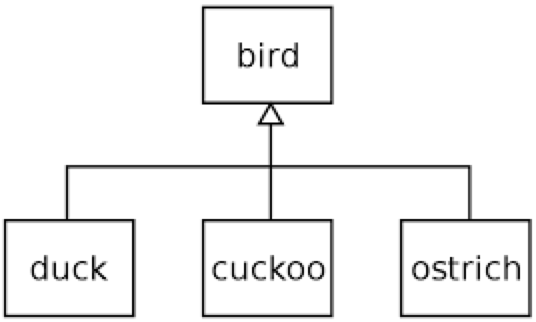
Inheritance

Introduction

Inheritance is a useful concept of OOPS supported in C++. By using this concept, we may define a new class in such a way that it automatically includes member data and member functions of an existing class. The existing class whose features are inherited in new class is called the "base class", or "parent class" or the "super class". The new class which inherits the features of existing class is called the "child class" or the "derived class" or "sub-class"



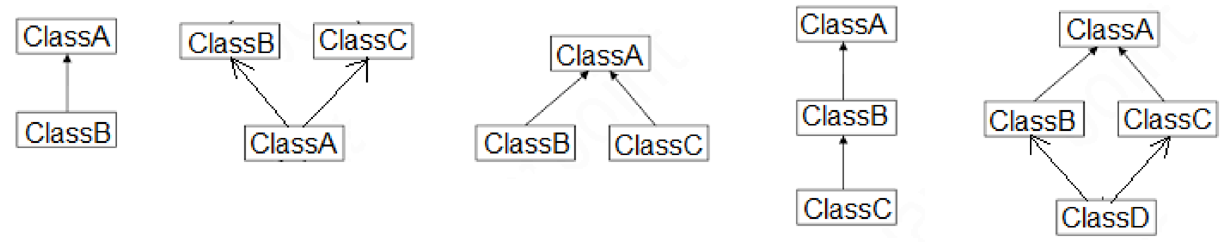
In the figure, the duck, cucoo and ostrich are birds and hence inherit the features of bird class. In addition, these derived classes will also have their own features.

The derived or the child or the sub-class may additionally have its own member data and member functions. Thus, the inheritance allows the reusability of the code of an existing class, thus saving time and efforts of writing the programs since the base class has already been tested and debugged and used many times. Inheritance, when used to modify and extend the capabilities of the base classes, becomes a very powerful tool for incremental program development. The syntax for inheriting the features of the base class into the derived class is :

```
Class <derived_class_name> : <access_specifier> <base_class>
{

// other declarations of member data and member functions
};
```

Forms of Inheritance:

Inheritance may be of different forms as discussed below:

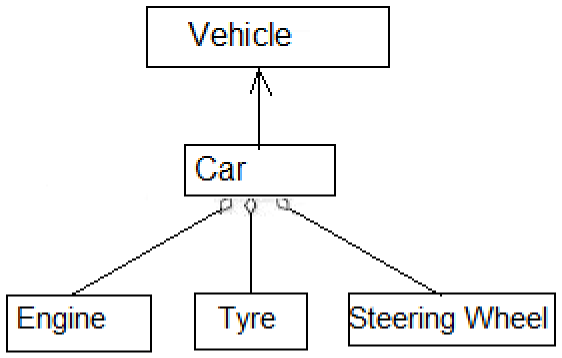


(i) Single level Inheritance (ii) Multiple Inheritance (iii) Hierarchical Inheritance (iv) Multilevel Inheritance (v) Hybrid Inheritance

i. Single level inheritance: In this form, a single class is derived from the base or parent class.
ii. Multiple Inheritances: When a single class is derived from many base or parent class, the inheritance is called multiple inheritance.
iii. Hierarchical Inheritance: In this form of inheritance, the traits of the base or parent class is inherited by more than one class.

iv. Multilevel Inheritance: When a class is derived from another derived class, the inheritance is called multilevel inheritance.

v. Hybrid Inheritance: It is a combination of multiple and hierarchical inheritance where a base class derives multiple derived class which in turn derive a single derived class.



Note down the arrows in the diagram.

Inheritance implements an “is-a” relationship. That is, a derived class is a type of the base class just like cuckoo “is-a” type of bird (base class) or say an aircraft “is-a” type of vehicle (base class). Contrast this to containership that implements “has-a” relationship. A class may contain an object of another class or a pointer to a data structure that contains a set of objects of another class. Such a class is known as a container class. For example, referring to figure……… a car “is-a” type of vehicle but “has-a” relationship as it “has-a” an engine, tyres and steeringwheel etc.

