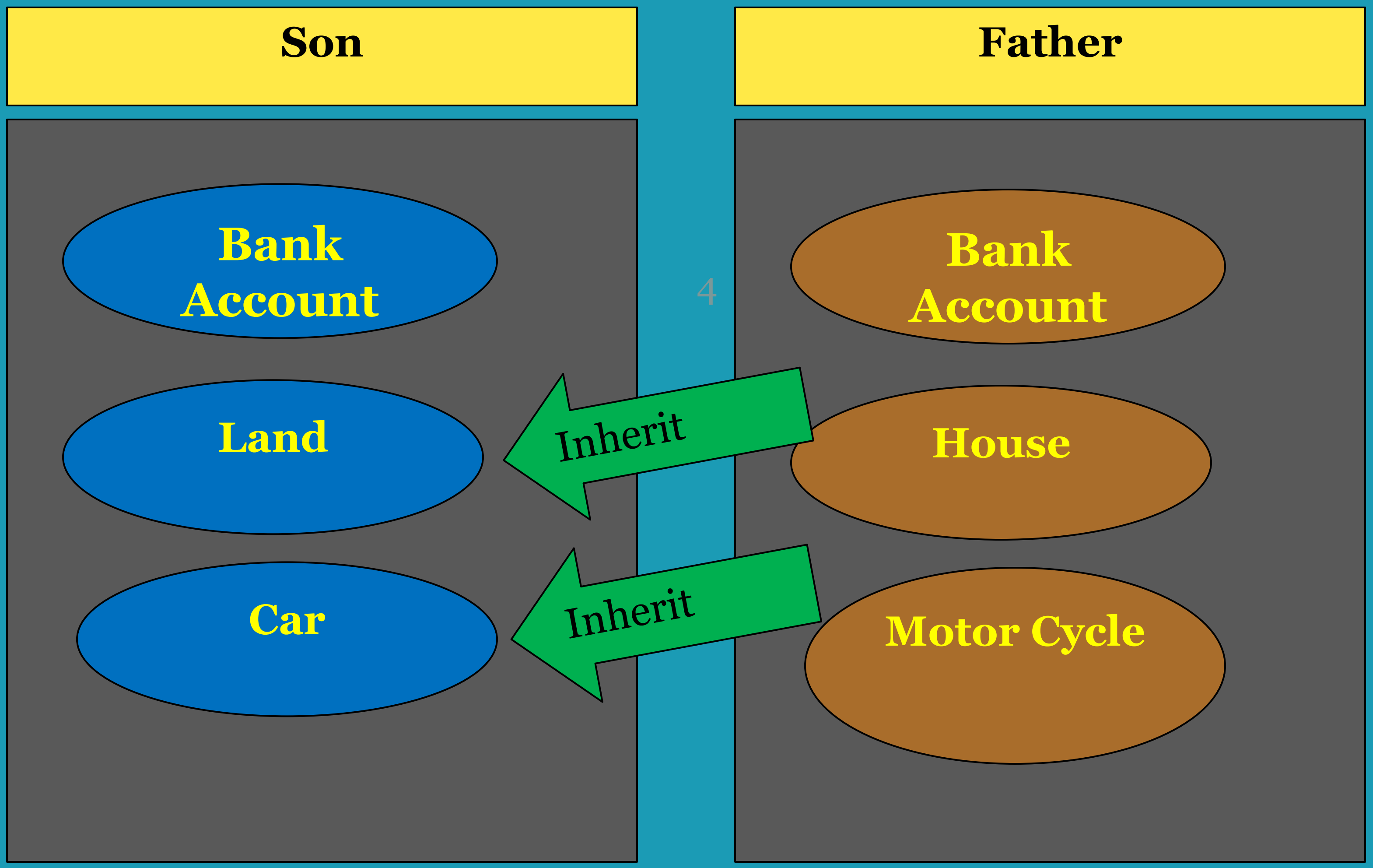
Inheritance

Learning Objectives

To know about:

- Inheritance
- Access Modifier
- Private Inheritance
- Protected Inheritance
- Public Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Constructor Calling Order in Inheritance

+4
© EPA



Son

Bank Account

Land

Car

Father

Bank Account

House

Motor Cycle

Inherit

Inherit

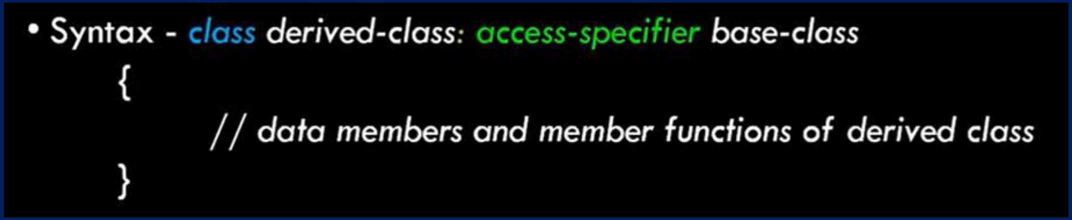
Inheritance Properties

- The mechanism of deriving a new class from an old class/previous written class in known as inheritance.
- It is also known as is a kind of relationship.
- The class which is inherited is called base class/parent class/super class.
- The class that inherits the base class is known as sub class/child class/derived class.
- Private members can never be inherited.
- A class can be inherited in 3 ways-

Public

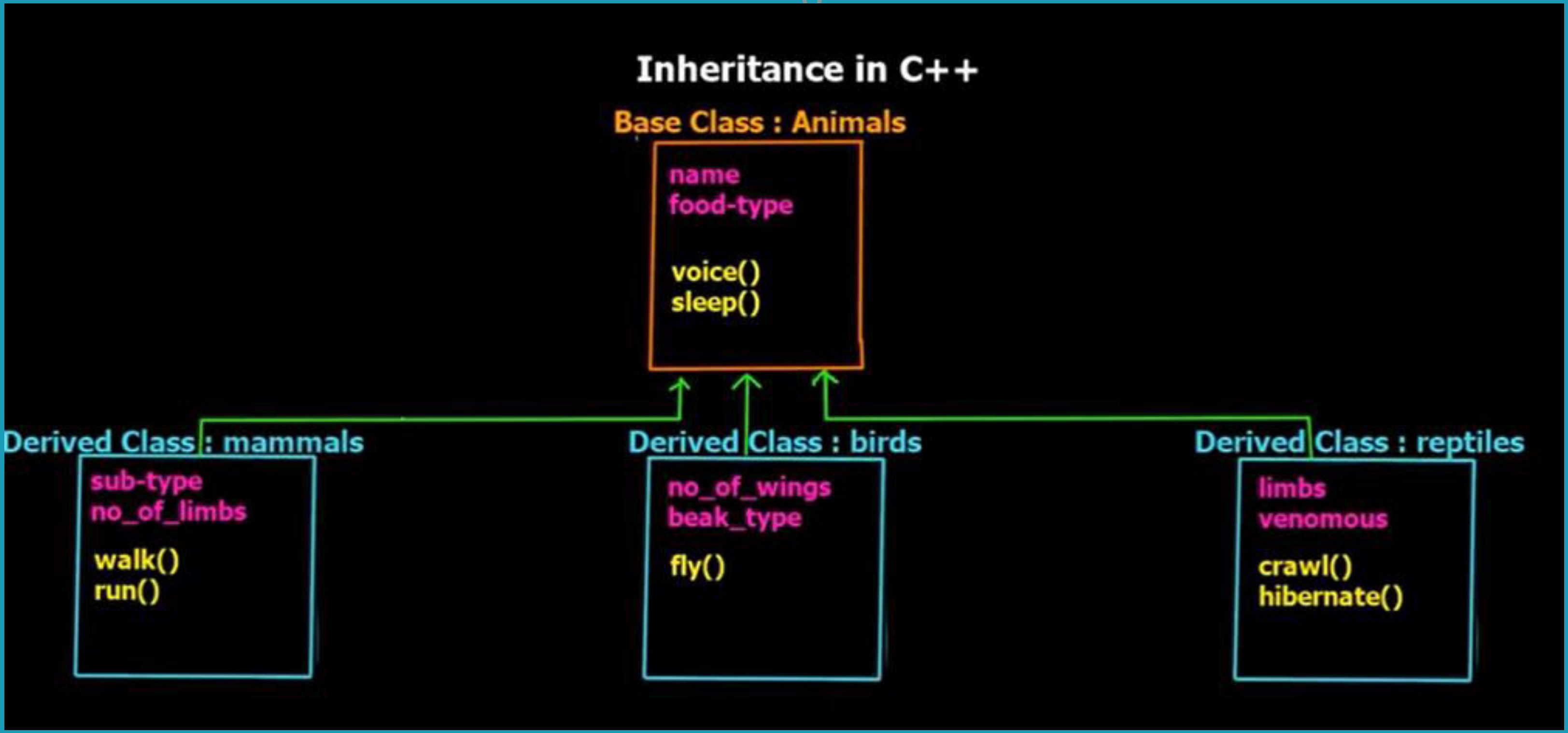
Protected

Private



```
• Syntax - class derived-class: access-specifier base-class
      {
             // data members and member functions of derived class
      }
```

Inheritance Example



Inheritance in C++

Base Class : Animals

name
food-type

voice()
sleep()

Derived Class : mammals

sub-type
no_of_limbs

walk()
run()

