CS2: Topic1. Inheritance, Virtual Functions & Abstract classes

Author: Satish Singhal Ph. D.

Version 1.1

Table of Contents

Introduction 1
Types of Inheritance 5
  Public inheritance 5
  Public inheritance 6
Writing the Constructor for the derived classes: 10
Sequence of Constructor and Destructor calls: 11
Static Vs. Dynamic Binding and Virtual Functions: 12
Abstract Classes and pure virtual functions: 18
  Methods for the Mammal class 24
Composition Relationship 27

Introduction

When we write a C++ class, we try to model a part of the reality of the world. For example, when we write a complex number class, we model the behavior of complex numbers in that class. However, in reality the things are related. By modeling the related things together, we can see that how modeling one thing, can help us model something related to it. Using C++ mechanism of invoking inheritance relationship between classes can model reality more efficiently. Let us take an example of vehicles in the world. Every vehicle has at least one property – speed. We can write a class called **vehicle**, which has one data member called **speed**. However, there are many kinds of vehicle in the world. For example, the vehicle may be a wheeled vehicle, which will move on roads and surfaces, or it may be an airplane that will fly in air. Both vehicles such as airplane and vehicle with wheels have a relationship with general concept vehicle. We can say that an *airplane is-a vehicle*. We can also say that a *vehicle with wheels is-a vehicle*. When objects have this "*is-a*" type relationship between each other, then an inheritance relationship exists between them. Using this concept of "*is-a*" type relationship, we can set up an inheritance hierarchy between vehicles of various types as follows (Figure 1.1).

Microsoft PowerPoint Presentation

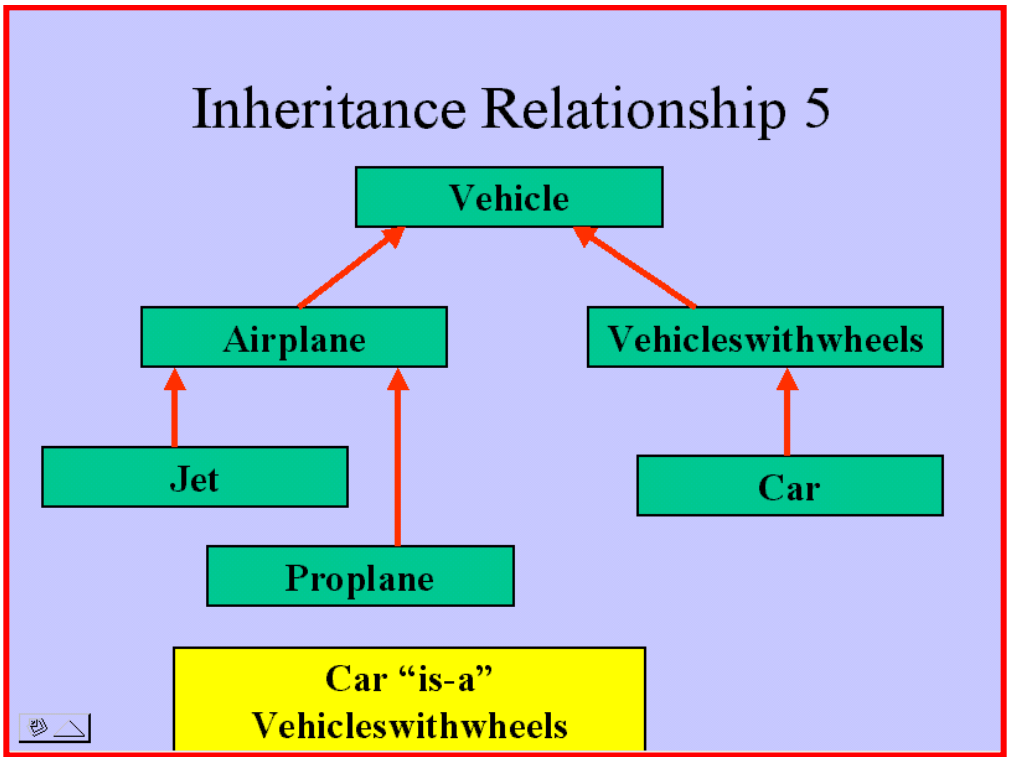


FIG. 1.1

Now, at each level of above inheritance hierarchy, there is a new property added to the system of related objects. For example, we may ask the question, as to what is the most common property to all vehicles? Obvious answer is that all vehicles have speed. So now, we take this system of inheritance relationships and start filling it with properties. This is dynamically presented by the presentation below.

We can see that all vehicles have a property speed, and all airplanes have a property called "engine thrust". We keep adding one or more new property at each level of inheritance derivation in our system. The Figure 1.2 below shows the result.

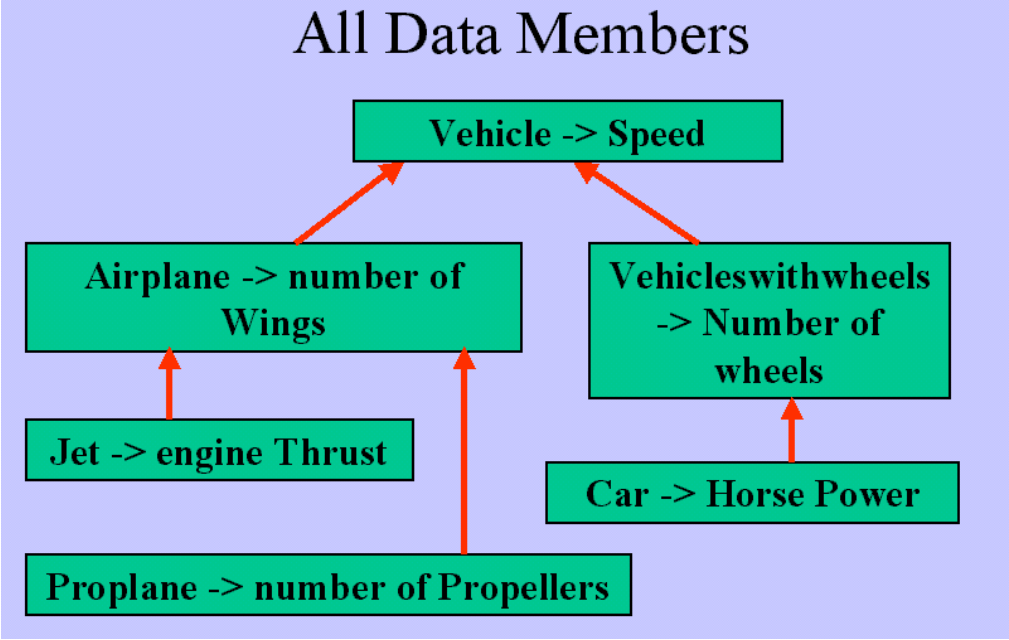


FIG. 1.2(-> shows property added at each level of inheritance hierarchy)

Based on the inheritance relationship we established above, and based on the property assigned at each level, now we may start writing the code for this system of related classes. However, let us look at few definitions here. These are summarized in the Figure 1.3 below.

