C++ Inheritance

What is Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
  - specialization
  - augmentation

  of existing types

Define a Class Hierarchy

- Syntax:

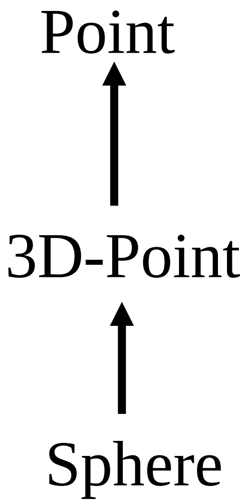
class *DerivedClassName* : access-level *BaseClassName*

where

- access-level specifies the type of derivation
  - private by default, or
  - public

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

Class Derivation



Point

3D-Point

Sphere

```
class Point{
    protected:
       int x, y;
    public:
       void set (int a, int b);
};
```

```
class 3D-Point : public Point{
    private:
       double z;
    … …
};
```

```
class Sphere : public 3D-Point{
    private:
       double r;
    … …
};
```

Point is the base class of 3D-Point, while 3D-Point is the base class of Sphere

Introduction

- Inheritance
  - Single Inheritance
    - Class inherits from one base class
  - Multiple Inheritance
    - Class inherits from multiple base classes
  - Three types of inheritance:
    - `public`: public members of base class remain public in derived class
    - `private`: public members of base class become private members of derived class
    - `protected`: public members of base class become protected members of derived class
  - Default type of inheritance is private

What a derived class doesn't inherit

- The base class's constructors and destructor
- The base class's assignment operator
- The base class's friends

What a derived class can adds

- New data members
- New member functions (also overwrite existing ones)
- New constructors and destructor
- New friends

Access Rights of Derived Classes

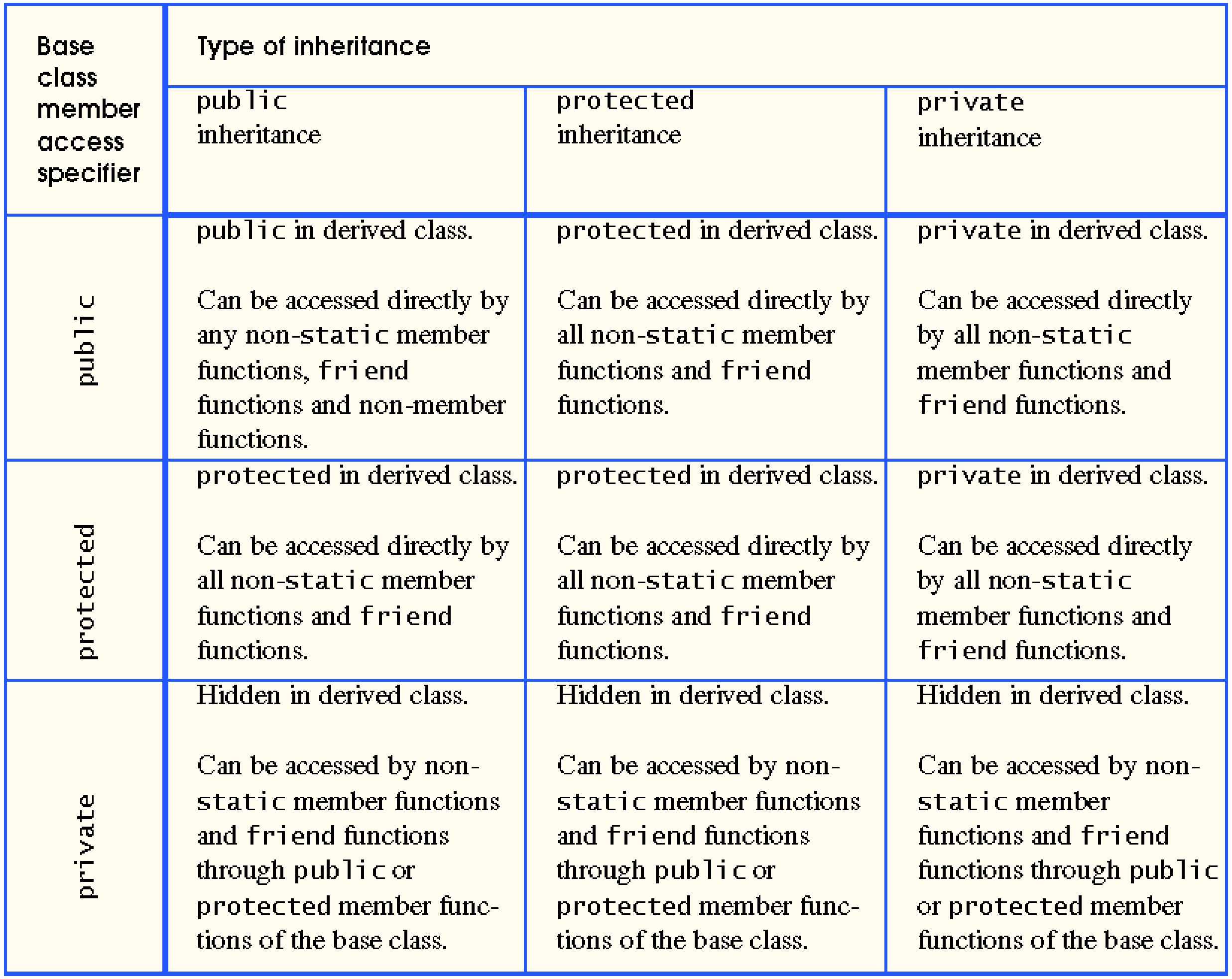
Type of Inheritance

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

Public, Private, and Protected Inheritance



Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	public in derived class. Can be accessed directly by any non-static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
protected	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
private	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.