Derived Class : birds

no_of_wings
beak_type

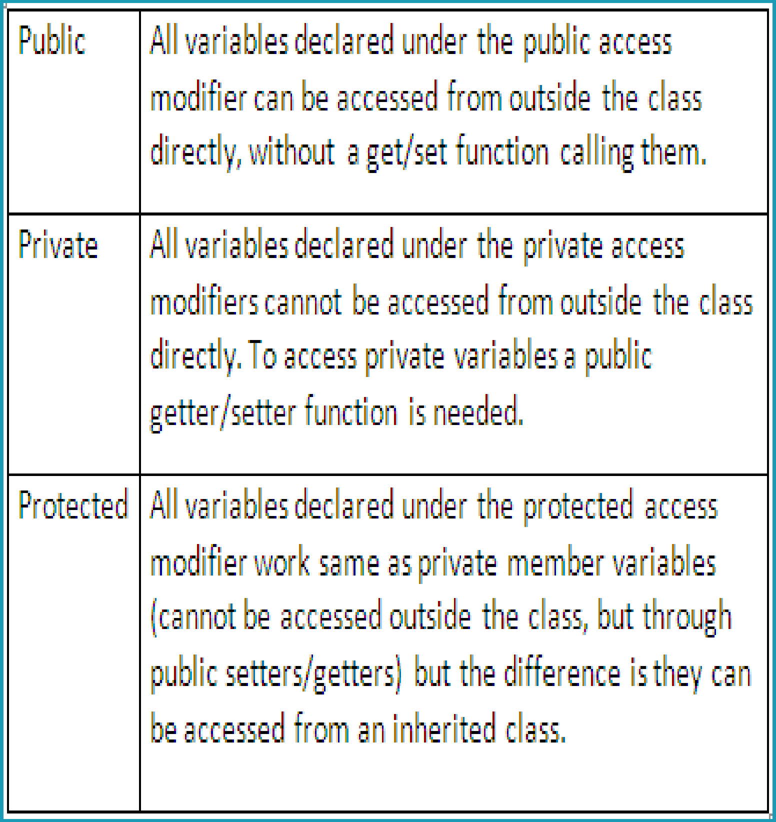
fly()

Derived Class : reptiles

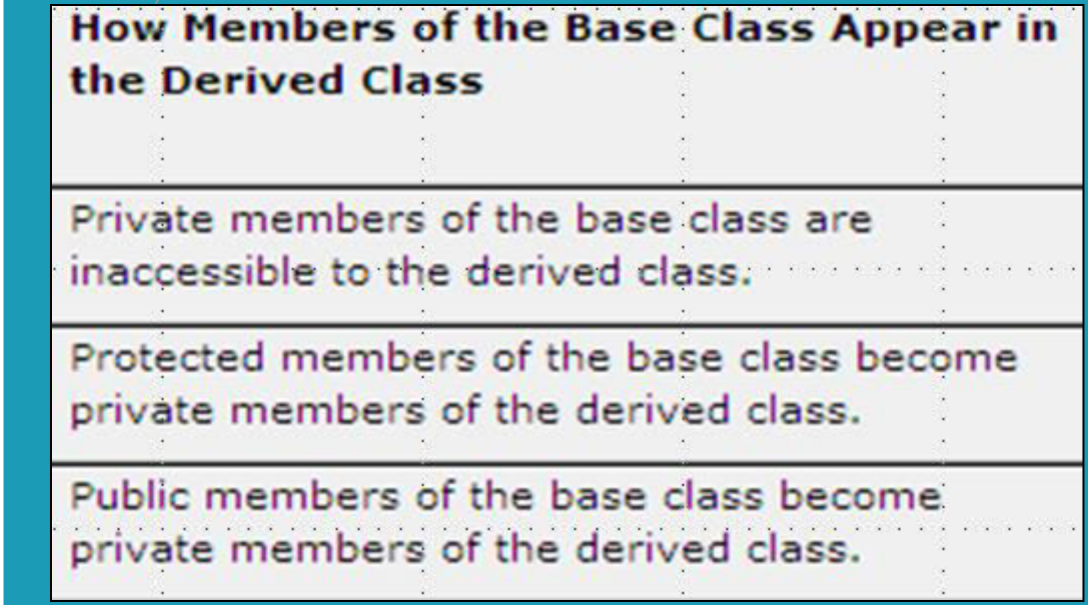
limbs
venomous

crawl()
hibernate()

Access Modifier

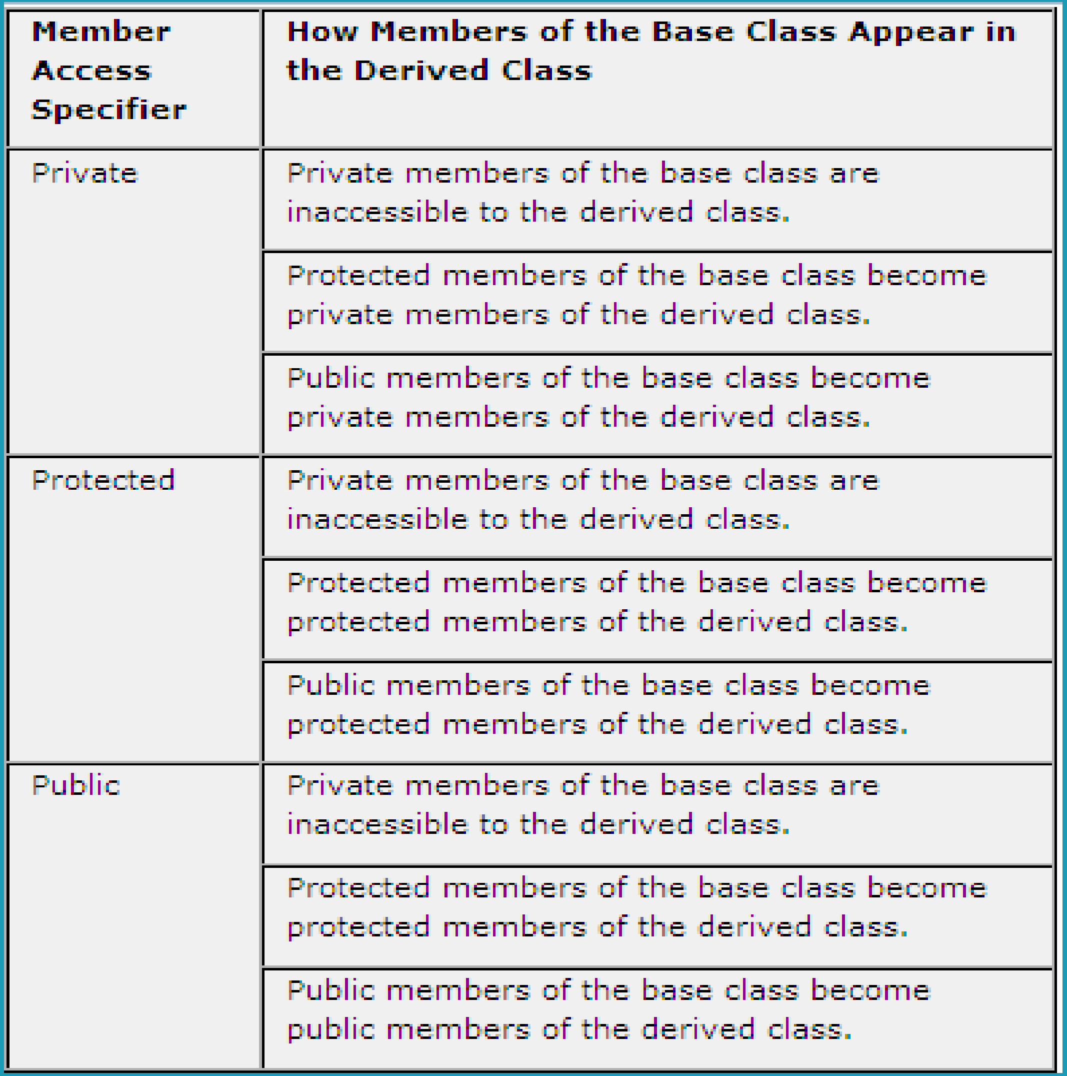


Public	All variables declared under the public access modifier can be accessed from outside the class directly, without a get/set function calling them.
Private	All variables declared under the private access modifiers cannot be accessed from outside the class directly. To access private variables a public getter/setter function is needed.
Protected	All variables declared under the protected access modifier work same as private member variables (cannot be accessed outside the class, but through public setters/getters) but the difference is they can be accessed from an inherited class.

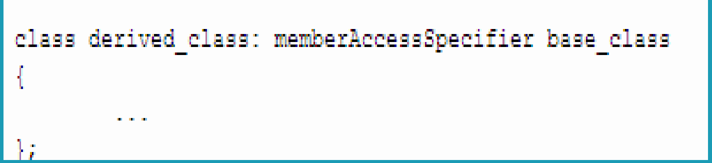


How Members of the Base Class Appear in the Derived Class
Private members of the base class are inaccessible to the derived class.
Protected members of the base class become private members of the derived class.
Public members of the base class become private members of the derived class.