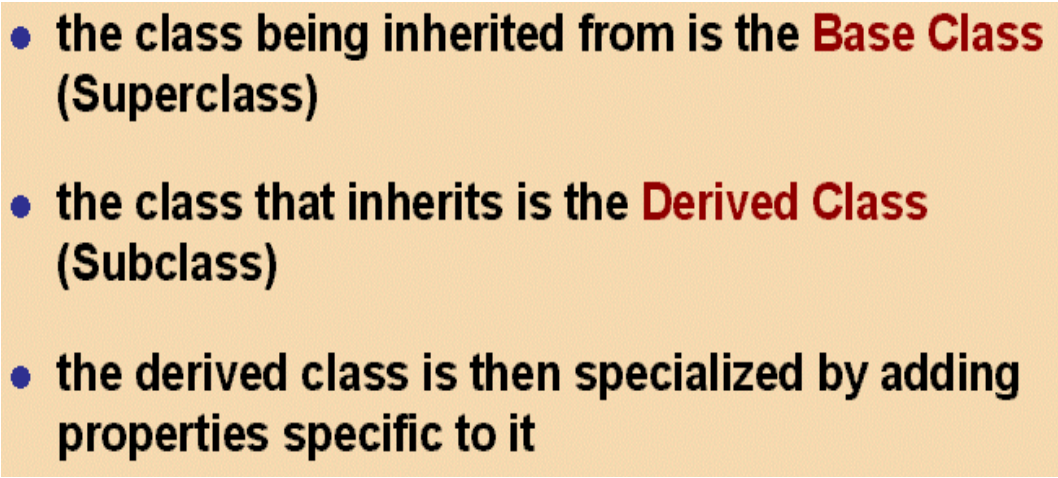


FIG. 1.3 (Taken from Prof Nell Dale Slides)

At each level of inheritance, the class from which the inheritance is done is called a base class or a superclass. Moreover, the class that does the inheritance is called a derived class or subclass. The phrase subclass should not be taken to mean that somehow the derived class is "lesser" in some way. In fact, if anything it has more data members. In the present system that we have built here, the Vehicle class is the base class or superclass, where as the with respect to the vehicle class, the class Airplane is a derived class or subclass. We illustrate that in the Figure 1.4 and 1.5

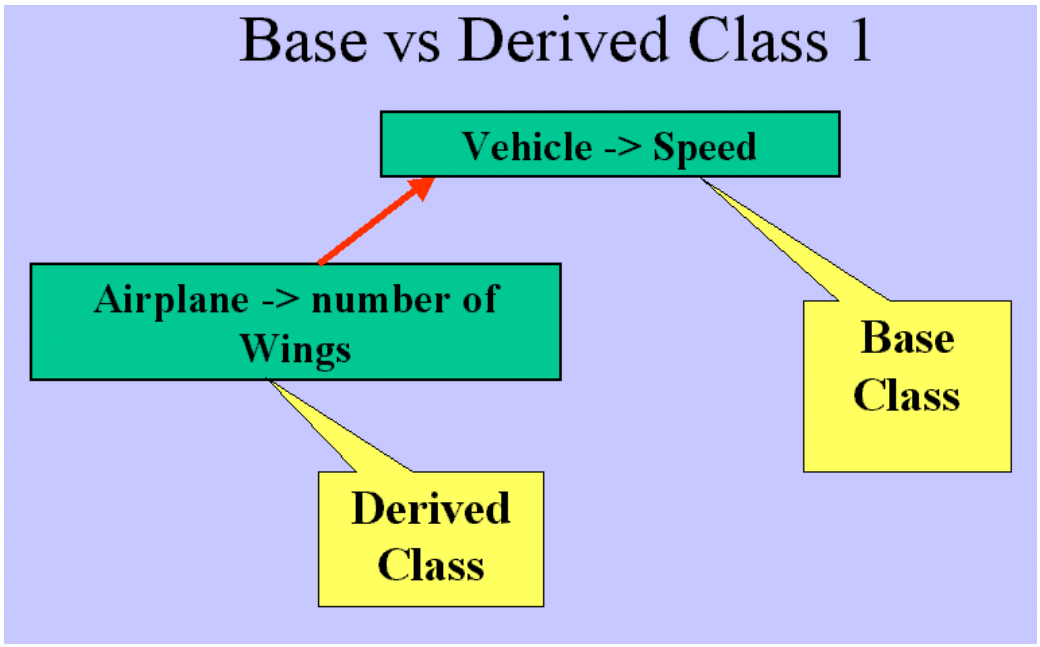


FIG. 1.4

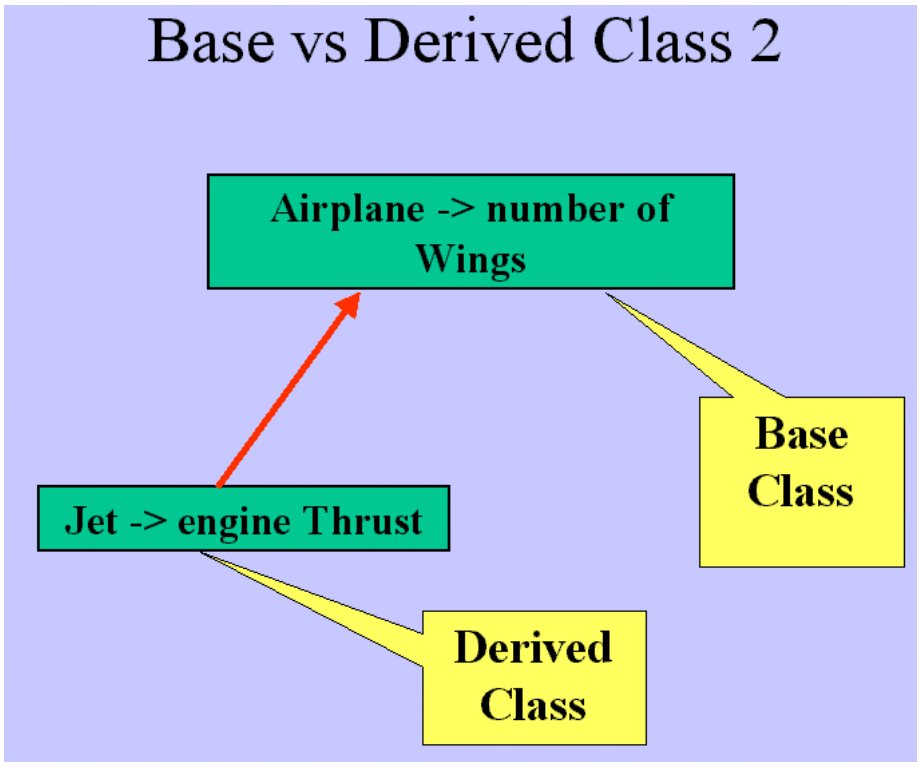


FIG. 1.5

Each derived class automatically inherits all members of all of its super classes. For example in Figures 1.4 and 1.5, the super classes to Jet are Airplane and Vehicle. Thus Jet automatically has three data members: speed, number of wings, and engine thrust. In these three data members in Jet class, the first two (speed, and number of wings) are

inherited, and engine thrust is declared in its own body. Thus inherited members behave *as if* a derived class declared them in its own body[1].

Types of Inheritance

There are three kinds of inheritance relationships in C++, which could be established between the base class and a derived class. These are:

1. **Public inheritance**
2. **Protected inheritance**
3. **Private inheritance**

Taking the example of the system of inheritance we built in the Figure 1.2, we show the syntax for establishing the inheritance relationship between classes below (Figure 1.6).

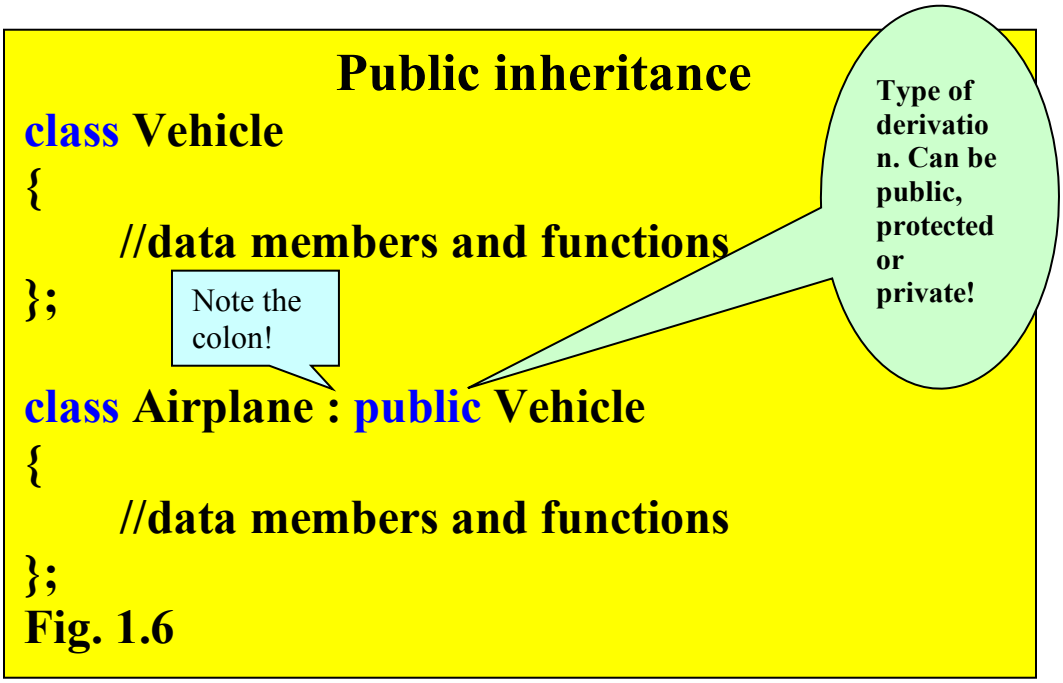


Fig. 1.6

Most useful form of inheritance derivation is the public inheritance. Therefore, we stay with this type of derivation for the moment. Note that so far we declared most data members in our classes as private. The characteristic of private data members is that, they are visible to the functions in that class and to the friend functions and classes, but invisible to rest of the world and other classes. The establishment of inheritance, with all data members being private complicates the matter somewhat. This complication is that though private members of base class are members of derived classes. Yet they are not visible inside the derived classes. Private

