Computer Programming

Inheritance

Hung-Yun Hsieh
December 2, 2022

Inheritance

Important features of OOP (object-oriented programming): abstraction, encapsulation, inheritance, polymorphism

- Software reusability
  - Create new classes from existing classes
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
  - ☞ Derived class inherits from base class
  - Base class typically represents a larger set of objects (with more general behaviors) than derived classes
  - ☞ Base class: Vehicle
    - Includes cars, trucks, boats, bicycles, etc.
  - ☞ Derived class: Car
    - Smaller, more-specific subset of vehicles
  - ☞ Car is a vehicle
    - Class Car inherits from class Vehicle

Inheritance vs. Composition

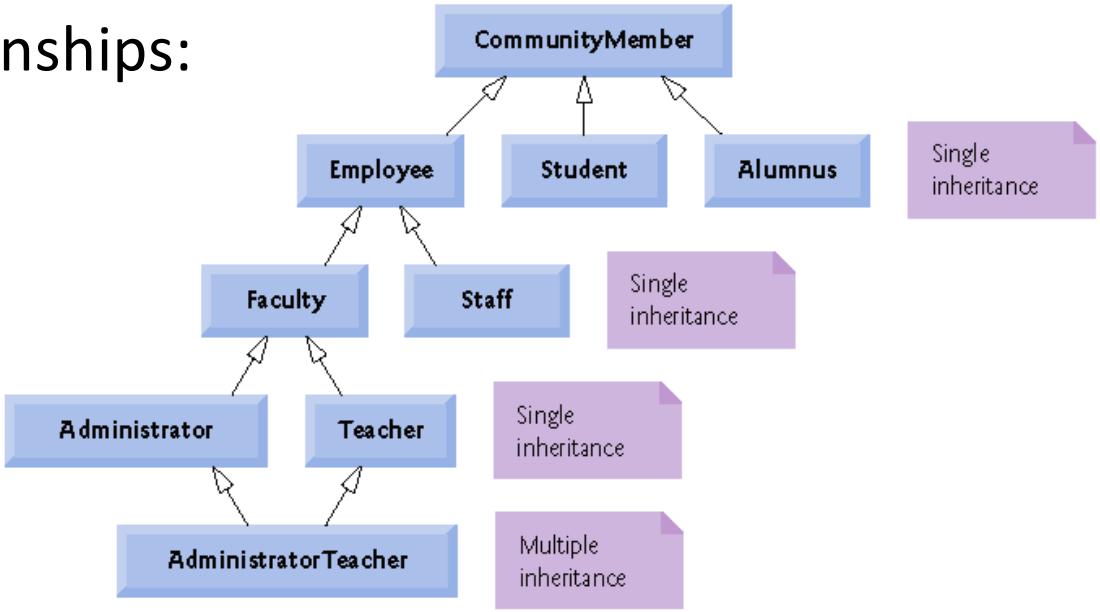
- Inheritance
  - A derived class object can also be treated as a base class object
  - Example: A car is a vehicle
    - Properties and behaviors of a vehicle also apply to a car
  - ☞ Relationship of "is-a"
- Composition
  - An object contains one or more objects of other classes as members
  - Example: A car has a steering wheel
    - A steering wheel has very different behaviors from a car
  - ☞ Relationship of "has-a"

Inheritance Hierarchy

Note the direction of the pointer to specify the inheritance relationship

- Inheritance hierarchy
  - Direct base class: One level up hierarchy
  - Indirect base class: Two or more levels up hierarchy
  - Single inheritance: Inherits from one base class
  - Multiple inheritance: Inherits from multiple base classes