Access Modifier

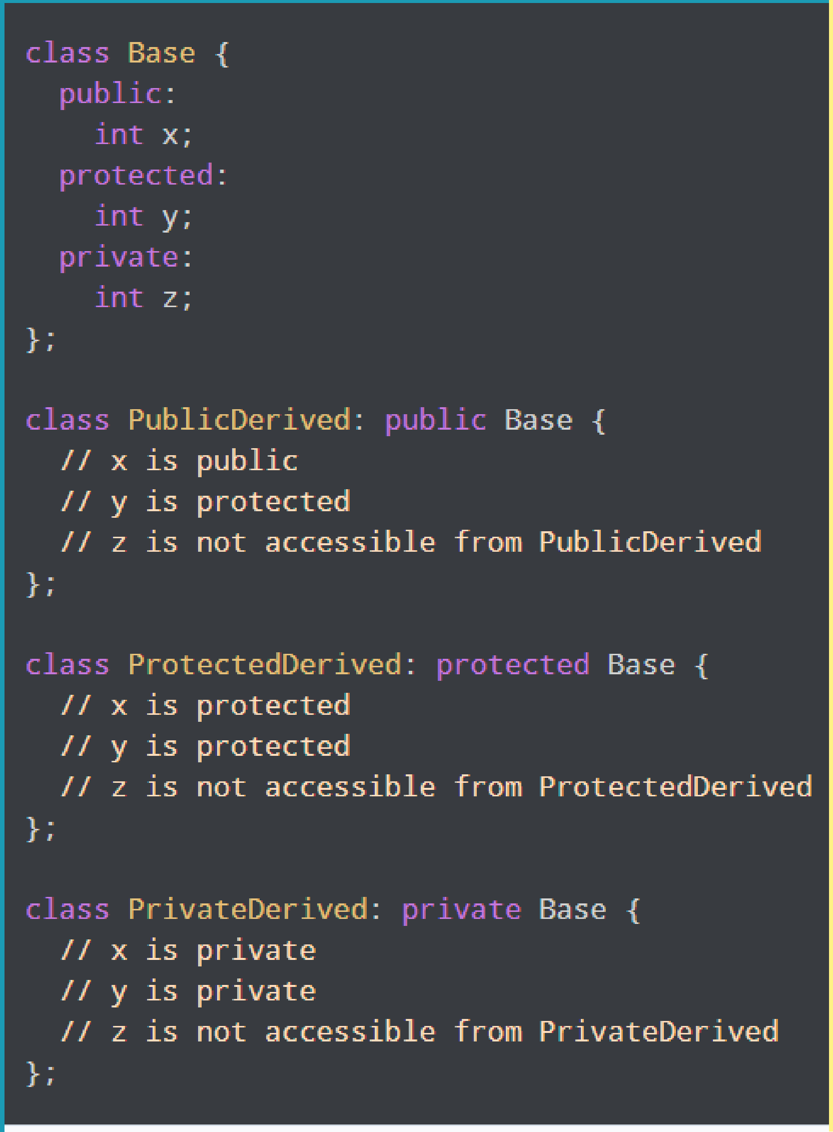


Member Access Specifier	How Members of the Base Class Appear in the Derived Class
Private	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
Protected	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
Public	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

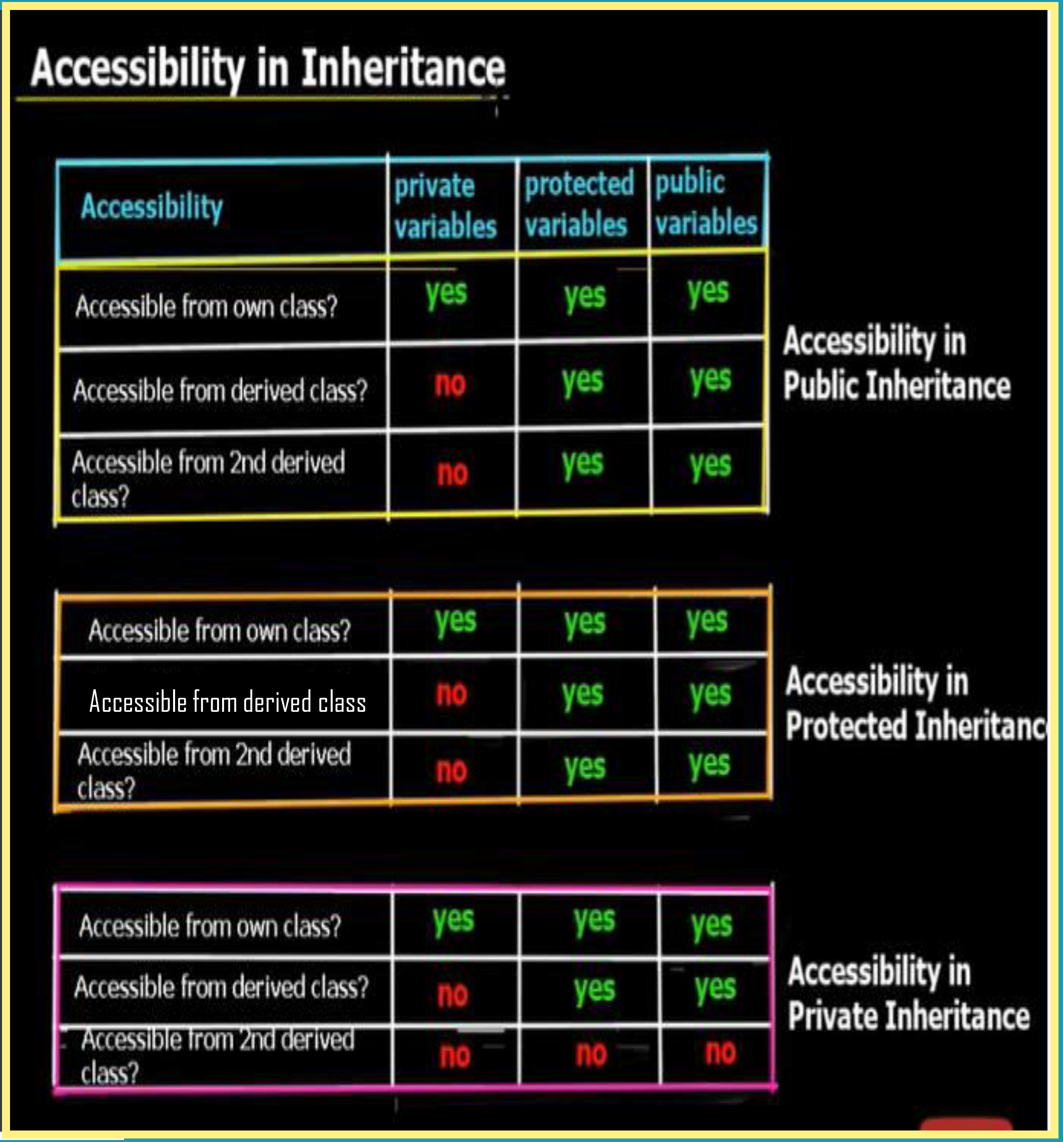


```
class derived_class: memberAccessSpecifier base_class
{
        ...
};
```

Inheritance: Access Modifier



```
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
};

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
};
```



Accessibility in Inheritance

Accessibility	private variables	protected variables	public variables
Accessible from own class?	yes	yes	yes
Accessible from derived class?	no	yes	yes
Accessible from 2nd derived class?	no	yes	yes

Accessibility in Public Inheritance

Accessible from own class?	yes	yes	yes
Accessible from derived class	no	yes	yes
Accessible from 2nd derived class?	no	yes	yes

Accessibility in Protected Inheritanc

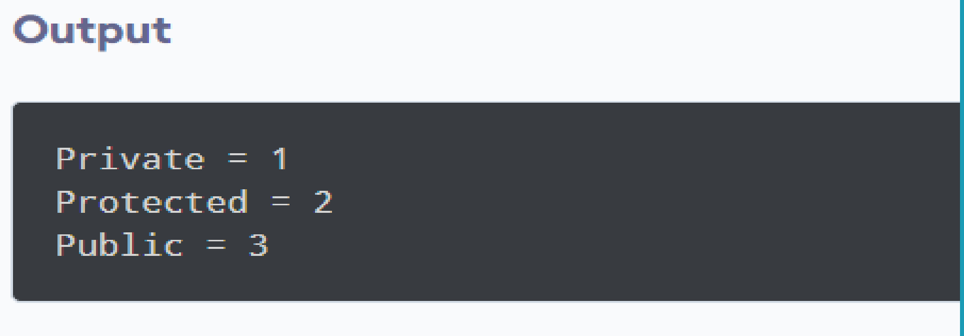
Accessible from own class?	yes	yes	yes
Accessible from derived class?	no	yes	yes
Accessible from 2nd derived class?	no	no	no