[1] **Understand that reverse is not true. The derived class members are not visible in base classes! Membership by inheritance mechanism only flows in one direction: top to downwards.**

data members of base classes in derived classes are like a safe that you own but you lost its key. You still own the safe but you cannot open it and you cannot see or use its contents.

C++ allows another identifier for the data members of a class, which makes them invisible to rest of the world, but still visible to derived classes. This identifier is "protected". The data members named as "protected", are available and visible to the derived classes, but are invisible to rest of the world and other classes that are not part of inheritance hierarchy. Thus we make the access level for all classes that are related by inheritance as protected. This is shown in Figure 1.7 for classes Vehicle, Airplane, and Jet. For example the speed data member of Vehicle class is visible to and *"is-a"* member of classes Airplane and Jet but is invisible to main function and all other classes that are not part of inheritance hierarchy.

Similarly, the numberOfwings data member of class Airplane is visible to and *"is-a"* member of derived class Jet, but is invisible to main function and other classes.

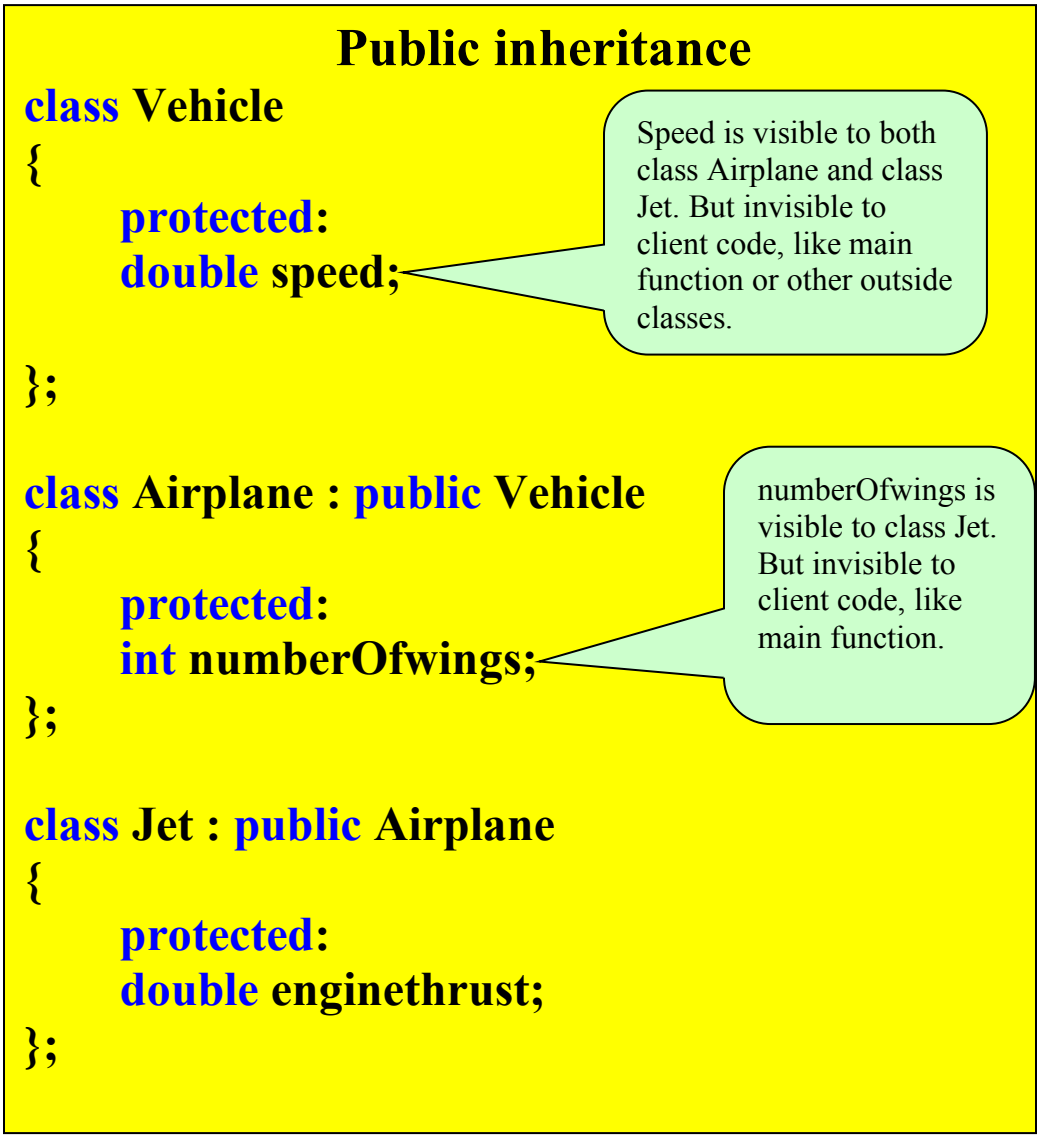


FIG. 1.7

By making the data member of class Vehicle, for example in this case, the speed protected, has the affect of class Airplane having two data members, speed and numberOfwings. Similarly, the class Jet effectively has three data members, speed, numberOfwings, and enginethrust. What we do now is that we add some data members, constructors, destructor, and a function called describe ( ) to all classes in the Figure 1.2. This results in the code given in Listing 1.1 below.

```
// VEHICLES.CPP -- Illustrates C++ inheritance
//-----------------------------------------
#include <iostream.h>
#include <afxwin.h>

class Vehicle  {
protected:
      double speed;
public:
      Vehicle( double init_speed = -99 );
      ~Vehicle();
      void describe();
} ;
//--------------------------------------
class Airplane : public Vehicle  {
protected:
      int numberOfWings;
public:
      Airplane( double init_speed = -99, int init_wings = -99 );
      ~Airplane();
      void describe();
} ;
//---------------------------------
class Jet : public Airplane  {
protected:
      double engineThrust;
public:
      Jet( double init_speed = -99, int init_wings = -99, double thrust
= -99);
      ~Jet();
      void describe();
}  ;
//---------------------------------
class PropPlane : public Airplane  {
protected:
      int numberOfProps;
public:
       PropPlane(double init_speed = -99, int init_wings = -99,
                                                              int
numProps = -99);
      ~PropPlane();
      void describe();
}  ;
//-----------------------------------
class VehiclesWithWheels : public Vehicle  {
protected:
      int numberOfWheels;
public:
      VehiclesWithWheels( double init_speed = -99, int nWheels = -99 );
```

```
      ~VehiclesWithWheels();
      void describe();
} ;
//---------------------------------------
class Car : public VehiclesWithWheels {
protected:
      int hp;
public:
      Car( double init_speed = -99,int nWheels = -99, int init_hp = -
99);
      ~Car();
      void describe();
} ;
//-----------------------------------------------
////////////////////////////////////////////////
Vehicle::Vehicle(double init_speed) : speed(init_speed)
{ cout<<("Vehicle constructor invoked\n");}
//----------------------------------------
void Vehicle::describe() { cout << "I'm an abstract Vehicle\n"; }
//----------------------------------------
Vehicle::~Vehicle()
{ cout<<("Vehicle destructor invoked\n");}
//----------------------------------------
Airplane::Airplane( double init_speed, int init_wings ) :
      Vehicle(init_speed), numberOfWings(init_wings)
{ cout<<("Airplane constructor invoked\n");}
//----------------------------------------
void Airplane::describe()
{
      Vehicle::describe();
      cout << "I'm a general Airplane.\nI travel through the air.\n";
}
//----------------------------------------
Airplane::~Airplane()
{ cout<<("Airplane destructor invoked\n");}
//-------------------------------------------------
Jet::Jet( double init_speed, int init_wings, double thrust  ) :
      Airplane(init_speed,init_wings ), engineThrust(thrust)
{ cout<<("Jet constructor invoked\n");}
//----------------------------------------
void Jet::describe()
{     Airplane::describe();
      cout << "I'm a Jet.\nI am propelled by a fast jet engine.\n"; }
//-----------------------------------------------------------
Jet::~Jet() { cout<<("Jet destructor invoked\n");}
//-------------------------------------------------
PropPlane::PropPlane( double init_speed, int init_wings, int numProps
) :
      Airplane(init_speed,init_wings ), numberOfProps(numProps)
{ cout<<("PropPlane constructor invoked\n");}
//----------------------------------------
void PropPlane::describe()
{     Airplane::describe();
      cout << "I'm a PropPlane.\nI am propellors to fly.\n"; }
//-----------------------------------------------------------
PropPlane::~PropPlane()
{ cout<<("PropPlane destructor invoked\n");}
```

```
//-------------------------------------------------
VehiclesWithWheels::VehiclesWithWheels( double init_speed, int nWheels)
      :
      Vehicle(init_speed), numberOfWheels(nWheels)
{ cout<<("VehiclesWithWheels constructor invoked\n");}
//-------------------------------------------------
VehiclesWithWheels::~VehiclesWithWheels()
{ cout<<("VehiclesWithWheels destructor invoked\n");}
//-------------------------------------------------
void VehiclesWithWheels::describe()