Defining the derived class:

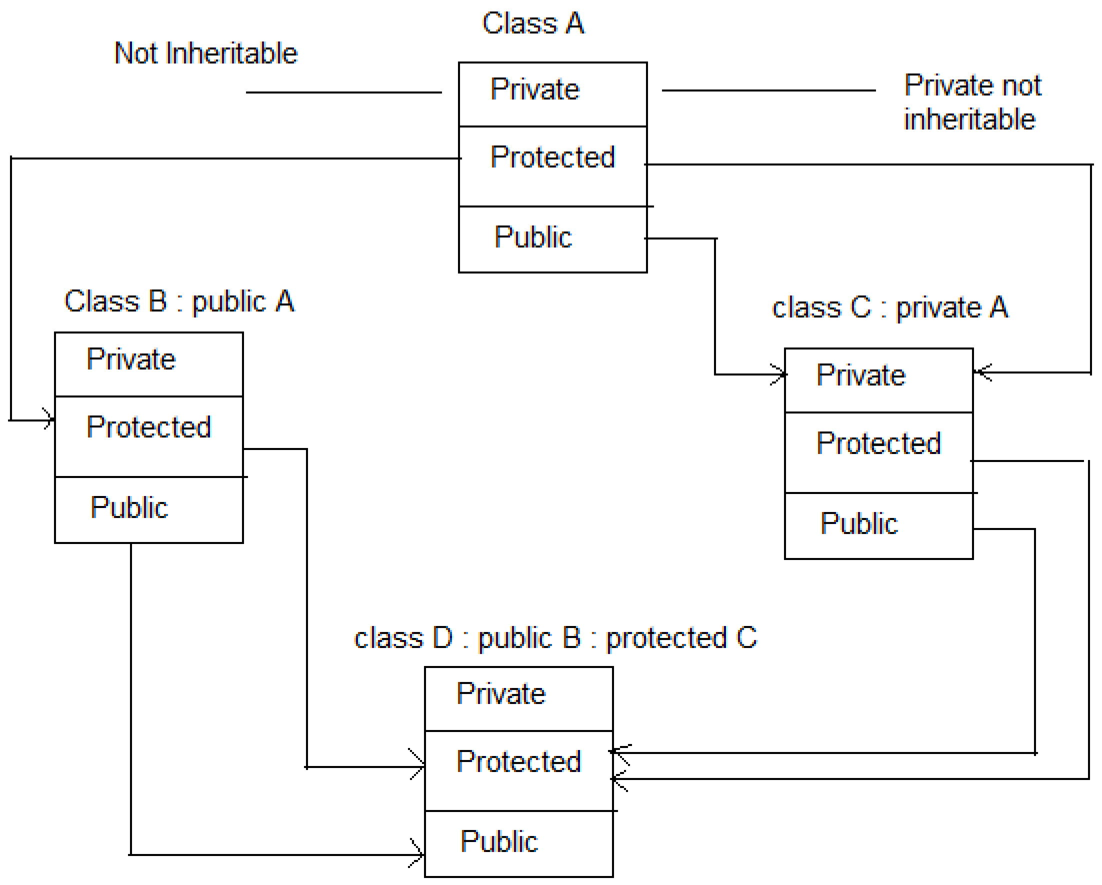
The derived class has a relationship with the base class from which it is derived. This relationship is indicated by the colon(:). The access specifier / or also called the visibility mode is optional, if not specified then visibility of base class is private by default, or the visibility mode may be specified as public, private or protected.

```
Class <derived_class_name> : <access_specifier> <base_class>
{

// other declarations of member data and member functions
};
```

Role of visibility-mode/ access specifiers :

How is base class Accessed in derived class	Visibility of public members in derived class	Visibility of private members in derived class	Visibility of protected members in derived class
Base class is Publicly inherited	All pubic member data and member function become public to derived class. Therefore, they are also accessible to the objects of the derived class	Not accessible in derived class, but may be accessed indirectly by the public member function of base class called in derived class public member functions	protected member of base class become protected in derived class too, and thus accessible by the member function of derived class
Base class is privately accessed	All pubic member data and member function of base class become private in derived class. Thus the public members of base class can be accessed by the	Private members of the base class are not inherited and therefore the private member of base class will never become members of its derived class.	A protected member will become private in derived class, although it Is available to member function of derived class, but is not

	member functions of the derived class, but such functions are not accessible through the objects of derived class		available for further inheritance(since private member cannot be inherited)
Protected	The public member(data and function) of base class become protected in derived class, hence accessible by methods of derived class	Private member of base class of are not inheritable	Protected member of base class become protected in derived class also