☞ Inheritance relationships: tree-like hierarchy structure



Inherited Members

Some use the statement that constructors/destructors are not inherited

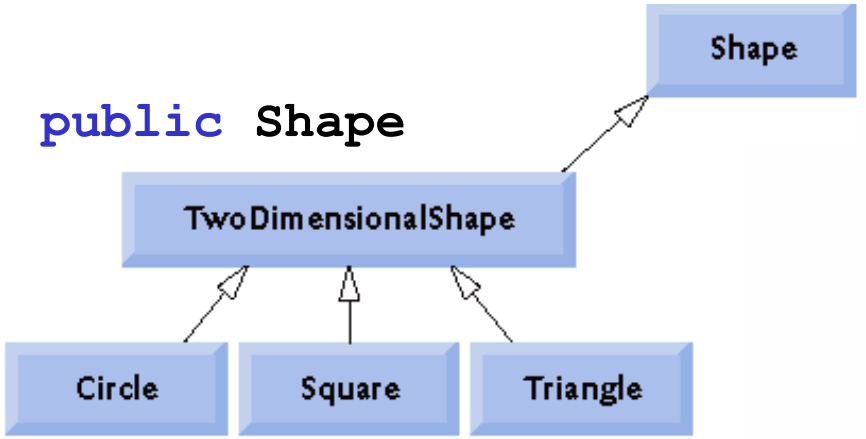
- A derived class inherits
  - Every data member and function member defined in the base class
    - But inherited members may not be accessible in the derived class (e.g. *private* members in the base class)
    - ☞ The derived class can determine how the inherited members are available (accessibility) to the outside
- A derived class *does not* inherit
  - The friends of the base class
  - ☞ Some member functions are inherited but cannot be used for the same purpose in the derived class
    - ☞ The constructor/destructor *of the base class* do not perform construction/destruction automatically *for the derived class*

`public` Inheritance

A derived class can access the private members of the base class indirectly through the public member functions of the base class

- Types of inheritance
  - `public`, `private`, and `protected`
- Public inheritance

```
class TwoDimensionalShape : public Shape
{
…
};
```



  - Publicly accessible members inherited from the base class become public members in the derived class
  - Private members are still inaccessible in the derived class
    - ☞ The derived class cannot directly access private members inherited from the base class

`protected` Access Specifier

- Protected access
  - Intermediate level of protection between `public` and `private`
  - `protected` members are accessible to
    - Base class members and friends
    - Derived class members and friends

Protected members are not accessible to the outside (of the class) otherwise (e.g. behave like private to the outside)

- Access base-class members in derived class
  - Simply use member names to refer to `public` and `protected` members of base class
  - Redefined (Overridden) base class members can be accessed by using base-class name and scope resolution ::
    - Redefined members are shadowed (overridden) otherwise

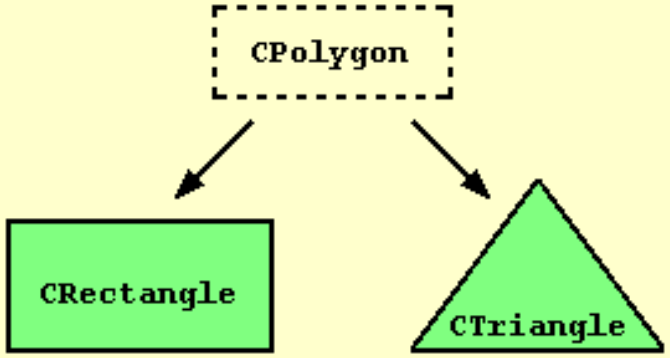
Example on Polygon (1/2)

```
#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); }
};
```



specific to each derived class

Example on Polygon (2/2)

How should the program be changed if the member `width` and `height` are private in `CPolygon`?

```
int main ( )
{
  CRectangle rect;
  CTriangle trgl;

  rect.set_values (4,5);
  trgl.set_values (4,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;
}
```

`sizeof(rect)=sizeof(trgl)=8`

The inherited function `set_values()` from the base class becomes the public member of the derived class

- Each object of the classes CRectangle and CTriangle contains members inherited from the CPolygon class
  - `width`
  - `height`
  - `set_values()`