Accessibility in Private Inheritance

Inheritance: Public Modifier

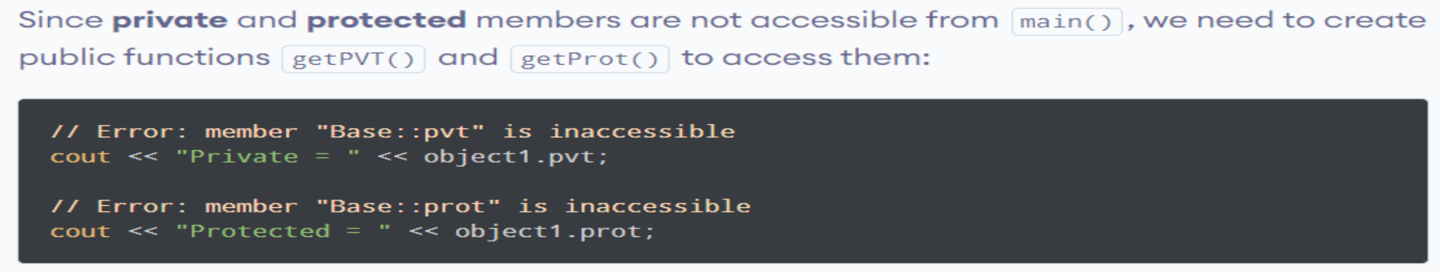
```
// C++ program to demonstrate the
//working of public inheritance
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#include <string>
using namespace std;

class Base {
 private:
  int pvt = 1;

 protected:
  int prot = 2;

 public:
  int pub = 3;

  // function to access private member
  int getPVT() {
   return pvt;
  }
};
```

```
class PublicDerived : public Base {
 public:
  // function to access protected member from Base
  int getProt() {
   return prot;
  }
};

int main() {
 PublicDerived object1;
 cout << "Private = " << object1.getPVT() << endl;
 cout << "Protected = " << object1.getProt() << endl;
 cout << "Public = " << object1.pub << endl;
 return 0;
}
```



Output

```
Private = 1
Protected = 2
Public = 3
```



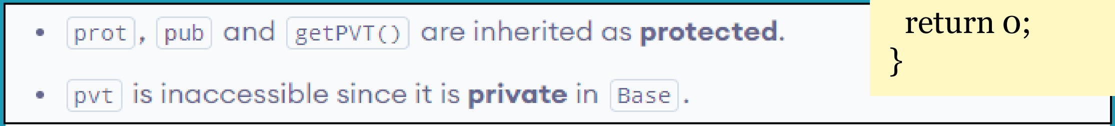
Since **private** and **protected** members are not accessible from `main()`, we need to create public functions `getPVT()` and `getProt()` to access them:

```
// Error: member "Base::pvt" is inaccessible
cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible
cout << "Protected = " << object1.prot;
```

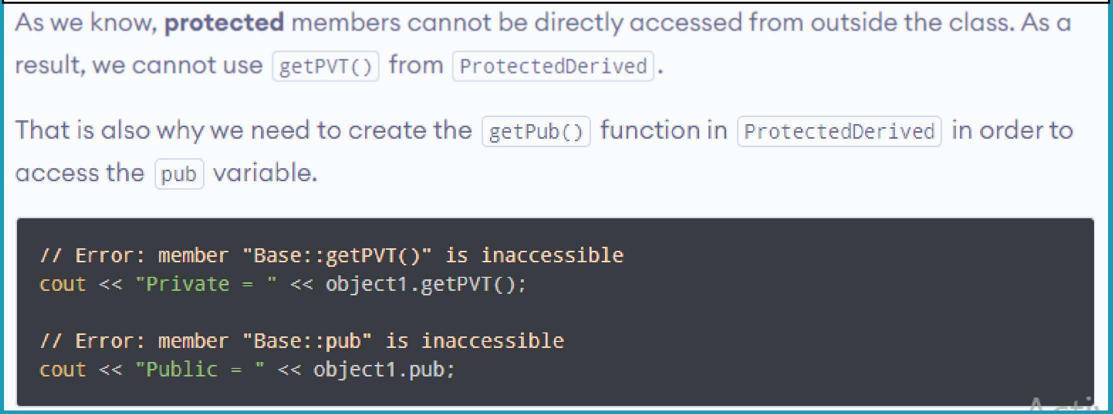
Inheritance: Protected Modifier

```
class Base {
 private:
  int pvt = 1;

 protected:
  int prot = 2;

  public:
  int pub = 3;

  // function to access private member
  int getPVT() {
    return pvt;
  }
};
```

```
class ProtectedDerived : protected Base {
 public:
  // function to access protected member from Base
  int getProt() {
    return prot;
  }

  // function to access public member from Base
  int getPub() {
    return pub;
  }
};

int main() {
 ProtectedDerived object1;
 cout << "Private cannot be accessed." << endl;
 cout << "Protected = " << object1.getProt() << endl;
 cout << "Public = " << object1.getPub() << endl;
 return 0;
}
```



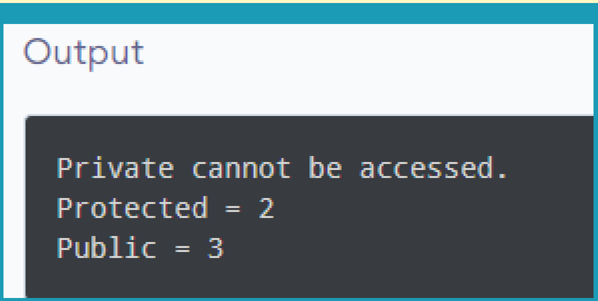
- `prot`, `pub` and `getPVT()` are inherited as **protected**.
- `pvt` is inaccessible since it is **private** in `Base`.



As we know, **protected** members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `ProtectedDerived`.

That is also why we need to create the `getPub()` function in `ProtectedDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```



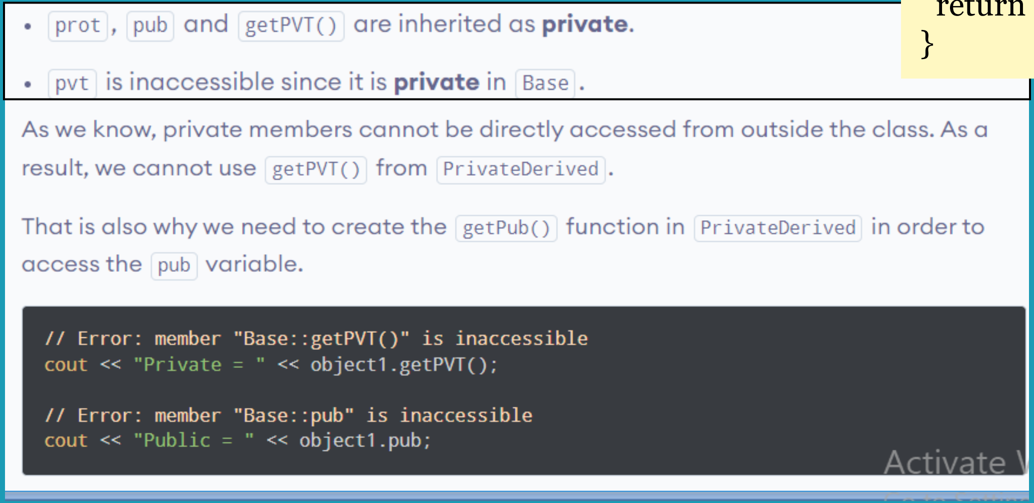
Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Inheritance: Private Modifier

```
class Base {
  private:
    int pvt = 1;

  protected:
    int prot = 2;

  public:
    int pub = 3;

    // function to access private member
    int getPVT() {
      return pvt;
    }
};
```

```
class PrivateDerived : private Base {
  public:
    // function to access protected member from Base
    int getProt() {
      return prot;
    }

    // function to access public member
    int getPub() {
      return pub;
    }
};

int main() {
  PrivateDerived object1;
  cout << "Private cannot be accessed." << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.getPub() << endl;
  return 0;
}
```



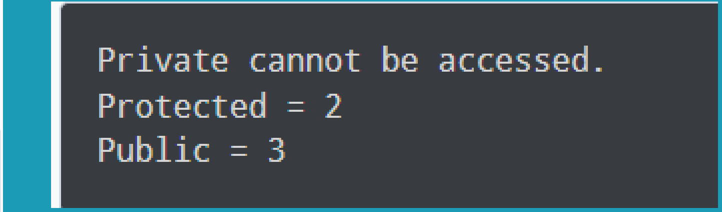
- `prot`, `pub` and `getPVT()` are inherited as **private.**
- `pvt` is inaccessible since it is **private** in `Base`.

As we know, private members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `PrivateDerived`.