{ Vehicle::describe(); cout << "I use wheels to travel on the land.\n";
}
//-------------------------------------------------
Car::Car( double init_speed,int nWheels, int init_hp) :
      VehiclesWithWheels(init_speed,nWheels), hp(init_hp)
{ cout<<("Car constructor invoked\n");}
//-------------------------------------
void Car::describe()
{ VehiclesWithWheels::describe();cout << "I'm the most common wheeled
vehicle.\n"; }
//-------------------------------------
Car::~Car() { cout<<("Car destructor invoked\n");}
void main()
{
      Airplane plane;
      Car car;
      cin.get();
}
```

Listing 1.1

In Listing 1.1, the function describe ( ) for each class has a single output statement describing the type of vehicle the class represents. For example the function describe ( ) for class Vehicle is coded as follows:

```
void Vehicle::describe( )
{
      cout << "I'm an abstract Vehicle\n";
}
```

We also add a constructor and destructor to each class. Note that the constructor for the derived classes must have all the inherited data also as arguments. For example, we write the prototype for the constructor for the class Airplane as follows (Figure 1.8):

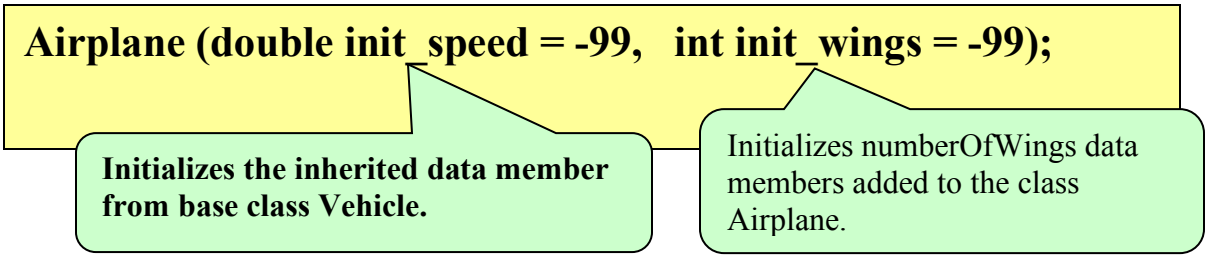


Fig. 1.8

Writing the Constructor for the derived classes:

The full definition of derived class constructors requires that a call be made to the base class constructor it derives from. For example in writing the constructor for class Airplane, we first need to make a call to the constructor for the base class Vehicle as well. The constructor definition for the Airplane class is shown below.

```
Airplane::Airplane( double init_speed, int init_wings ) :

Vehicle (init_speed),
NumberOfWings (init_wings)
{
      //Constructor body
}
```

Base class constructor called first!

Initialize the Airplane class data member next!

FIG. 1.9

Notice that by writing the constructor the way we have in the Figure 1.8, and 1.9, we are providing the default and the explicit constructor by just using one piece of code as we have provided default arguments for all constructor formal arguments. Therefore, an extra default constructor is not needed. This way of writing constructor is also more memory efficient.

When the object Airplane is instantiated in the client code, we would make a call like the one below.

```
Airplane my_plane;
```

In making the above call; first the Vehicle part of the object my_plane is built. The program making a call to the Vehicle constructor does this. Then call to the Airplane constructor is completed. You would notice that inside each constructor and destructor, we put a piece of code like:

```
cout<< ("Message……");
```

Including the header file <afxwin.h>, setting the options in project tab to "using MFC in shared DLL", and calling function cout<< ( ) from with in each constructor and destructor, allows us to see the sequence of construction and destruction of all program objects. The messages passed as an arguments to function cout<<, show up in the debug window in Visual C++.

Sequence of Constructor and Destructor calls:

It is very important to understand that how the objects of derived classes are built and destroyed. When objects are built, the calls are made to constructors. When objects are destroyed, the calls are made to destructors. Number of constructor and destructor calls must always be equal. We show this, first by instantiating the class Vehicle, which is on the top of the hierarchy, and then doing the same for derived class objects. We use the Listing 1.1, except that we change the code of main ( ) function as follows(Listing 1.2). For each case below, we show the code in the main ( ) and the output in the debug window.

```
void main()
{
    Vehicle my_vehicle;
    cin.get();
}
```

```
Vehicle constructor invoked
Vehicle destructor invoked
```

Output in debug window

Listing 1.2

One can easily see the expected result, that when the class Vehicle, which happens to be on the top of hierarchy is instantiated, then process is completed simply by calling the Vehicle call constructor. Calling the Vehicle class destructor then destroys the object my_vehicle. Now we look the construction and destruction of objects of type Airplane, which is a derived class from Vehicle (Listing 1.3).

```
void main()
{
    Airplane my_airplane;
    cin.get();
}
```

Output in debug window

```
Vehicle constructor invoked
Airplane constructor invoked
Airplane destructor invoked
Vehicle destructor invoked
```

Listing 1.3

We can see that in the above Listing, in order to create the Airplane object, first a call is made to the constructor of base class Vehicle, to construct the speed data member of object my_airplane. Then finally the call to derived class constructor Airplane is completed. More important to note is the sequence of destructor calls. The destruction occurs in the reverse sequence of construction. ***The parts of the object created last are destroyed first***. Thus in destroying the object my_airplane, the Airplane destructor is called first and then finally the vehicle destructor is called. Finally, we see the construction and destruction mechanism for an object, like an instance of Car, which is third from the top of hierarchy in the inheritance diagram (Listing 1.4).

```
void main()
{
        Car my_car;
        cin.get();
}
```

```
Vehicle constructor invoked
VehiclesWithWheels constructor invoked
Car constructor invoked
Car destructor invoked
VehiclesWithWheels destructor invoked
Vehicle destructor invoked
```

Listing 1.4

In order to build an instance of object type Car, the constructor calls take place in the following sequence: Vehicle ( ), VehicleWithWheels ( ), and finally Car ( ). The destruction takes place in exact reverse sequence.