Member Accessibility

Distinguish between:
(1) whether the member can be accessed in the derived class or not, and
(2) whether the member can be accessed outside the derived class or not

- Access specifier & types of inheritance

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class.	`protected` in derived class.	`private` in derived class. (2)
	Can be accessed directly by member functions, `friend` functions and nonmember functions. (1)	Can be accessed directly by member functions and `friend` functions. (1)	Can be accessed directly by member functions and `friend` functions. (1)
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Extension to the Base Class

- A derived class can
  - Add new member data or functions
  - Redefine existing member functions
    - Redefined (Overridden) base class members are shadowed but they can be accessed by using base-class name and scope resolution ::
  - Hide or expose base-class member functions
    - Change the access specifier of accessible base-class member functions
    - ☞ Cannot change from private to any others
  - Add new functionalities to existing member functions
    - Use the scope resolution `::` to access base-class member functions in the redefined member function

Accessibility Change

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

class Base
{
    int value;
public:
    Base(int nValue) : value(nValue) { }
protected:
    void PrintValue() { cout << "Base: " << value << endl; }
};

class Derived: public Base
{
    int value;
public:
    Derived(int nValue) : Base(nValue*2), value(nValue) { }
    //void PrintValue() { cout << "Derived: " << value << endl; }
    Base::PrintValue;
};
```

```
int main()
{
    Derived cDerived(7);
    cDerived.PrintValue();
}
```

Note how the base class member is *initialized* in the derived class

`Derived::PrintValue()` now is a public member in class `Derived`

or `using Base::PrintValue;`

Example on Multiple Inheritance (1/2)

- Multiple inheritance
  - A derived class inherits the members of two or more base classes

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
};

class COutput {
  public:
    void output (int i) { cout << i << endl; }
};
```

Example on Multiple Inheritance (2/2)

```
class CRectangle : public CPolygon, public COutput {
  public:
    int area () { return (width * height); }
};

int main () {
  CRectangle rect;

  rect.set_values (4,5);
  rect.output (rect.area());
}
```

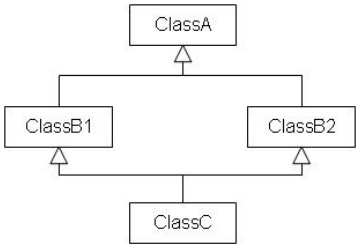
```
class ClassB1:virtual public ClassA{…};
Class ClassB2:virtual public ClassA{…};
```

Okay to write:

```
rect.CPolygon::set_values(4, 5)
```

Can be used to call the member function defined in the base class that is overridden by the derived class

- Multiple inheritance is powerful, but can cause a variety of ambiguity problems
  - Members with the same name in multiple base classes
  - Different base classes inherit from the same indirect base class (diamond hierarchy) → advanced topic on **virtual inheritance**



Computer Programming

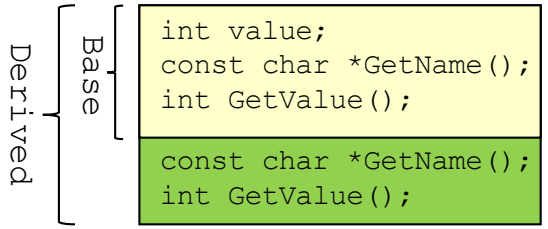
Polymorphism

Base Class and Derived Class

It is a compilation error to aim a derived-class pointer at a base-class object

- Derived class
  - A derived class inherits from the base class
  - A derived class is composed of at least two parts: one part for each inherited class, and one part for itself
  - ☞ A derived class object is an object of its base class, so it can be *converted* to a base class object
- Base-class reference and pointer
  - Can only see members of the base class (or any classes that the base class inherits)
  - Can not see members of the derived class
  - ☞ A base-class pointer however is a handle for manipulating all derived objects in a unified, compatible fashion

Inheritance Revisited (1/2)