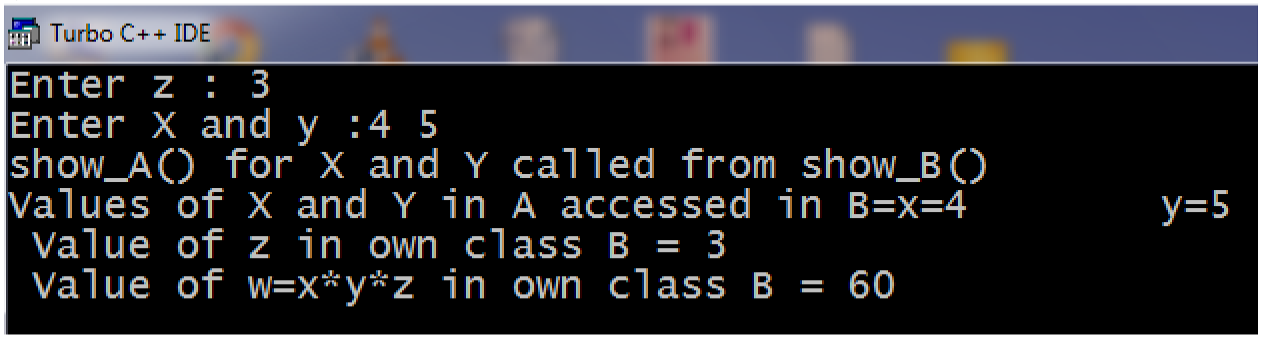


1. Single Inheritance using public visibility mode of base class

```
#include<iostream.h>
#include<conio.h>
class A
{
 int  x;
 public:
       void get_x();
       int show_A();
 };
  void A :: get_x()
  {
        cout <<"Enter x : ";
        cin >> x;
}
int A::show_A()
{
      return x;
}
class B :public A
{
       int  y;
public:
      void get_y();
      void show_B();
};
      void B :: get_y()
        {
               cout <<"Enter y : ";
               cin >> y;
        }
        void B::show_B()
        {
               cout <<" \n Value of X in B = "<<show_A();    // class A function used to return the value of x
               cout <<" \n Value of y in B = "<<y;
        }
   void main()
   {
         clrscr();
         B objB;
         objB.get_x();                  //class A public function is accessible by object of derived class
         objB.get_y();
         objB.show_B();
         getch();

  }
```

2. **Single inheritance with private visibility mode of base class**

```
#include<iostream.h>
#include<conio.h>
class A
{
       int  x;
  public:
       int y;
       int get_x();
       void get_xy()
       {
          cout <<"Enter X and y :";
          cin >> x>>y;
       }
       void show_A();
 };
  int A :: get_x()
  {
       return x;
  }
  void A::show_A()
  {
    cout<<"\nValues of X and Y in A accessed in B="<<"x="<<x<<"\t"<<"y="<<y;
  }
  class B :private A
{
 int  z;
 public:
      int w;
      void get_z();
      void mult()
      {
           get_xy();         //y will be visible here
        w = z*y*get_x();

      }
       void show_B();
  };
       void B :: get_z()
       {
             cout <<"Enter z : ";
             cin >> z;
       }
       void B::show_B()
       {
            cout<<"show_A() for X and Y called from show_B()";
            show_A();                                    //show_A() in B is private visibility, hence called from here
             cout <<" \n Value of z in own class B = "<<z;
             cout <<" \n Value of w=x*y*z in own class B = "<<w;
    }
  void main()
  {
            clrscr();
             B objB;                            //object of class B declared
              //objB.get_xy();     //won't work here
              //objB.get_x();       //won't work here
              objB.get_z();
              objB.mult();
              objB.show_B();
              getch();

  }
```



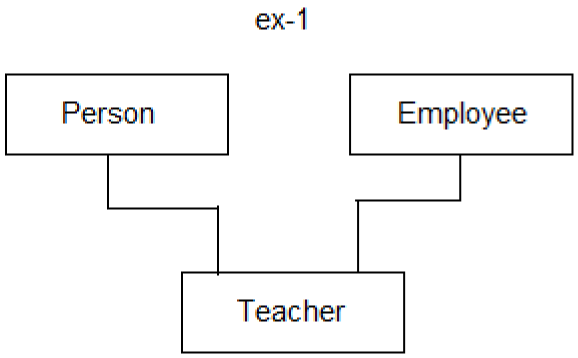
```
Enter z : 3
Enter X and y :4 5
show_A() for X and Y called from show_B()
Values of X and Y in A accessed in B=x=4        y=5
 Value of z in own class B = 3
 Value of w=x*y*z in own class B = 60
```