Static Vs. Dynamic Binding and Virtual Functions:

The code in the Listing 1.1 contains a function describe ( ) for the base and for all derived classes. The purpose of function describe ( ) in each case is for the object to give its complete description, starting from the description of the class on the top of hierarchy down to the object itself. We see the functioning of describe ( ) by making calls to it for the instances of classes Vehicle, Airplane and Car (Listing 1.4).

```
void main()
{
        Vehicle my_vehicle;
        Airplane my_airplane;
        Car my_car;
        my_vehicle.describe();
        my_airplane.describe();
        my_car.describe();
        cin.get();
}
```

```
I'm an abstract Vehicle
I'm an abstract Vehicle
I'm a general Airplane.
I travel through the air.
I'm an abstract Vehicle
I use wheels to travel on the land.
I'm the most common wheeled vehicle.
```

Output from describe( ) of Vehicle class.

Output from describe ( ) of Airplane class.

Output from the describe ( ) of Car class.

Listing 1.5

In Listing 1.5, the function describe ( ) for each class is invoked by the object of that class. In this case, the function name binds to the object at the compile time. ***This kind of binding is called static binding or compile-time binding***. There is no ambiguity in making function call because the object making the call is clearly established at the compile time. However, situation may change when reference to the objects are used to invoke function calls. For example, the property of inheritance allows us to do two things:

- A reference to a base class can be used as an alias for a derived class object[2].
- If the formal parameter of a function expects a base class object, then any of the derived class objects may be passed as an actual argument. (We discuss this point later on in more detail).

Therefore we try to invoke the call to function describe ( ) for Airplane object through a Vehicle reference now. We create an Airplane object and then create a Vehicle reference Vref, and then set it as an alias for the Airplane object Plane. Then we use Vref to call function describe ( ) and see the results. The code in main, the results in DOS Shell are shown (Listing 1.6).

[2] **Same is true for the pointer of a base class which can be used to store the memory address of any of the derived class. We discuss this issue after discussion of pointers.**

```
void main()
{
      Airplane Plane;
      Vehicle & Vref = Plane;
      Vref.describe();
      cin.get();
}
```

```
I'm an abstract Vehicle
```

Listing 1.6

The result of Listing 1.6 is surprising.

- The describe function only gave the first line of output. The other two lines of output expected from the describe ( ) function for class Airplane are missing.

This mystery clears up when we analyze the code in the main in more detail and see as to what kind of function and object name binding is being done at the compile time. Since the memory for the Airplane object is not allocated at the compile time, the reference Vref is rightly assumed to represent an object of type Vehicle. Same is true of the function call Vref.describe( ). This statement binds Vref to the describe ( ) of class Vehicle, and not to the class Airplane at the compile time. Therefore when program runs, the function describe ( ) of Vehicle class is called. There is no run-time binding between the object to which the reference is pointing to and the function name in this case.

Both of these problems disappear when we put the C++ keyword virtual in front of the function describe ( ) in the class Vehicle. The description of class Vehicle is the modified as follows (Listing 1.7).

```
class Vehicle
{
protected:
    double speed;
public:
    Vehicle( double init_speed = -99 );
    ~Vehicle();
virtual void describe();
} ;
```

Listing 1.7

Now we run the code in main shown in Listing 1.6 again and we get the expected results (Figure 1.10).

FIG 1.10

Putting the keyword virtual in front of the describe ( ) for the Vehicle class tells compiler, to not make a function name and object binding until the run time. (Compiler does this by making a table of virtual functions and keeping track as to which function and object name bindings are to be done at runtime, rather than at compile time). Therefore declaring a function virtual shifts the object function name binding to the run time, allowing correct object and function call pairing to be performed. For all classes, which act as a base class, it is customary in C++ object oriented programming to declare all their member functions and destructor to be declared virtual to facilitate the run-time binding. Class that does not have any children now, may be used as a base class at a future date. Therefore, it is best to declare all functions and destructors as virtual in all classes. Using virtual functions has some overhead, but it is a safer approach. The modified code for Listing 1.1, where all class member functions and destructors (excluding constructors) have been declared virtual is shown in Listing 1.8 below. We also note that making functions virtual does not change its definition or implementation in any way. Therefore, we only show the modified class definitions in the Listing 1.8.

```
//*****************************************************************
class Vehicle
{
protected:
      double speed;
public:
      Vehicle( double init_speed = -99 );
      virtual ~Vehicle();
       virtual void describe();
} ;
//-------------------------------------
class Airplane : public Vehicle  {
protected:
      int numberOfWings;
public:
      Airplane( double init_speed = -99, int init_wings = -99 );
      Virtual ~Airplane();
      virtual void describe();
} ;
//---------------------------------
class Jet : public Airplane  {
protected:
      double engineThrust;
public:
      Jet( double init_speed = -99, int init_wings = -99,
      double thrust = -99);
      virtual ~Jet();
      virtual void describe();
}  ;
//---------------------------------
```

```
class PropPlane : public Airplane  {
protected:
      int numberOfProps;
public:
       PropPlane(double init_speed = -99, int init_wings = -99,
      int numProps = -99);
      virtual ~PropPlane();
      virtual void describe();
}  ;
//-------------------------------------
class VehiclesWithWheels : public Vehicle  {
protected:
      int numberOfWheels;
public:
      VehiclesWithWheels( double init_speed = -99, int nWheels = -99 );
      virtual ~VehiclesWithWheels();
      virtual void describe();
} ;
//-------------------------------------
class Car : public VehiclesWithWheels {
protected:
      int hp;
public:
      Car( double init_speed = -99,int nWheels = -99,
      int init_hp = -99);
      virtual ~Car();
      virtual void describe();
} ;
```

Listing 1.8

We summarize our conclusions so far:

- If the base class reference is used as an alias for the derived class object,
- If the base class and derived class have a function with identical signatures,
- If the base class function referred to above is not a virtual function,
- If the above referred reference is used to call the function common to both base and derived class, then

there is a static binding between the object reference and the function name. What that means is that at even though the base class reference is pointing to a derived class object, at run time the reference will invoke the base class function, and not the intended derived class member function. We got over this problem by declaring the base class function as virtual.
There are situations, however that even when the base class function is a virtual function, the dynamic binding does not take place. Let us look at one of such situations and study the remedies to it.

Once again, we use our Listing of the Vehicle and its derived class as discussed earlier (Listing 1.8). But we make one modification, that we add a function in main, which is called print ( ), and print takes an object of Vehicle class as an argument. The definition of print function – which is not a member function of any class, is as follows:

```
void print (Vehicle  Object)
{
      Object.describe( );
}
```

Listing 1.9

The purpose of function print ( ) is that it should be able to take any object that is either of type Vehicle or is derived from class Vehicle as an argument, and invoke the function describe ( ) for the object. The Listing 1.8, which declares, describe functions and destructors of all classes as virtual is used as a system of classes here.

After making all describe functions virtual, one would expect that a dynamic binding between a function name and object would be assured. Now we run the following code in the main, where we pass the object my_airplane to function print ( ) by value, as described in Listing 1.9. The main and the result outputted in the DOS shell is shown below in Listing 1.10.

```
void print(Vehicle Object)
{
      Object.describe();
}
//////////////////////////
void main()
{
      Airplane my_airplane;
      print(my_airplane);
      cin.get();
}
```

```
I'm an abstract Vehicle
```

Listing 1.10

To our dismay, the dynamic binding between the function name and the object does not take place, even when we declared the base class function describe ( ) to be a virtual function. This has to do with function print ( ) taking the parameter Vehicle by value, rather than by reference. When parameters are passed by value, only a copy is sent, and that decision is made at the compile time. Therefore, in pass-by-value mechanism, the function print will assume the incoming object to be only of type Vehicle and not its derived class type. This problem can be alleviated if we pass the parameter Vehicle to the function print ( ) by reference or by pointer. When parameters are passed by reference, then the matching of actual and formal parameters is done at the run time, because addresses of the variables are available only then. That facilitates the run-time binding between the object and the function name. Therefore we alter

the parameter passing mechanism for the function print ( ) to pass by reference and we get the intended results (Listing 1.11).

```
void print(Vehicle& Object)
{
    Object.describe();
}
//////////////////////////////
void main()
{
    Airplane my_airplane;
    print(my_airplane);
    cin.get();
}
```

```
I'm an abstract Vehicle
I'm a general Airplane.
I travel through the air.
```

Listing 1.11

Conclusion: Pass a parameter of the base class to a function by reference only, especially if you need to establish a run time binding between the object and function call.