C++ does not allow derived classes to directly initialize inherited member variables in the initialization list of the constructor

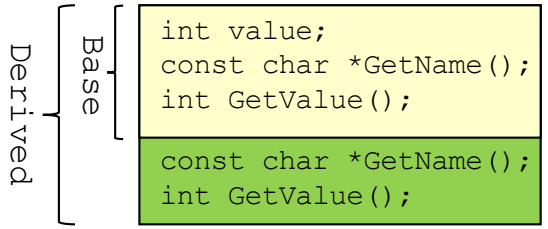
```
#include <iostream>
using namespace std;

class Base
{
protected:
    int value;
public:
    Base(int nValue) : value(nValue) { }
    const char *GetName() { return "Base"; }
    int GetValue() { return value; }
};

class Derived : public Base
{
public:
    Derived(int nValue) : Base(nValue) { }
    const char *GetName() { return "Derived"; }
    int GetValue() { return value * 2; }
};
```

The base class member is initialized in the derived class by calling the *constructor of the base class* in the member initialization list

Inheritance Revisited (2/2)




```
int main()
{
    Derived cDerived(5);
    Base  aBase = cDerived;
    Base &rBase = cDerived;
    Base *pBase = &cDerived;
    cout << "cDerived is a " << cDerived.GetName()
         << " and has value " << cDerived.GetValue() << endl;
    cout << "aBase is a " << aBase.GetName()
         << " and has value " << aBase.GetValue() << endl;
    cout << "rBase is a " << rBase.GetName()
         << " and has value " << rBase.GetValue() << endl;
    cout << "pBase is a " << pBase->GetName()
         << " and has value " << pBase->GetValue() << endl;
}
```

It is okay to convert (up cast) a derived class to the base class since a derived class has the base class in it ("is-a")

```
Derived *pDer = &cDerived;
```

The **handle determines** the function to call

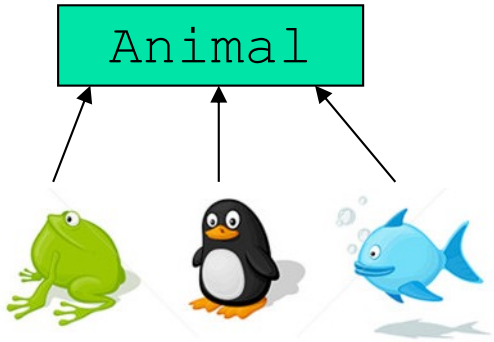
```
cDerived is a Derived and has value 10
aBase is a Base and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

If `protected` or `private` inheritance is used, then we still cannot access the members using the base class reference or pointer

An Example

It is possible to pass different animal objects individually as a function argument declared as a base-class pointer

- Animal class hierarchy
  - The Animal base class
    - Derived classes: Fish, Frog, and Bird
    - Every derived class has the function move()
  - A function (trigger) to trigger actions by events
    - Writing 3 functions vs. writing 1 function
    - The function accepts the Animal pointer as the argument (since the Animal base class is common to all objects)




```
Animal *a[3]={&f, &g, &b};

for (int i=0;i<3;i++)
trigger(a[i]);
```

```
Fish f;
Frog g;
Bird b;

trigger(&f);
trigger(&g);
trigger(&b);
```

```
void trigger(Animal *p)
{
  …
  p->move();
  …
}
```

p->move(): same function call, but different functions should be invoked

An Example (cont.)