Fig. 19.6 Summary of base-class member accessibility in a derived class.

Using 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

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
   public:
      A ( )
        {cout<< “A:default”<<endl;}
      A (int a)
        {cout<<“A:parameter”<<endl;}
};
```

```
class B : public A
{
   public:
      B (int a)
           {cout<<“B”<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Implicit Derived-Class to Base-Class Conversion

- **Upcasting**

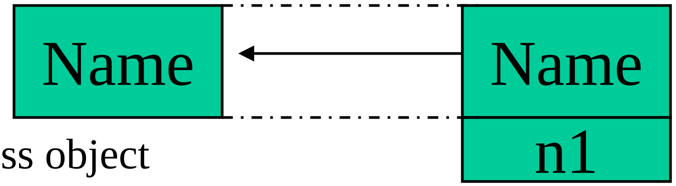
```
baseClassObject = derivedClassObject;
e.g. veh1 = car1;
```

  - This will work
    - Remember, the derived class object has more members than the base class object
    - Extra data is not given to the base class
  - **Called Object slicing**

- Downcasting

```
derivedClassObject = baseClassObject;
```

  - Will not work



Base class object

Derived class object

Overloading v/s overriding

- Redefining a member function in a derived class – **overriding**
- Many definition with same name in same class – **overloading**

Object slicing

- A derived class object can be assigned to a base class variable.
- Now will contain only base class info
- Slicing of derived class object

Casting Base Class Pointers to Derived Class Pointers

- Object of a derived class can be treated as an object of the base class
- Reverse not true - base class objects not a derived-class object

➢ Downcasting a pointer

- Use an explicit cast to convert a base-class pointer to a derived-class pointer
- Be sure that the type of the pointer matches the type of object to which the pointer points

```
derivedPtr = static_cast< DerivedClass * > basePtr;
```

Multiple Inheritance

- Derived class has multiple base classes
- Inherits all members of all base classes
- Use class name explicitly

```
class frog:public landanimal,public wateranimal
{
}
```

- Here frog class has two base classes.
- May cause ambiguity in member names
  - If both base classes have eat() function, then frog1.eat() causes ambiguity error
- If multiple inheritance is used with common ancestor class, this class members are repeated
- Avoid this using Virtual inheritance

Polymorphism

- Achieved with virtual functions.
- These are overridden in the derived classes
- Base class pointer – pointing to derived class object will call the correct version of the virtual function
- This is called dynamic polymorphism
- Function must be virtual
- Object **must be referred** through pointer or reference
- Size of polymorphic object is increased by one pointer
- Polymorphism is Achieved through vptr and vtable
- Refer Thinking in C++ by Bruce Eckel

Late binding

```
// pointers to base class
 #include <iostream>
using namespace std;
class Cpolygon
 {
      protected:
                int width, height;
       public:
                void set_values (int a, int b)
                 {
                                width=a; height=b;
                }
       };
 class CRectangle: public CPolygon
{
      public:
                 int area ()
                 {
                                return (width * height);
                }
};
```

```
class CTriangle: public CPolygon
{
       public:
                int area ()
                {
                return (width * height / 2);
                 }
};
int main ()
{
      CRectangle rect;
      CTriangle trgl;
       CPolygon * ppoly1 = &rect;
      CPolygon * ppoly2 = &trgl;
      ppoly1->set_values (4,5);
      ppoly2->set_values (4,5);
      cout << rect.area() << endl;
      cout << trgl.area() << endl;
      return 0;
}
```

Output: 20 10

```
#include<iostream>
using namespace std;
class base{
int i;
public:
base()
{
        cout<<"\n we are in base no argumented constructor";
}
base(int x)
{
        cout<<"\n we are in base constructor";
        i=x;
}
virtual void print()
{
cout<<"\n we are in base print which is virtual, member value
        is::"<<i;
}
```

```
void display()
{
cout<<"\n we are in base display, member value is::"<<i;
}
};
class derived:public base
{
int j;
public:
derived(int y)
{
cout<<"\n we are in derived constructor";
j=y;
}
void print()
{
cout<<"\n we are in derived print, member value is::"<<j
;
}
void display()
{
cout<<"\n we are in derived display, member value is::"<
<j;
}
};
```

```
int main()
{
        class base b(10);
        class derived d(20);
        class base *b1;
        cout<<"\n we are calling base member functions by using base object";
        b.print();
        b.display();
        cout<<"\n we are calling derived functions by using derived object";
        d.print();
        d.display();
        b1=&b;
        cout<<"\n we created base pointer and we assigned the base obj address";
        b1->print();
        b1->display();
        b1=&d;
        cout<<"\n we created base pointer and we assigned the derived obj address";
        b1->print();
        b1->display();
        cout<<"\n";
        return 0;
}
```

```
we are in base constructor
 we are in base no argument constructor
 we are in derived constructor
 we are calling base member functions by using base object
 we are in base print which is virtual, member value is::10
 we are in base display, member value is::10
 we are calling derived functions by using derived object
 we are in derived print, member value is::20
 we are in derived display, member value is::20
 we created base pointer and we assigned the base obj address
 we are in base print which is virtual, member value is::10
 we are in base display, member value is::10
 we created base pointer and we assigned the derived obj address
 we are in derived print, member value is::20
 we are in base display, member value is::8390773
```

Abstract class

- A class whose definition is incomplete
- Can not create an object of abstract class
- Virtual function is equated to zero
- Is called Pure virtual function
- The derived classes **must** override this pure virtual function.

```
class shape
{
  public:
   void draw()=0;
  };

  int main()
  {
    shape s1;//syntax error
```