Abstract Classes and pure virtual functions:

The classes on the top of inheritance hierarchy, from which other classes are derived, do two main things:

1. Model behavior of descendent classes. We saw that in vehicle inheritance hierarchy, in which every class had a function called describe, including the base class. This "describe" function, when made virtual in base class provided a dynamic binding between object and function call at the runtime.
2. Model data members and their re-usability by the child classes. This was exemplified by the protected data member speed in vehicle class that was inherited by all child classes. [After all can we call anything that does not have speed as a vehicle?].

However, understand that vehicle children class was under no obligation to have a function called describe. They can just as well not have any function with same name as in their base class. That would of course prevent dynamic binding, but that is another issue.

But sometime, in a system of classes, all classes should have some of the behaviors to be same. How can certain behaviors be enforced on child classes? This is done by two-prong approach.

1. Base class should be such that we do not need to instantiate it. That is easily done. After all I cannot go around the world and point to anything, which would just be a "vehicle", because it would always be a specialized version of general and abstract concept vehicle, be it a car, but, airplane, or jet. So it is possible that base class represent an abstraction of entire system of child classes, but it is not something that we actually would need to instantiate. In addition, the need to instantiate an abstract concept base class is also not there, because no real world object will match it. Here are some abstract concept classes, for example, shape, mammal, door, and DisplayDevice.
2. What gives the compiler a hint to not allow instantiation of an abstract class is that it contains at least one special function, which is called pure virtual function. These function(s) have only header and no body. A class with pure virtual function is called abstract, it cannot be instantiated, and it exists only to be derived from. In process it enforces the behaviors on its children that are captured in pure virtual functions of base class.

The pure virtual functions only have function signatures and no function definitions, even in the implementation file. The typical signature of a pure virtual function[3] would be as follows:

```
virtual void function1( ) = 0;
```

Any class having at least one pure virtual function becomes an abstract class. An abstract class cannot be instantiated. The syntax of writing an abstract class is as follows:

```
class MyAbstractClass
{
public:
virtual int getAge( ) = 0; //pure virtual function
//---------------
};
```

Let us first take a simple example of enforcing behavioral inheritance by using abstract class as a base class. Imagine all the doors in the world, for example, car door, house door, garage door, French doors etc. Well all doors have at least two behaviors, they open and they close. How can we ascertain that all door classes are forced to have those two behaviors? (Recall that behaviors in classes are imparted by functions in them).

[3] **Understand the difference between virtual and pure virtual functions. Virtual functions are like regular functions (with bodies) and with the word virtual as first word in the header. Pure virtual functions on the other hand have no bodies.**

Well we can have a Door abstract class with two pure virtual functions in it, open and close, and then derive classes, CarDoor, HouseDoor, GarageDoor, and FrenchDoor derive from abstract class Door.

We throw another chemical in this witches brew of abstract class based inheritance. The derived classes are required to override the pure virtual functions of their base class (by providing their full definitions). A base class that does not do that cannot be instantiated, and becomes an abstract class as well. A class that cannot be instantiated, well, it cannot do much. Figure below shows the inheritance from abstract class Door, with its child classes.

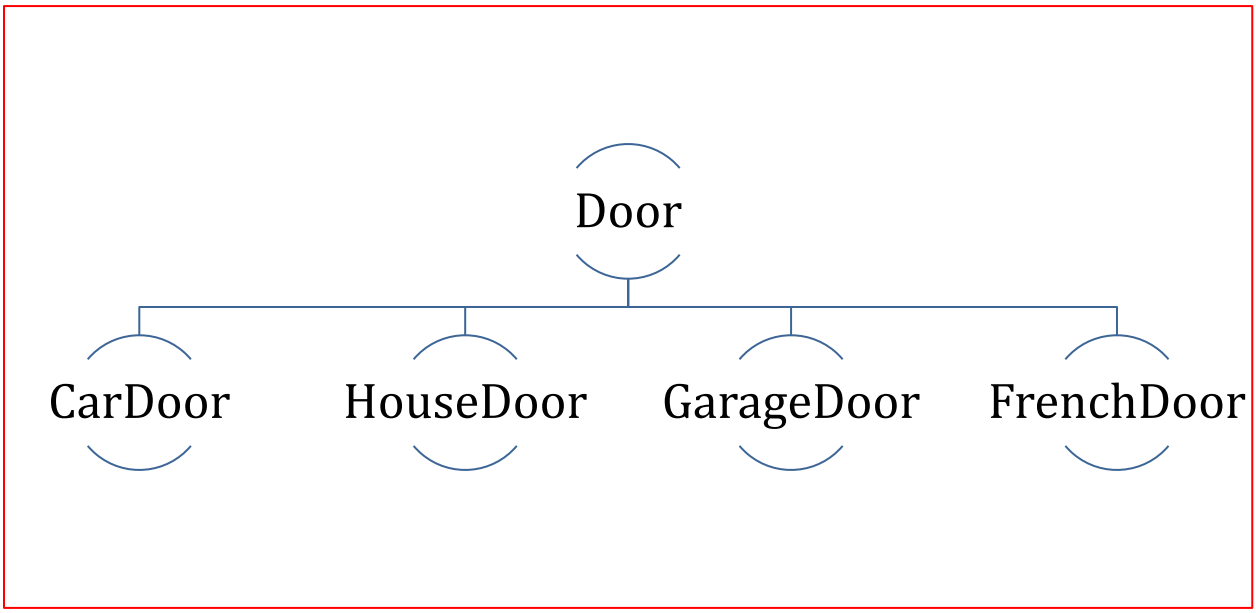


Figure: Inheritance Hierarchy from abstract class door

The table below shows the code for the system of inheritance hierarchy shown in above figure. Notice that while CarDoor, HouseDoor, and GarageDoor override pure virtual functions of base class, Door, but the class FrenchDoor does not do that. That has consequences.

```
1  #include <iostream>
2  using namespace std;
3
4  class Door{
5  public:
6     virtual void open() = 0;
7     virtual void close() = 0;
8     Door(){} // constructor is still needed, even though Door will not be instantiated
9  };
10 class CarDoor: public Door{
11 public:
12    CarDoor():Door(){}
13    void open(){
14       cout<<"Look out into traffic befor opening the door."<<endl;
15    }
```

```
16      void close(){
17          cout<<"Close the car door before starting the car."<<endl;
18      }
19  };
20
21  class HouseDoor:public Door{
22  public:
23      HouseDoor():Door(){}
24      void open(){
25          cout<<"Turn the key in the door and push to open house door."<<endl;
26      }
27      void close(){
28          cout<<"Close and lock the house door."<<endl;
29      }
30  };
31
32  class GarageDoor: public Door{
33  public:
34      GarageDoor():Door(){}
35      void open(){
36          cout<<"Press the button on the remote to open the garage door"<<endl;
37      }
38      void close(){
39          cout<<"Press the remote button to close the garage door before driving away."<<endl;
40      }
41  };
42
43  class FrenchDoor:public Door{
44  public:
45      FrenchDoor():Door(){}
46  };
47  int main(int argc, const char * argv[]) {
48      CarDoor CD;
49      HouseDoor HD;
50      GarageDoor GD;
51      Door & D1 = CD;
52      D1.open();
53      D1.close();
54      Door & D2 = HD;
55      D2.open();
56      D2.close();
57      Door & D3 = GD;
58      D3.open();
59      D3.close();
60      //FrenchDoor FD;
61      return 0;
62  }
63  //Output is below
```

```
64  /*Look out into traffic before opening the door.
65  Close the car door before starting the car.
66  Turn the key in the door and push to open house door.
67  Close and lock the house door.
68  Press the button on the remote to open the garage door
69  Press the remote button to close the garage door before driving away.
    */
```

Code for Door class inheritance hierarchy

In Listing above the base abstract class Door is coded (lines 4 to 9). Notice that it has two pure virtual functions: open and close. Though Door cannot be instantiated, it is still required to have a constructor since the Door part of children classes (CarDoor, HouseDoor etc.), cannot be built without such constructor. Class CarDoor derives from Door (lines 10 to 19). It implements and overrides the base class purr virtual functions open and close (lines 13 to 18). The classes HouseDoor (lines 21 to 30), and GarageDoor (lines 32 to 41), also override and give full implementation to base class pure virtual functions. But class FrenchDoor (lines 43 to 46) does not do that.