That is also why we need to create the `getPub()` function in `PrivateDerived` in order to access the `pub` variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

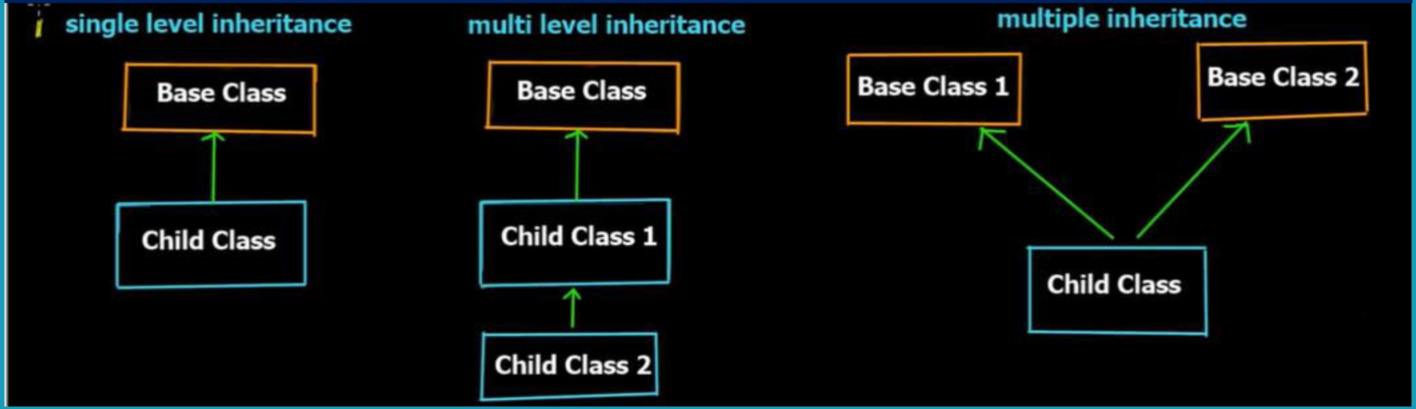
Output

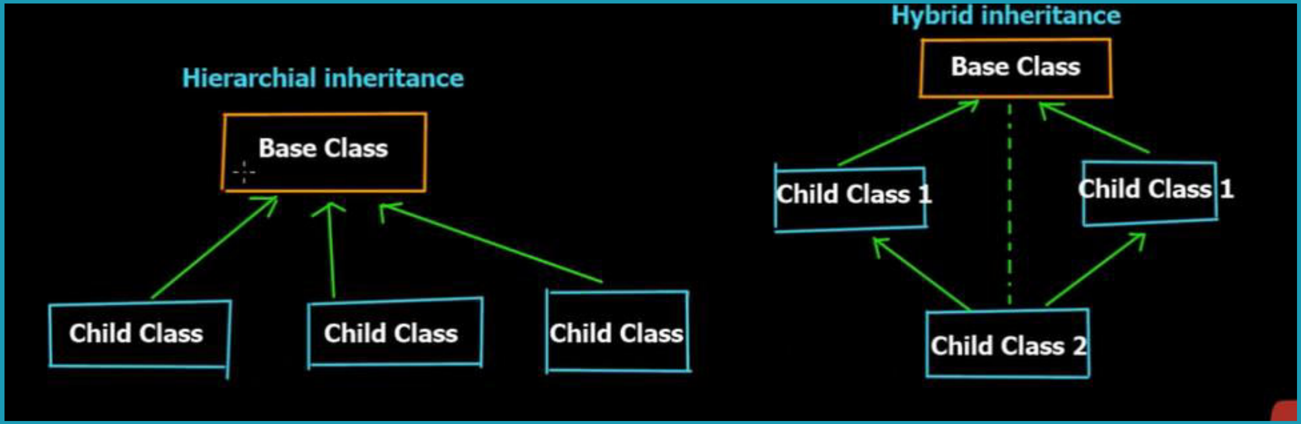


```
Private cannot be accessed.
Protected = 2
Public = 3
```

Inheritance

Types of Inheritance





Multilevel Inheritance

- It is not uncommon that a class is derived from another derived class.
- The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C.
- The class B is known as intermediate base class since it provides a link for the inheritance between A and C.
- The chain ABC is known as inheritance path.

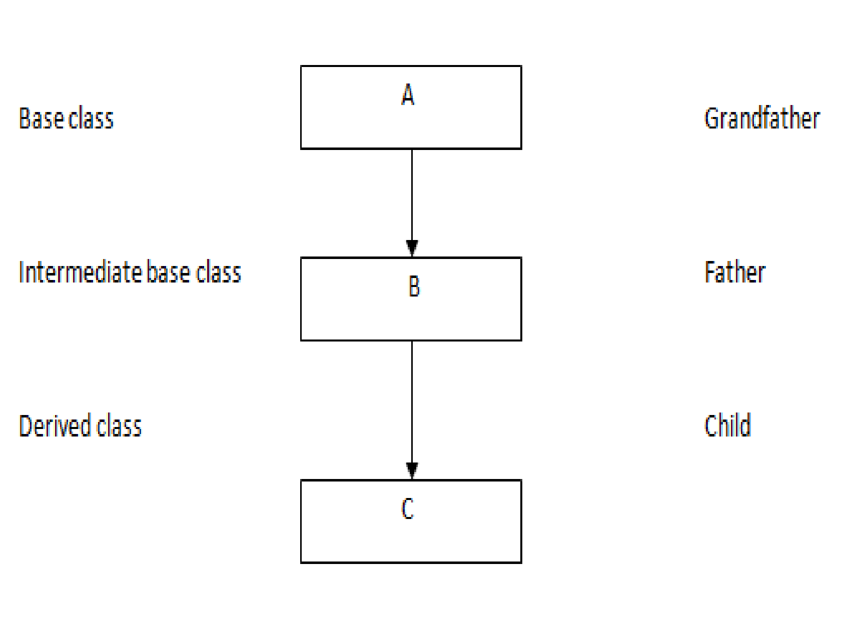
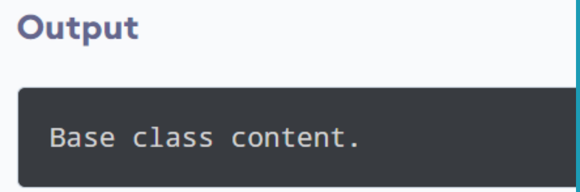


Figure: Multilevel Inheritance

Multilevel Inheritance

```
class A {
   public:
     void display() {
         cout<<"Base class content.";
     }
};

class B : public A {};

class C : public B {};

int main() {
   C obj;
   obj.display();
   return 0;
}
```

In this program, class C is derived from class B (which is derived from base class A).
The obj object of class C is defined in the main() function.
When the display() function is called, display() in class A is executed. It's because there is no display() function in class C and class B.
The compiler first looks for the display() function in class C. Since the function doesn't exist there, it looks for the function in class B (as C is derived from B).
The function also doesn't exist in class B, so the compiler looks for it in class A (as B is derived from A).

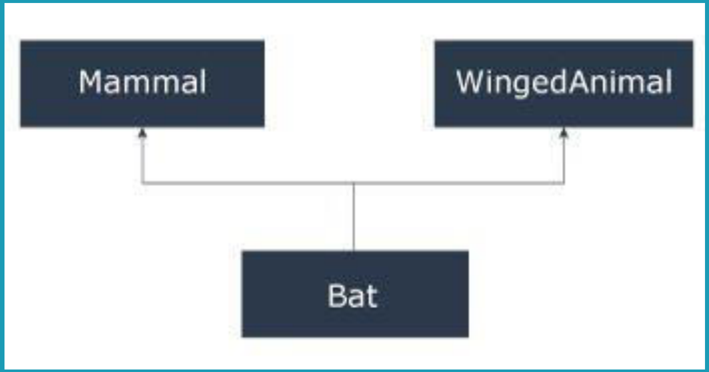


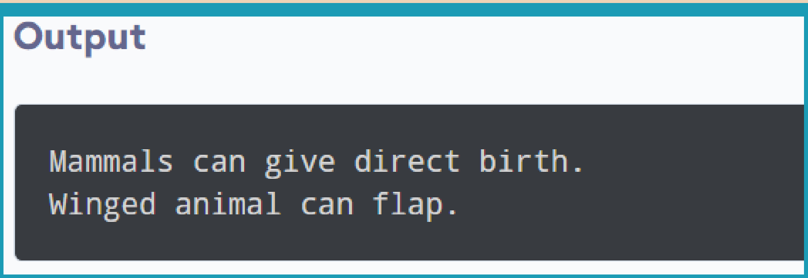
Output

```
Base class content.
```

Multple Inheritance

```
class Mammal {
  public:
    Mammal() {
      cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
  public:
    WingedAnimal() {
      cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {};

int main() {
    Bat b1;
    return 0;
}
```