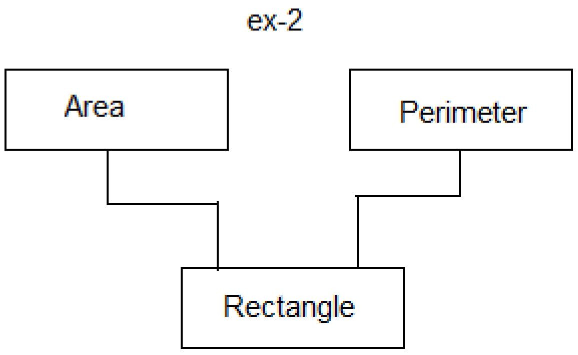
Protected class
As we have understood by now is that a private member is not available to derived class directly. We may make the members visible by making them public, but then they will be visible to all the methods of the derived class which we do want. Another method is by changing them to protected visibility.

Multiple inheritances

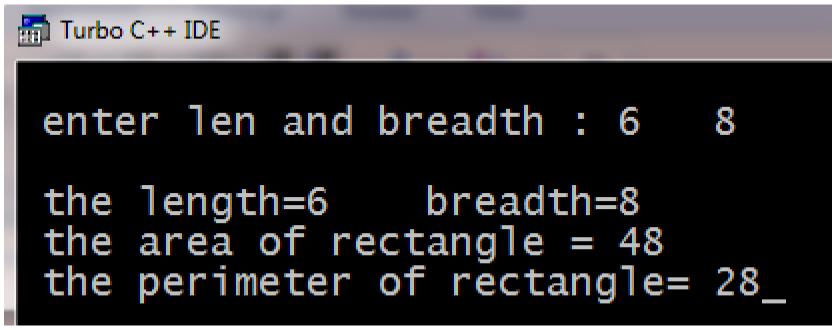
Figures on the right show the example of the multiple inheritance. A common problem in this type of inheritance is that if we have a function with the same name in both the classes area and perimeter in the code given below, which display function is refered to by an object of derived class. To resolve the ambiguity, we use the class resolution operator when referring to such function. See the code below for this.

```
#include<iostream.h>
#include<conio.h>
class area
{
public:
int cal_area(int len, int wide)
{
            return len*wide;
}
};
class perimeter
{
public:
 int cal_perimeter(int len, int wide)
 {
            return 2*(len+wide);
 }

};
class rectangle: private area, private perimeter
```





```
{
  int len, breadth;
  int    perimeter, area;
  public:
  rectangle()
  {
               len=0;
               breadth=0;
  }
  void get_data()
  {
               cout<<"\n enter len and breadth : ";
               cin >> len >> breadth;
  }
  void cal_area()
  {
               area = area::cal_area(len, breadth);
  }
  void cal_peri()
  {
               perimeter = perimeter::cal_perimeter(len, breadth);
  }
  void disp()
  {
                cout<< "\n the length="<<len<<"\t breadth="<<breadth;
                cout<<"\n the area of rectangle = "<<area;
                cout <<"\n the perimeter of rectangle= "<<perimeter;
  }

};
void main()
{
             clrscr();
             rectangle R;
             R.get_data();
             R.cal_area();
             R.cal_peri();
             R.disp();
             getch();

}
```



```
enter len and breadth : 6   8

the length=6    breadth=8
the area of rectangle = 48
the perimeter of rectangle= 28_
```

Example-2

```
#include<iostream.h>
class test
```

```
{
int marks1, marks2;
public:
void get_marks()
{
cout<<"\n enter marks :";
cin >>marks1>>marks2;
}
int marks_1()
{
return marks1;
}
int marks_2()
{
return marks2;
}
};

class sports
{
int score;
public:
void get_score()
{
cout<<"\n enter score :";
cin >>score;
}
int get_score()
{
return score;
}

};

class result  : public test, public sports
{
int result:
public:
void cal_result()
{
result = marks_1() + marks_2() + get_score();
}
void disp_result()
{
 cout<<"\n marks1="<<marks_1() <<"  marks_2()= "<<marks2 <<"  score = "<<get_score();
}

};
```

```
void main()
{
Result obj;
obj.cal_score();
obj.dis_result();
getch()

}
```