```
trigger(Animal *p)
{ …
  p->move();…
}
```

- Base class pointer
  - It is desirable that the proper function gets called
    - A Fish will move forward by swimming
    - A Frog will move forward by jumping
    - A Bird will move forward by flying
  - However, the base class pointer can only see members of the base class
- Polymorphism is needed when a program aims to invoke a function in the derived class *through a base-class pointer or reference*
  - ☞ Which movement function is invoked depends dynamically on the type of object the pointer points to, rather than the *handle itself*

Polymorphism

- Multiple forms
  - The state of existing in or assuming many forms
  - Ability of objects belonging to different types to respond to method calls of the same name, each one according to an appropriate type-specific behavior
- Class-based polymorphism in C++
  - Calls to such functions then get dispatched according to the actual type of the object which owns them, rather than the apparent type of the pointer (or reference) through which they are invoked
  - Member functions can be marked as polymorphic using the `virtual` keyword

Virtual Function

- Virtual function
  - A member function that is declared with the `virtual` keyword is called a virtual function

```
virtual void move();
```

- Function invocation
  - Normally the type of the *handle* determines which class's function member to invoke
  - With virtual functions, the type of the object being pointed to (not type of the handle) determines which version of a virtual function to invoke
    - Allows program to dynamically (at runtime rather than compile time) determine which function to use
    - ☞ Referred to as dynamic binding or late binding

The Polygon Take Two (1/2)

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area () { return (0); }
};

class CRectangle : public CPolygon {
  public:
    int area () { return (width * height); }
};

class CTriangle : public CPolygon {
  public:
    int area () { return (width * height / 2); }
};
```

Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy even if the function is not explicitly declared virtual when a derived class overrides it

The Polygon Take Two (2/2)

```
int main ( )
{
    CRectangle rect;
    CTriangle  trgl;
    CPolygon   poly;

    CPolygon * ppoly[] = {&rect, &trgl, &poly};

    for (int i=0;i<3;i++)
    ppoly[i]->set_values (4, 5);

    for (int i=0;i<3;i++)
    cout << ppoly[i]->area() << endl;
}
```

Note the use of the base-class pointer `ppoly[i]` for providing a "compatible" interface among *all derived objects*

Which `area()` function is called depends on the object that the `CPolygon` base-class pointer points to

```
20
10
0
```

More on the Polygon

It is mandatory for the derived class to implement such virtual functions for instantiation of the derived-class object

- The virtual function `area()` in CPolygon
  - The only purpose is to act as a *place-holder* for "virtualizing" implementations in specific derived classes
- Abstract base class
  - A class with an empty virtual function

```
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area () = 0;
};
```

Different from:
`virtual int area () {}`

  - An abstract class cannot be instantiated (why else?)

```
CPolygon poly;                    // wrong!!
CPolygon *ppoly;                  // okay!!
```

The Polygon Take Three (1/2)

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

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area () = 0;
    void print_area (void) { cout << area() << endl; }
};

class CRectangle : public CPolygon {
  public:
    int area () { return (width * height); }
};
class CTriangle : public CPolygon {
  public:
    int area () { return (width * height / 2); }
};
```

The Polygon Take Three (2/2)

```
int main ( )
{
    CRectangle rect;
    CTriangle trgl;

    CPolygon * ppoly[] = {&rect, &trgl};

    for (int i=0;i<2;i++)
    ppoly[i]->set_values (4, 5);

    for (int i=0;i<2;i++)
    ppoly[i]->print_area();
}
```

Note that although `CPolygon` implements `print_area()`, it does not implement the function `area()` so it is an abstract class and can not be instantiated

```
20
10
```

More on the Virtual Function

- The size of a polymorphic class

```
cout << "The size of CPolygon=" << sizeof(CPolygon) << endl;
cout << "The size of CRectangle=" << sizeof(CRectangle) << endl;
cout << "The size of CTriangle=" << sizeof(CTriangle) << endl;
```

```
16
16
16
```

  - The size of a polymorphic class (one with at least one virtual function) adds an extra 8 bytes
    - ☞ The extra 8 bytes are for a "virtual pointer" that **points** to a "virtual table" of the class containing addresses of all the virtual functions of the class (one virtual table per class) – for resolving the function addresses through the object
    - The pointer is *inherited* -- even if the derived class overrides a virtual function in its base class without using the "virtual" modifier, the overridden function is still virtual
    - ☞ Once the base class is polymorphic, all of its derived class are polymorphic

Virtual Table (1/2)

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};
```

```
int main()
{
    D1 cClass;
    Base *pClass = &cClass;
    pClass->function1();
}
```

```
class D1 : public Base
{
public:
    virtual void function1() {};
};
```

```
int main()
{
    Base cClass;
    Base *pClass = &cClass;
    pClass->function1();
}
```