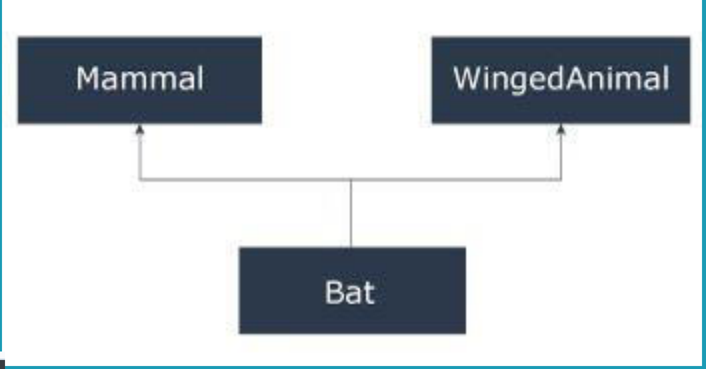
Output

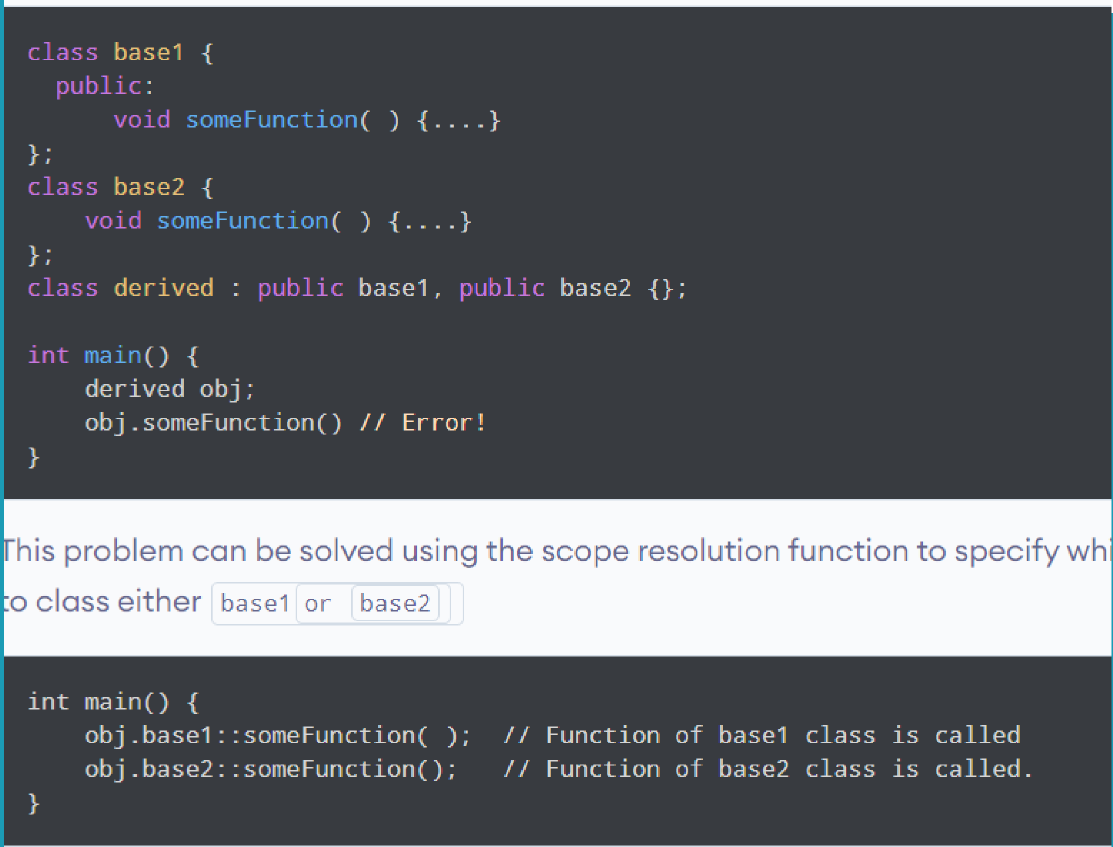
```
Mammals can give direct birth.
Winged animal can flap.
```

Multiple Inheritance

Ambiguity in Multiple Inheritance

Suppose, two base classes have a same function. If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call. For example,



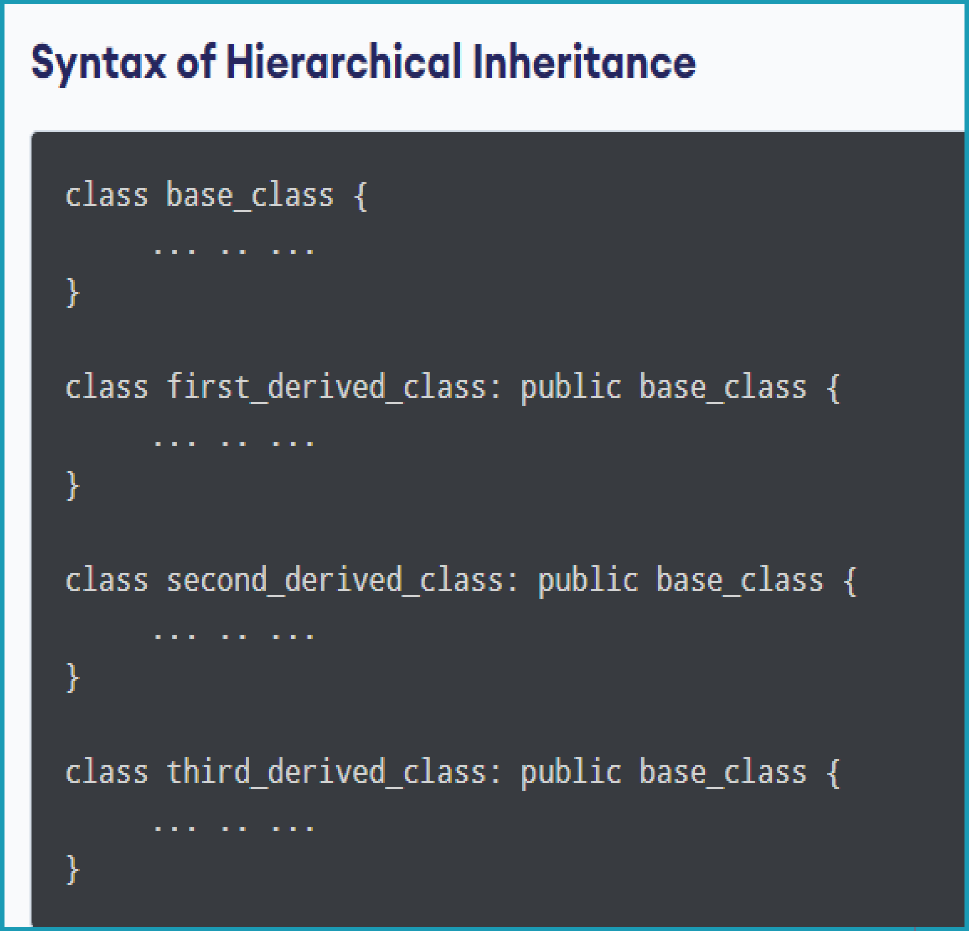


```
class base1 {
  public:
      void someFunction( ) {....}
};
class base2 {
    void someFunction( ) {....}
};
class derived : public base1, public base2 {};

int main() {
    derived obj;
    obj.someFunction() // Error!
}
```

This problem can be solved using the scope resolution function to specify whi
to class either `base1` or `base2`

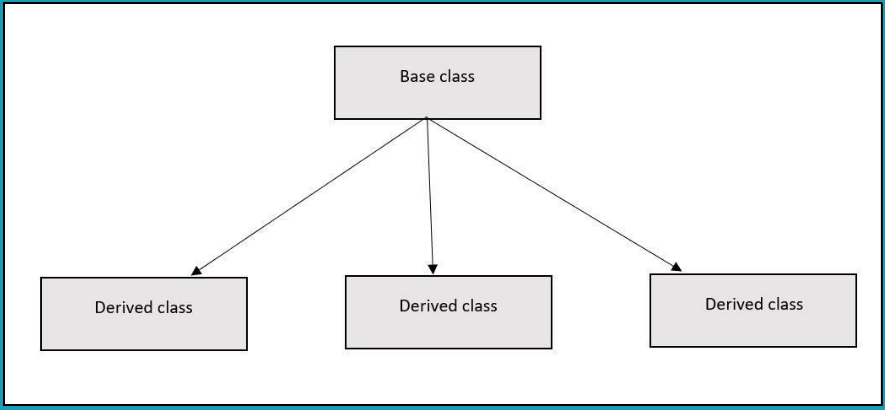
```
int main() {
    obj.base1::someFunction( );  // Function of base1 class is called
    obj.base2::someFunction();   // Function of base2 class is called.
}
```

Hierarchical Inheritance



Syntax of Hierarchical Inheritance

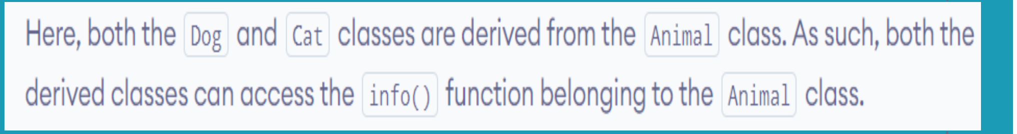
```
class base_class {
     ... .. ...
}

class first_derived_class: public base_class {
     ... .. ...
}

class second_derived_class: public base_class {
     ... .. ...
}

class third_derived_class: public base_class {
     ... .. ...
}
```



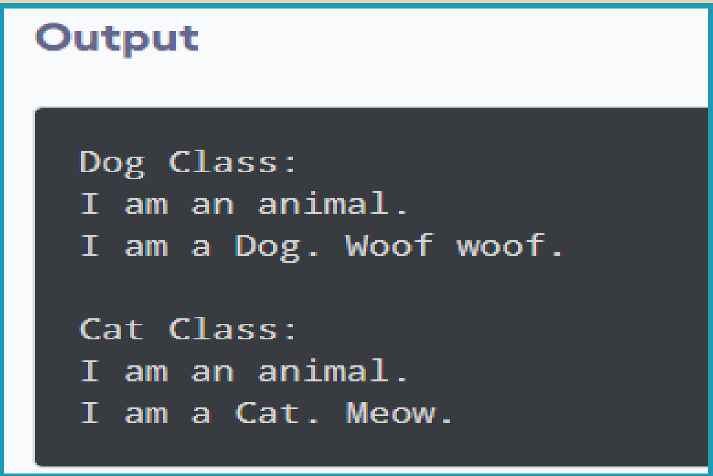
Hierarchical Inheritance

```
// base class
class Animal {
  public:
   void info() {
      cout << "I am an animal." << endl;
   }
};

// derived class 1
class Dog : public Animal {
  public:
   void bark() {
      cout << "I am a Dog. Woof woof." << endl;
   }
};

// derived class 2
class Cat : public Animal {
  public:
   void meow() {
      cout << "I am a Cat. Meow." << endl;
   }
};
```

```
int main() {
   // Create object of Dog class
   Dog dog1;
   cout << "Dog Class:" << endl;
   dog1.info();  // Parent Class function
   dog1.bark();

   // Create object of Cat class
   Cat cat1;
   cout << "\nCat Class:" << endl;
   cat1.info();  // Parent Class function
   cat1.meow();

   return 0;
}
```