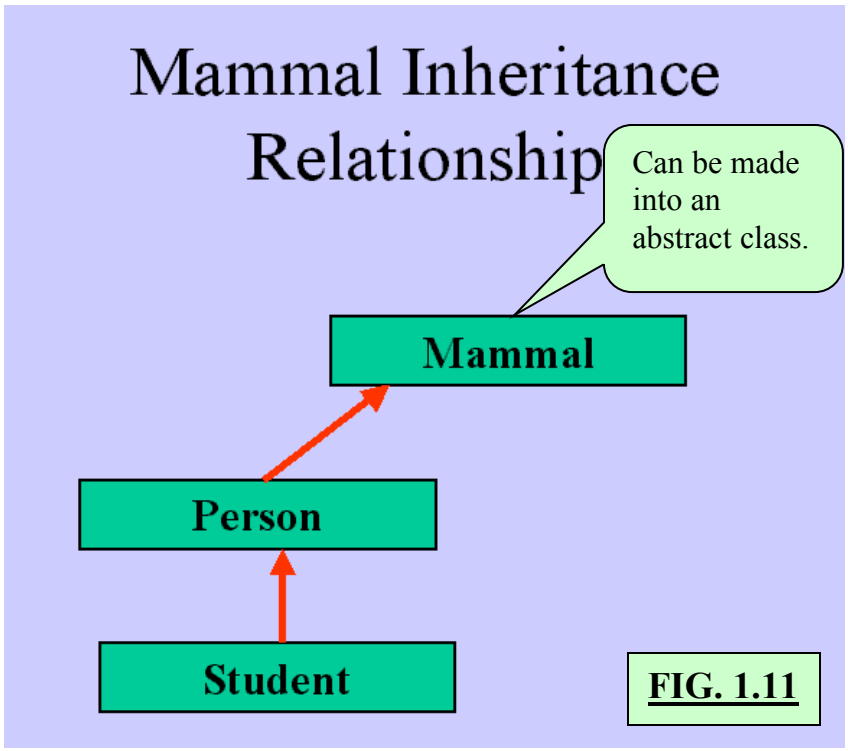
In main function, the line 48 creates a CarDoor object CD. In line 51, the Door reference D1 stores the address of CarDoor object CD. Then dynamically bonded virtual functions open and close are called on lines 52 and 53, and output from those function calls is shown in lines 65 and 66 respectively. The classes HouseDoor and GarageDoor behave in similar manner (see mapping between code lines in main function and output).

Uncommenting line 60 would cause a compile error. Not overriding Door class pure virtual functions, open and close by FrenchDoor class makes latter abstract as well. Abstract classes cannot be instantiated.

Notice that the contract with base abstract class is only that child classes will override its pure virtual functions. The details inside them are not part of that contract. Each child class can design the code inside the body of overridden functions anyways it sees fit. That is why the open function of CarDoor advises to look for traffic, before opening the car door, but open function of GarageDoor advises to press the remote button. Child classes have total freedom to implement code inside the overridden functions, anyways they need.

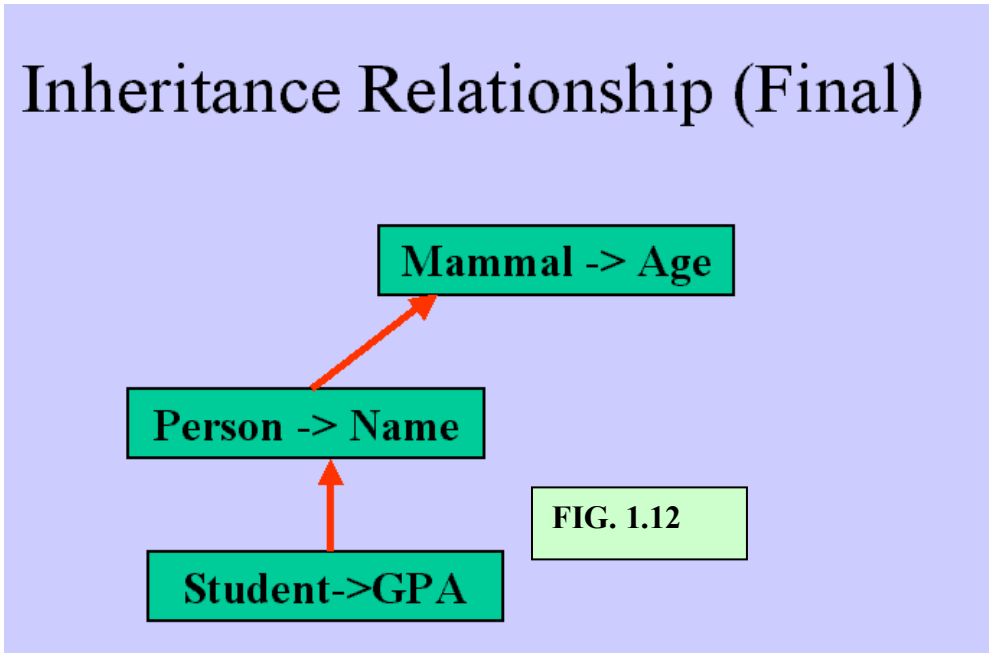
Mammal Class Inheritance Hierarchy

Another example of abstract class inheritance hierarchy is shown as we derive person class from a mammal class. We know that according to biological sciences the humans are classified under Mammals. So we can say that a *human is-a Mammal*. So with Mammal being the base class, the class Person may be derived from it. However, since we are not interested in instantiating an object of class Mammal, we can make it an abstract class. Similarly the students in a college are related to human beings by the relationship that a *student is-a person*. Therefore, the student class can be derived from the class Person. We can keep going with the derivation, for example, deriving a graduate student from class student, but for our current work, that may not be necessary. The derivation from base class Mammal to the class Person and finally to the class Student are shown in Figure 1.11 below.



Even though we make a Mammal class as an abstract class, we still need to see if some properties are common to all Mammals. One such property is age. Therefore, we now add the data members at various levels of our inheritance system. We show this data addition by our system of inheritance slides below.

Figure 1.12 below shows the final inheritance diagram with data members.



Methods for the Mammal class

Note here that since we plan to make the Mammal class as an abstract class, we may choose its functions in a manner that each non-abstract class in its derived hierarchy must have and must need those functions. First, we would need a function to print the data members of all class that are derived from Mammal. Therefore, we can say that Mammal class needs the following pure virtual functions:

```
virtual void print ( ) = 0;
virtual void getAge ( ) = 0;
```

Then of course the class Mammal will need a constructor and destructor to build the Mammal part of each object as well as destroy it. We show the header files for all classes below in Listing 1.12.

```
class Mammal //base class is an abstract class. The word abstract is optional{
protected:
        int age;
public:
      virtual void print() = 0;//Pure virtual Function
      virtual int getAge( ) = 0; // pure virtual function
        Mammal(int init_age=-99) : age(init_age){
                cout<<("Mammal constructor Invoked\n");
        }
    virtual ~ Mammal() {
                cout<<("Mammal Destructor Invoked\n");
        }
};
```

```
class Person : public Mammal{
        friend istream& operator>>(istream&, Person&);
protected:
        string m_name;
public:
        Person( const string& name="", int age=-99 );
        virtual void print();
        virtual int getAge( );
        virtual ~ Person() {
                cout<<("Person Destructor Invoked\n");
        }
} ;
```

```
class Student : public Person  {
        friend istream& operator>>(istream&, Student&);
private:
        mutable float m_gpa;
public:
Student( const string& name="", int age=-99, float gpa=-99 );
```

```
        virtual void print();
      virtual int getAge( );

        virtual ~ Student() {
                cout<<("Student Destructor Invoked\n");
        }
};
```