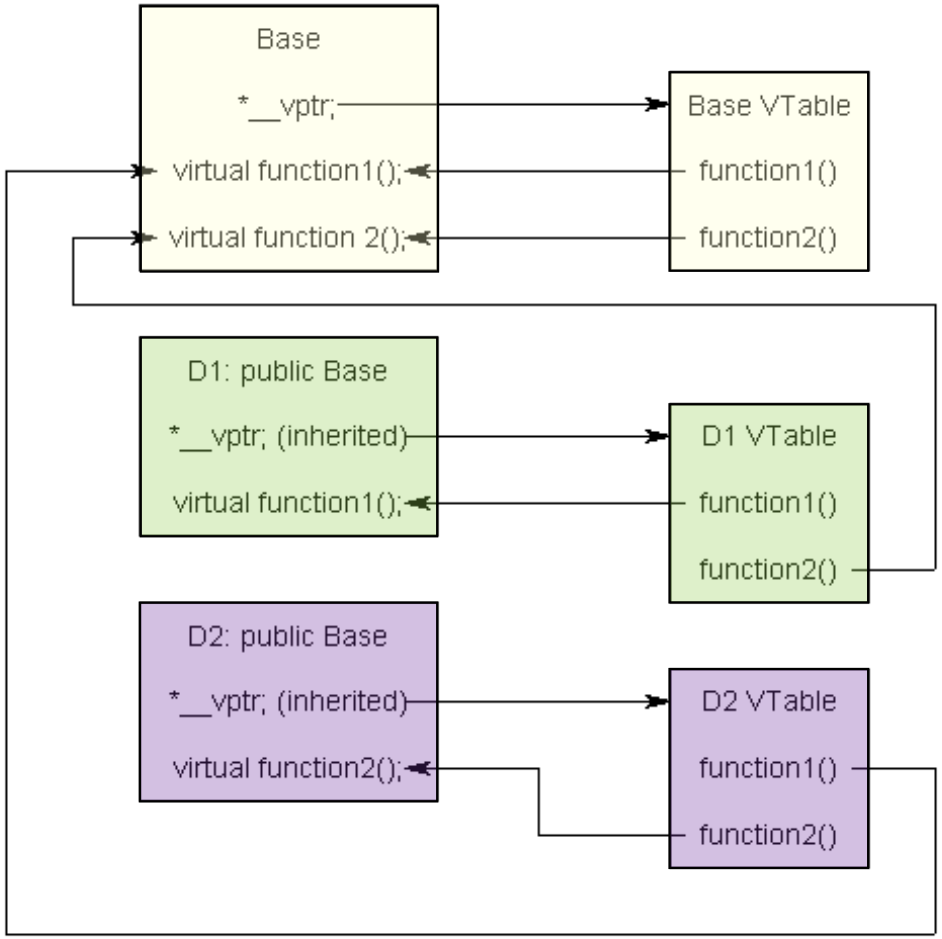
```
class D2 : public Base
{
public:
    virtual void function2() {};
};
```

☞ Three virtual tables are set up

Virtual Table (2/2)

Actual implementation of polymorphism differs from one compiler to another

- The base class has a hidden pointer `__vptr`
  - Starting from the base class that has a virtual function, each derived class inherits the pointer
  - When a class object is created, its `__vptr` points to the virtual table of that class
  - The "non-shadowed" function an object of that class can call is filled in the table



Virtual Destructor (1/2)