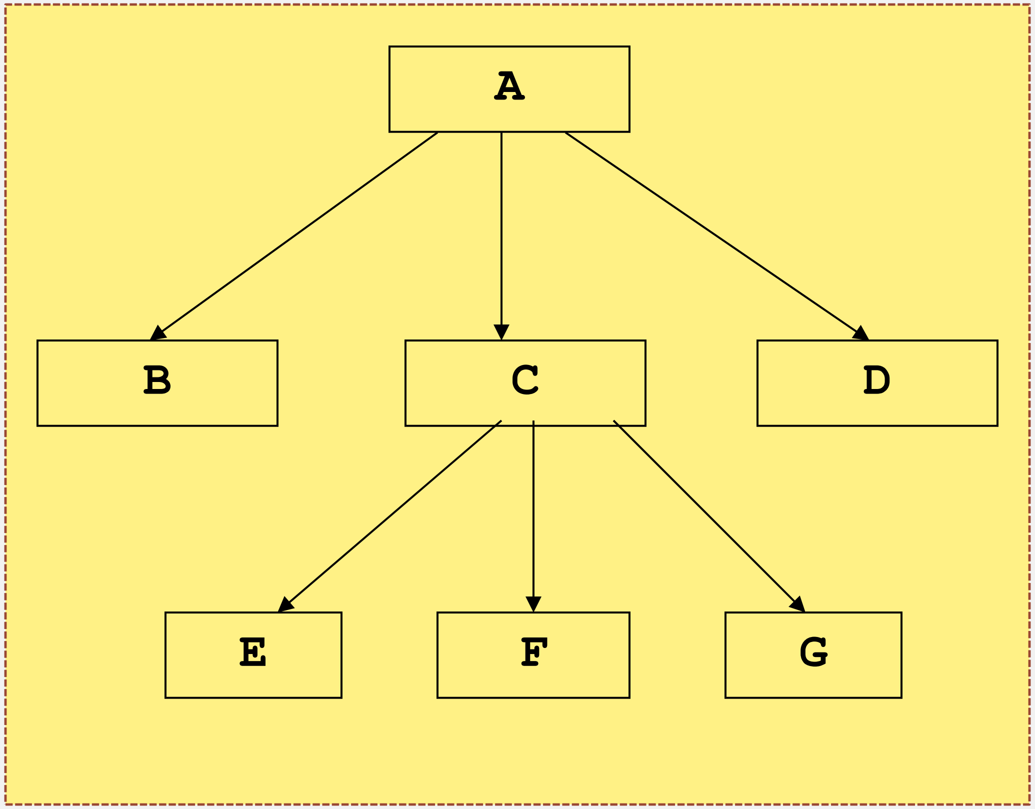
Here, both the `Dog` and `Cat` classes are derived from the `Animal` class. As such, both the derived classes can access the `info()` function belonging to the `Animal` class.



Output

```
Dog Class:
I am an animal.
I am a Dog. Woof woof.

Cat Class:
I am an animal.
I am a Cat. Meow.
```

Inheritance: Hierarchical



Class A {……};

Class B : public A {……};

Class C : public A {……};

Class D : public A {……};

Class E : public C {……};

Class F : public C {……};

Class G : public C {……};

Constructor Calling Order in Inheritance

- When any object of child class is created, parent constructor is always called before the child constructor.

- When the parent has more than one constructor, then the child constructor will call the default constructor of parent if not *specifically* mentioned.

Constructor Calling Order in Inheritance

```
class A {
     int a;
public:
     A()
     { cout<<"default A\n"; }
     A(int a)
     {
              this->a=a;
              cout<<"non-default A = “;
              cout<<a<<endl;
     }
     ~A()
     { cout<<"destructor A\n"; }
};
```

```
class B:public A {
     int b;
public:
     B()
     { cout<<"default B\n“;}

     B(int a,int b)
     {
              this->b=b;
              cout<<"non-default B=“;
              cout<<b<<endl;
     }
     ~B()
     { cout<<"destructor B\n"; }
};
```

```
int main()
{
     A aa;
     A aa1(5);
     B bb;
     B bb1(10,20);
     return 0;
}
```

Output:

```
default A
non-default A = 5
default A
default B
default A
non-default B=20
destructor B
destructor A
destructor B
destructor A
destructor A
destructor A
```

Constructor Calling Order in Inheritance

```
class A
{
    int a;

public:
    A()
    {
            cout<<"default A\n";
    }
    A(int a)
    {
            this->a=a;
            cout<<"non-default A = “;
            cout<<a<<endl;
    }
    ~A()
    {
            cout<<"destructor A\n";
    }
};
```

```
class B:public A
{
    int b;

public:
    B()
    {
            cout<<"default B\n";
    }
    B(int a,int b):A(a)
    {
            this->b=b;
            cout<<"non-default B=“;
            cout<<b<<endl;
    }
    ~B()
    {
            cout<<"destructor B\n";
    }
};
```

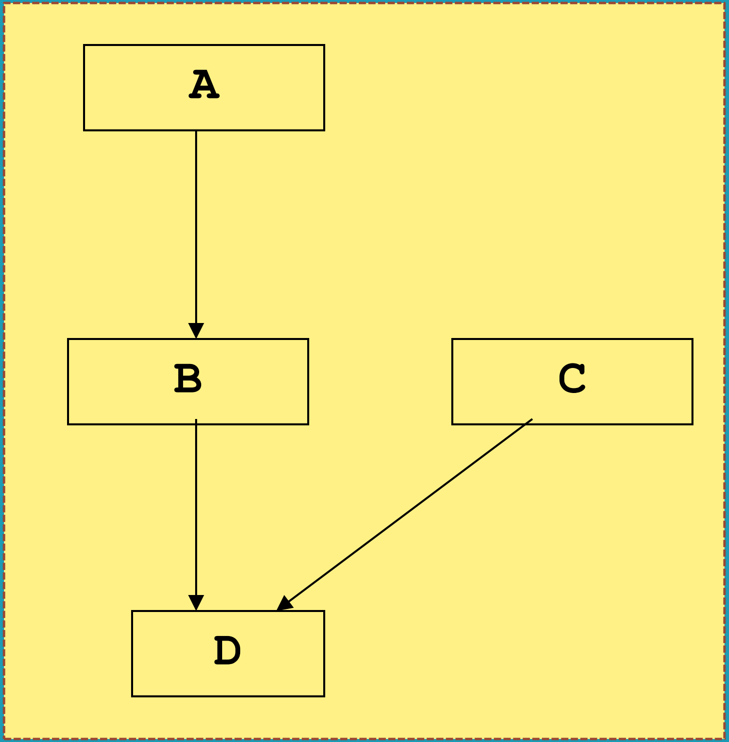
Constructor Calling Order in Inheritance

```
int main()
{
   A aa;
   A aa1(5);
   B bb;
   B bb1(10,20);
   return 0;
}
```

Output:

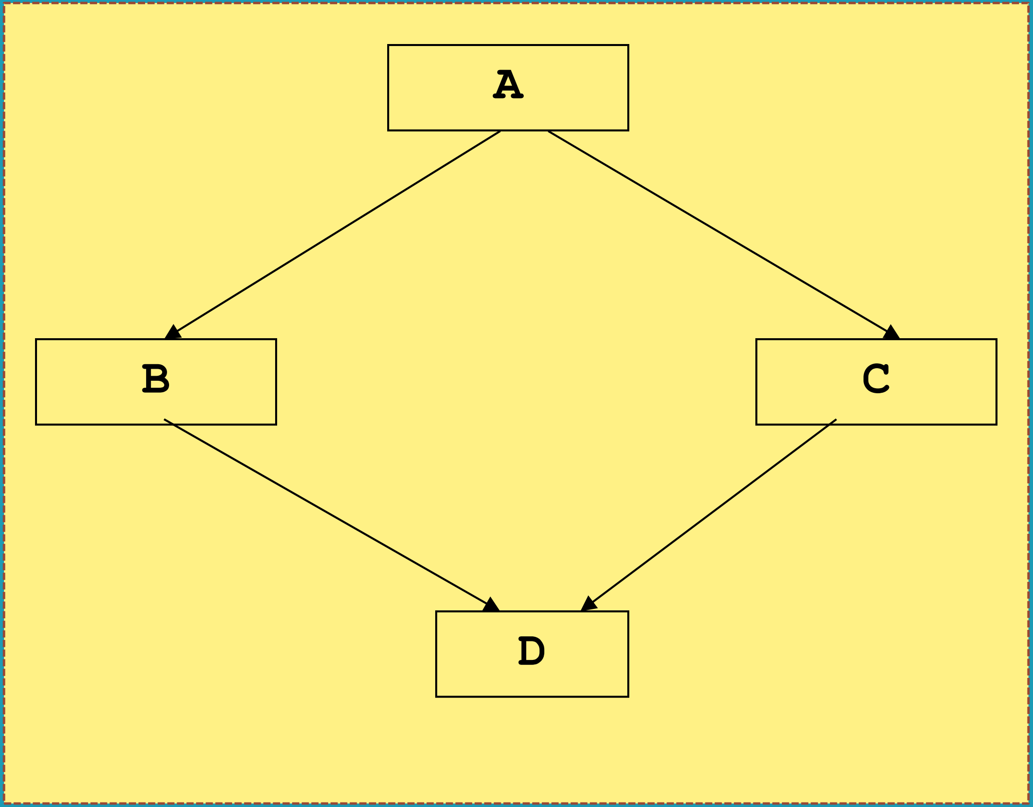
```
default A
non-default A = 5
default A
default B
non-default A = 10
non-default B=20
destructor B
destructor A
destructor B
destructor A
destructor A
destructor A
```