Listing 1.12A

The Listing implementation file is shown below in Listing 1.12B.

```
#include <iostream>
#include <string>
using namespace std;
//Insert the class definitions shown in Listing 1.12A here.
Person::Person( const string& name, int age )
: Mammal (age), m_name (name){
        cout<<("Person Constructor Invoked\n");
}
//-----------------------------------
void Person::print(){
        cout << "Name: " << m_name << endl
             << "Age : " << age << endl;
}
//-------------------------------------------
int Person::getAge() const{
        return age;
}
//-------------------------------------------
istream& operator>>(istream& input, Person& Person1){
                cout<<"Please enter the age : ";
                input>>Person1.age;
                cout<<"Please enter the name : ";
                input>>Person1.m_name;
                input.ignore(1000,'\n');
                return input;
}
//-------------------------------------------------
Student::Student(const string& name, int age, float gpa)
: Person(name,age), m_gpa(gpa){
cout<<"Student constructor Invoked\n";
}
//-----------------------------
int Student::getAge() const{
        return age;
}
//---------------------------------
```

```
void Student::print(){
	Person::print();
	cout << "Gpa  : " << m_gpa << endl;
}
//-------------------------------------------------
istream& operator>>(istream& input, Student& Student1){
			cout<<"Please enter the age : ";
			input>>Student1.age;
			cout<<"Please enter the GPA : ";
			input>>Student1.m_gpa;
			cout<<"Please enter the name : ";
			input>>Student1.m_name;
			input.ignore(1000,'\n');

			return input;
}
//==============================
void main(){
	Person Person1;
	cin>>Person1;
	Person1.print();
	cout << "--------------\n";
	Student Student1;
	cin>>Student1;
	Student1.print();
	cout << "--------------\n";
	Student Student2;
	cin>>Student2;
	Student2.print();
	cout << "--------------\n";
	cin.get();
}
```

Listing 1.12B

```
//*****************************************************
```

A typical output from the Listing 1.12B is shown below in Figure 1.13

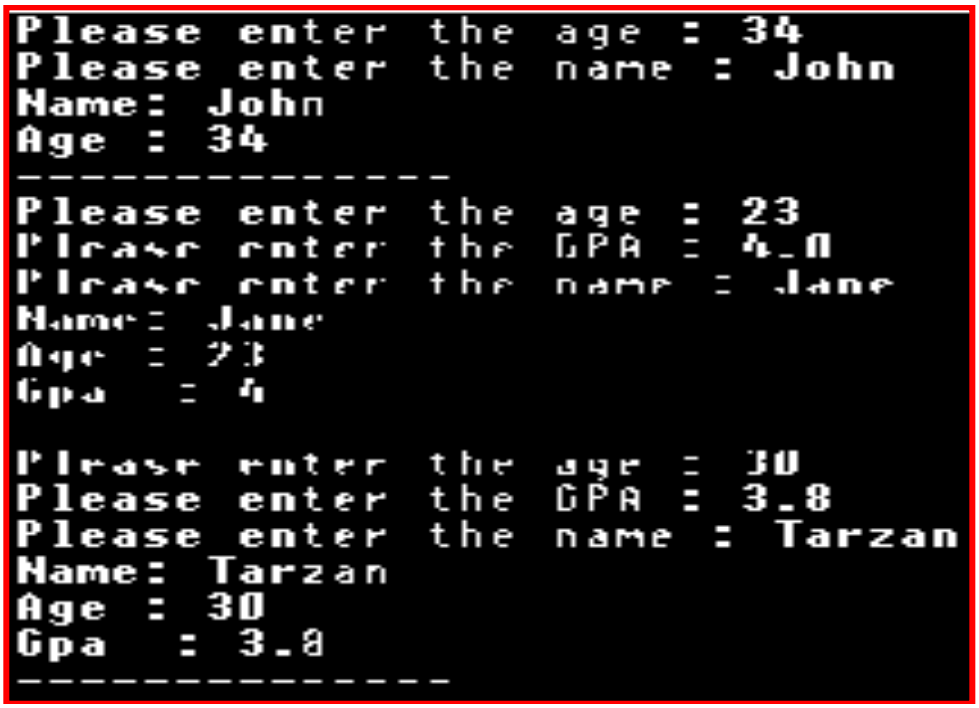


FIG. 1.13

Understand that even though we may provide a constructor for the abstract class Mammal, still it cannot be instantiated as the abstract classes exist only to become a base class to classes derived from it. A statement such as:

Mammal Temp;

will cause compile error. Abstract classes are also useful as interfaces, which contain method names that all deriving classes must implement. When we discuss stack data structure, we will show that abstract classes can be used for pure expression of data structure design from which the concrete data structure classes can be derived.

Composition Relationship

When a class is a data member of another class, they have, what is called the composition relationship. Composition relationship at times is adequate for modeling a software solution and is simpler to implement. A good example of composition relationship would be if we wished to write a complex number class, where both real and imaginary parts are represented as rational numbers. Then, we would write a separate stand-alone class for rational number and have two data members of Rational type to represent the real and imaginary parts of a complex number. Then the typical complex number will be represented as follows:

Number = 10/7 + 12/13*I

Listing 1.13 shows the typical design of the Rational number class.

```
//************************************************************
//Rational Number Arithmatic, Using Operator Overloading:
//************************************************************

class Rational
{
    public:
        Rational();                         //Default Constructor
        Rational(double, double);  //Explicit Constructor
        Rational operator+(const Rational&)const; //+ operator
        Rational operator-(const Rational&)const; //- operator
        Rational operator*(const Rational&)const; //* operator
        Rational operator/(const Rational&)const; // "/" operator
        void Read();   //Read from the key board
        void Print(int);   // Print to the key board
        double Numr(Rational)const; // Will show the numerator part only
        double Denom(Rational)const;//Will show the denominator part only

    private:
        double Num;//Numerator
        double Den;//Denominator
};
```

Listing 1.13

We can design a class called ComplexRational as follows (Listing 1.14):

```
00001 #ifndef Complex_H
00002 #define Complex_H
00003 #include <iostream>
00004 #include <fstream>
      #include "Ratinal.h"
00005 using namespace std;

00014 class ComplexRational
00015 {
00016 private:
00020         Rational Real;
00024         Rational Imag;
00025 public:
00033         ComplexRational(double Real1=0.0,double Imag1 =
0.0);
00037         ~ComplexRational();
00042         Rational getReal() const;
00049         void setReal(Rational Real1);
00055         Rational getImaginary() const;
```

```
00062          void setImaginary(Rational Imag1);
00075          friend const ComplexRational operator + (const
ComplexRational& Num1, const ComplexRational& Num2);
00089          friend const ComplexRational operator - (const
ComplexRational& Num1, const ComplexRational& Num2) ;
00099          friend const ComplexRational operator * (const
ComplexRational& Num1, const ComplexRational& Num2);
00109          friend const ComplexRational operator / (const
ComplexRational& Num1, const ComplexRational& Num2) ;
00118          friend bool operator == (const ComplexRational&
Num1, const ComplexRational& Num2) ;
00127          friend bool operator != (const ComplexRational&
Num1, const ComplexRational& Num2) ;
00135          bool operator ! () const;
00141          const ComplexRational operator ++(); //pre-
increment
00157          const ComplexRational operator ++(int marker);
//post-increment
00163          const ComplexRational operator --(); //pre-
decrement
00179          const ComplexRational operator --(int marker);
//post-decrement
00187          const ComplexRational operator - ();
00203          friend ostream&  operator << (ostream& out, const
ComplexRational& Num);
00217          friend istream&  operator >> (istream& input,
ComplexRational& Num);
00224          double absolute() const;
00225 };
00226
00227 #endif
```

//Listing 1.14

Design given in two listings above is approximate and must be refined further. We leave refinement and complete implementation as a laboratory exercise. Additional member of friend functions in Rational and ComplexRational classes may be necessary for full implementation. Design of Rational class may need to be altered to use the friend functions further.