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

class Base
{
public:
    ~Base() { cout << "Calling ~Base()" << endl; }
};

class Derived : public Base
{
private:
    int* pArray;

public:
    Derived(int nLength) { pArray = new int[nLength]; }

    ~Derived() {
        cout << "Calling ~Derived()" << endl;
        delete[] pArray;
    }
};
```

Virtual Destructor (2/2)

```
int main()
{
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;
}
```

```
Calling ~Base()
```

- Deleting an object through its base pointer
  - It is desirable to make the destructor virtual when dealing with inheritance

```
class Base { … virtual ~Base() {…} …};
class Derived { … ~Derived() {…} …};
```

```
Calling ~Derived()
Calling ~Base()
```

Once the destructor of a base class is declared virtual, all derived-class destructors become virtual even though they do not have the same name as the base-class destructor

Derived Class Construction

- Instantiating a derived-class object
  - When a program creates a derived-class object,
    - ① The derived-class constructor immediately calls the base-class constructor (implicitly using the default constructor or explicitly through the member initializer)
    - ② The member initializer of the base class executes
    - ③ The constructor body of the base-class executes
    - ④ The member initializer of the derived class executes
    - ⑤ Finally the constructor body of the derived-class executes
    - ☞ This process cascades up the hierarchy if the hierarchy contains more than two levels
  - ☞ The base of the inheritance hierarchy
    - Last constructor called in chain
    - First constructor body to finish executing (construction)

Derived Class Destruction

- Destroying a derived-class object
  - Reverse order of the constructor chain
  - Destructor of derived-class called first
  - Destructor of the next base class up hierarchy called next
  - The process continues up hierarchy until the final base is reached
- ☞ After the final base-class destructor finishes execution, the object is removed from the memory

Example on Point (1/2)

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

class CPoint1d {
  public:
    CPoint1d(float x=0.0) : _x(x) {cout << "CPoint1d Constructor\n";}
    ~CPoint1d() {cout << "CPoint1d Destructor\n";}
    float x() { return _x; }
  protected:
    float _x;
};
class CPoint2d : public CPoint1d {
  public:
    CPoint2d(float x=0.0, float y=0.0) : CPoint1d(x), _y(y)
    {cout << "CPoint2d Constructor\n";}
   ~CPoint2d() {cout << "CPoint2d Destructor\n";}
    float y() { return _y; }
  protected:
    float _y;
};
```

member initialization

base class construction (initialization)

Example on Point (2/2)

```
class CPoint3d : public CPoint2d {
  public:
    CPoint3d(float x=0.0, float y=0.0, float z=0.0)
    : CPoint2d(x, y), _z(z) {cout << "CPoint3d Constructor\n";}
   ~CPoint3d() {cout << "CPoint3d Destructor\n";}
    float z() { return _z; }
  protected:
    float _z;
};

int main()
{
  cout << "size of Class CPoint1d=" << sizeof(CPoint1d) << endl
       << "size of Class CPoint2d=" << sizeof(CPoint2d) << endl
       << "size of Class CPoint3d=" << sizeof(CPoint3d) << endl;

  CPoint3d aPoint3d(1.1, 2.2);
  cout << "x=" << aPoint3d.x() << ", y=" << aPoint3d.y()
       << ", z=" << aPoint3d.z() << endl;
}
```

```
size of Class CPoint1d=4
size of Class CPoint2d=8
size of Class CPoint3d=12
CPoint1d Constructor
CPoint2d Constructor
CPoint3d Constructor
x=1.1, y=2.2, z=0
CPoint3d Destructor
CPoint2d Destructor
CPoint1d Destructor
```

Review

- Inheritance
  - Members inherited from the base class that can be used in the derived class
  - The `protected` access specifier
  - Public, private, and protected inheritance
- Polymorphism
  - One function call to the base-class pointer can invoke different functions depending on the specific derived objects the pointer points to
  - Dynamic binding and the use of the virtual function
  - Abstract class and the use of the pure virtual function