Inheritance: Hybrid



```
Class A {……};

Class B : public A {……};

Class C  {……};

Class D : public B, public C
 {……};
```

Inheritance: Hybrid



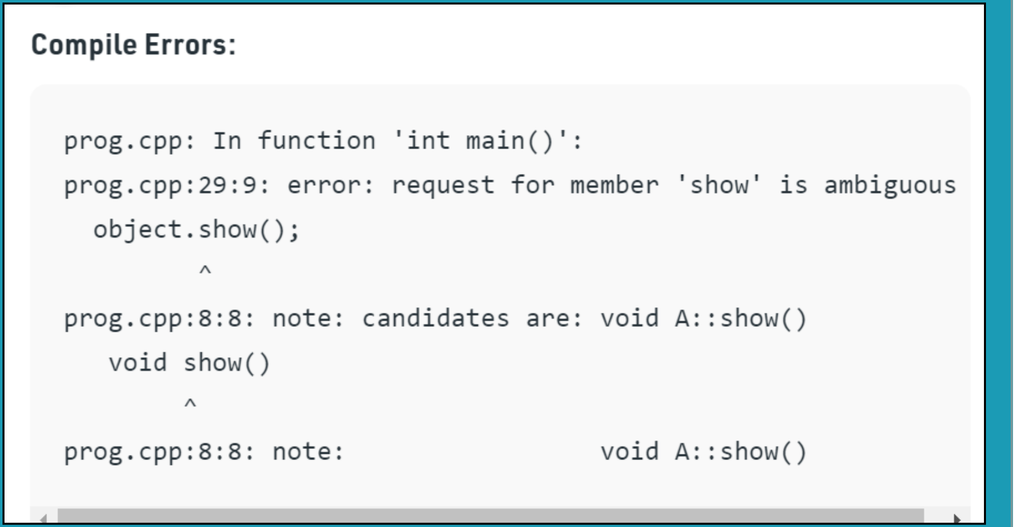
```
Class A {……};

Class B : public A {……};

Class C : public A  {……};

Class D : public B, public C
 {……};
```

```
D d;
```

d has two parents B & C
So which parent is to be
 used to access the Members
 of A (Ambiguity)

Inheritance: Hybrid

```
class A {
public:
   void show()
   {
      cout << "Hello form A \n";
   }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
   D d;
   d.show();
}
```

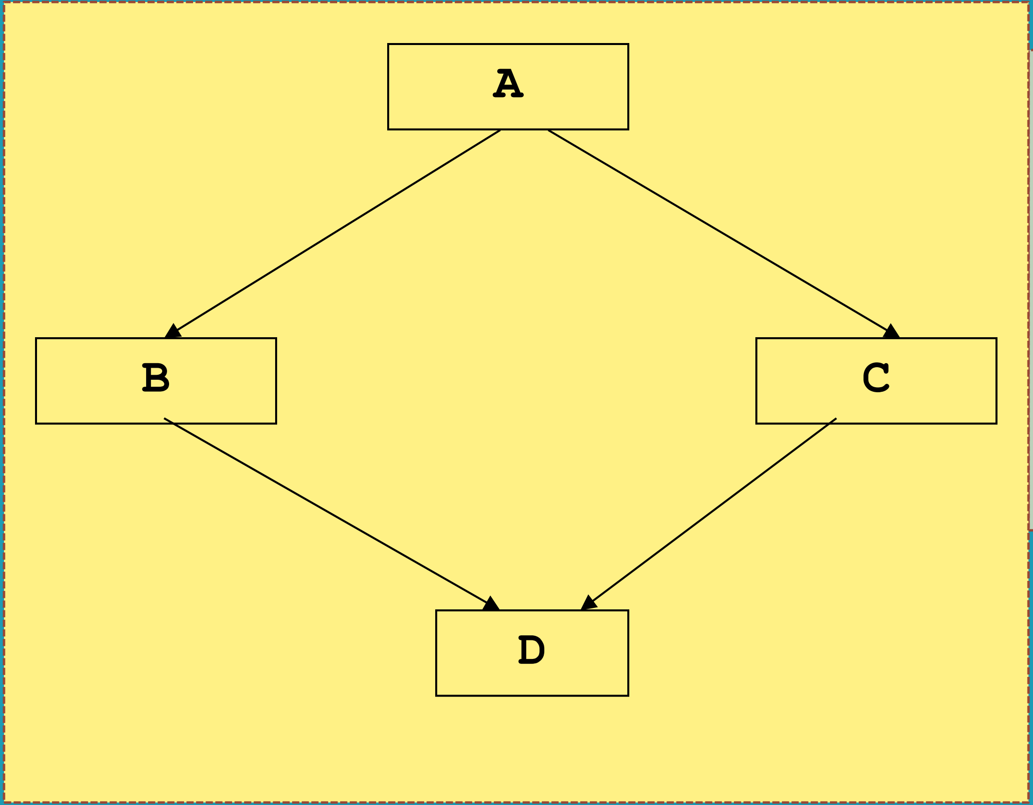


Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for member 'show' is ambiguous
  object.show();
         ^
prog.cpp:8:8: note: candidates are: void A::show()
   void show()
        ^
prog.cpp:8:8: note:                 void A::show()
```

Inheritance: Hybrid

Solution: virtual class



Class A {……};

Class B : virtual public A {…};

Class C : virtual public A {…};

Class D : public B, public C {……};

D d;

Inheritance: Hybrid

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

class A{
 protected:
     int x;
};
class B:public A{
    public:
        B(){
            cout<<“B constructor is called"<<endl;
            x=1;
        }
};
class C:public A{
    public:
        C(){
            cout<<"C constructor is called"<<endl;
            x=2;
        }
};
class D:public B,public C{
    public:
      void getX(){
        cout<<"x="<<x;
      }
}; int main()
{
   D d;
   d.getX();
}
```

When it is complied it produces an ambiguity error. When object d is created then constructor B & C are called. In B the value of x set to 1 and in C it is set to 2. Then which value will be available in class D? Complier can’t solve it. Now one solution is to provide the parent class using :: operator like

```
  cout<<"x="<<B::x;    or
  cout<<"x="<<C::x;
```

Another Solution is to use virtual keyword when parent class is created.

Virtual Class

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

class A{
 protected:
     int x;
};
class B: virtual public A{
    public:
        B(){
            cout<<“B constructor is called"<<endl;
            x=1;
        }
};
class C: virtual public A{
    public:
        C(){
            cout<<"C constructor is called"<<endl;
            x=2;
        }
};
class D:public B,public C{
    public:
      void getX(){
        cout<<"x="<<x;
      }
}; int main()
{
   D d;
   d.getX();
}
```

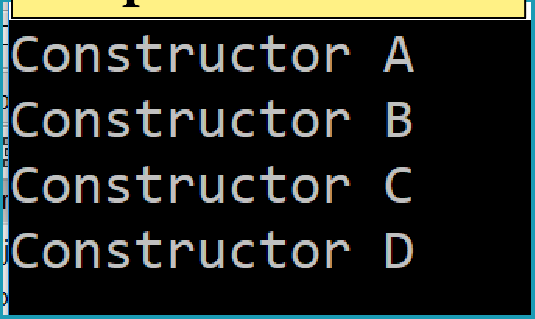
Here the classes B and C are inherited from A virtually by using virtual keyword.

Example: virtual class

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

class A {
public:
   A(){
    cout<<"Constructor A"<<endl;
   }
   void show()
   {
       cout << "Hello form A \n";
   }
};

class B : public virtual A {
   public:
   B(){
    cout<<"Constructor B"<<endl;
   }
};

class C : public virtual A {
   public:
   C(){
    cout<<"Constructor C"<<endl;
   }
};
```

```
class D : public B, public C {
   public:
   D(){
    cout<<"Constructor D"<<endl;
   }
};

int main()
{
   D d;
}
```

Output



```
Constructor A
Constructor B
Constructor C
Constructor D
```

Example: virtual class

```
#include <iostream.h>
class A{
protected:
            int ax;
public:
            void setA(int x){ ax=x; }
};
class B : virtual public A{
protected:
            int bx;
public:
            void setB(int x){ bx=x; }
};
class C : virtual public A{
protected:
            int cx;
public:
            void setC(int x){ cx=x; }
};
class P:public C, public B{
public:
            int volume(){ return ax*bx*cx; }
};

int main(void){
P p;
p.setA(2); p.setB(3); p.setC(4);
cout<<“volume="<<p.volume()<<"    ";
}
```

Output:

```
volume=24
```

THANK YOU