

# The C++ Programming Language

## 继承与派生类

陈辰

chenc@fudan.edu.cn

复旦大学软件学院

# 课程要点

- 继承与派生类
- 派生类的构造函数和析构函数
- 多继承
- 赋值兼容规则

## 4.1 继承与派生类

- 面向对象4个特征:抽象、封装、继承、多态
- 继承目的：代码的重用和代码的扩充
- 继承方法程序设计思路：一般 -> 特殊
- 继承种类：单继承、多继承
- 基本概念：基类（父类）、派生类（子类）
- 继承方式：public protected private
- 继承内容：除构造函数、析构函数、私有成员外的其他成员
- 重点掌握内容：

**继承方式及访问规则、构造函数定义**

# 继承的概念

- 2个过程：
  - 保持已有类的特性而构造新类的过程称为继承。
  - 在已有类的基础上新增自己的特性而产生新类的过程称为派生。
- 2个概念：
  - 被继承的已有类称为基类（或父类）。
  - 派生出的新类称为派生类。

# 继承与派生的目的

- 继承的目的：实现代码重用。（站在巨人的肩膀上）
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。（创新）

(1)修改基类成员:是在派生类中声明了一个与基类成员同名的新成员。在派生类作用域内或者在类外通过派生类的对象直接使用这个成员名，只能访问到派生类中声明的同名新成员，即新成员覆盖了从基类继承的同名成员，称为同名覆盖。

(2) 描述新的特征或方法。

区分“重载（描述新的特征和方法）”和“重写（同名覆盖）”

# 为什么要使用继承

```
class person {
```

```
    protected:
```

```
        char name[10];
```

```
        int age;
```

```
        char sex;
```

```
    public:
```

```
        void print();
```

```
};
```

```
class employee{
```

```
    protected:
```

```
        char name[10];
```

```
        int age;
```

```
        char sex;
```

```
        char department[20];
```

```
        float salary;
```

```
    public:
```

```
        print();
```

```
};
```

# 派生类的声明

//定义一个基类

```
class person{  
protected:  
    char name[10];  
    int age;  
    char sex;  
public:  
    //.....  
};
```

//定义一个派生类

```
class employee:public person {  
protected:  
    char department[20];  
    float salary;  
public:  
    //.....  
};
```

# 继承的访问控制

- 派生类继承了基类中除构造函数和析构函数之外的所有成员。
- 派生类的成员包括：
  - (1) 继承基类的成员；
  - (2) 派生类定义时声明的成员；



从已有类派生出新类时,可以在派生类内完成以下几种功能:

- (1) 可以增加新的数据成员;
- (2) 可以增加新的成员函数;
- (3) 可以重新定义基类中已有的成员函数;
- (4) 可以改变现有成员的属性。

基类成员在派生类中的访问属性与继承方式有关。

# 声明一个派生类的一般格式为:

```
class 派生类名:继承方式 基类名
{
    //派生类新增的数据成员和成员函数
};
```

由类person继承出类employee的三种格式:

(1) 公有继承

```
class employee:public person {
    //...
};
```

(2) 私有继承 (系统默认为私有继承)

```
class employee:private person{
    //...
};
```

(3) 保护继承

```
class employee:protected person{
    //...
};
```

# 继承的访问控制

- 三种继承方式:
  - 公有继承(public)
  - 私有继承(private)
  - 保护继承(protected)

派生类成员的访问权限:

- 1) inaccessible(不可访问)
- 2) public
- 3) private
- 4) protected

# 基类成员在派生类中的访问属性

在基类中的访问属性	继承方式	在派生类中的访问属性
<b>private</b>	<b>public</b>	不可直接访问
<b>private</b>	<b>private</b>	不可直接访问
<b>private</b>	<b>protected</b>	不可直接访问
<b>public</b>	<b>public</b>	<b>public</b>
<b>public</b>	<b>private</b>	<b>private</b>
<b>public</b>	<b>protected</b>	<b>protected</b>
<b>protected</b>	<b>public</b>	<b>protected</b>
<b>protected</b>	<b>private</b>	<b>private</b>
<b>protected</b>	<b>protected</b>	<b>protected</b>

- 不同继承方式的基类和派生类特性

继承方式	基类特征	派生类特征
公有继承	public protected private	public protected 不可访问
私有继承	public protected private	private private 不可访问
保护继承	public protected private	protected protected 不可访问

# 派生类对基类成员的访问规则

派生类对基类成员的访问形式主要有以下两种:

(1) **内部访问**: 由派生类中新增成员对基类继承来的成员的访问。

(2) **对象访问**: 在派生类外部,通过派生类的对象对从基类继承来的成员的访问。

- 不同继承方式决定的不同访问控制权限体现在：
  - 1、派生类**成员**对继承于基类的成员的访问控制。
  - 2、派生类**对象**对继承于基类的成员的访问控制。

# 1. 私有继承的访问规则

基类的`public`成员和`protected`成员被继承后作为派生类的`private`成员,派生类的其他成员可以直接访问它们,但是在类外部通过派生类的对象无法访问。

基类的`private`成员在私有派生类中是`不可直接访问`的,所以无论是派生类成员还是通过派生类的对象,都无法直接访问从基类继承来的`private`成员,但是可以通过基类提供的`public`成员函数间接访问。

通过`派生类的对象`不能访问基类中的任何成员。

表4-2 私有继承的访问规则

基类成员	private成员	public成员	protected成员
内部访问 对象访问	不可访问 不可访问	可访问 不可访问	可访问 不可访问



## 例4.1 私有继承举例

```
class Point { //基类Point类的定义
public:      //公有函数成员
    void InitP(float x = 0, float y = 0) {
        this->X = x; this->Y = y;}
    void Move(float offX, float offY) {
        X += offX; Y += offY; }
    float GetX() const { return X; }
    float GetY() const { return Y; }
private:   //私有数据成员
    float X, Y;
};
```

## 例4.1 私有继承举例

```
class Rectangle: private Point    //派生类声明
{public: //新增外部接口
    void InitR(float x, float y, float w, float h)
    {InitP(x,y);W=w;H=h;} //访问基类公有成员
    void Move(float xOff, float yOff) {Point::Move(xOff,yOff);}
    float GetX ( ) const {return Point::GetX ( ) ;}
    float GetY ( ) const {return Point::GetY ( ) ;}
    float GetH ( ) const {return H;}
    float GetW ( ) const {return W;}
private: //新增私有数据
    float W,H;
};
```

调用的是基类  
的函数，指  
定作用域::

InitP	private
Move	private
GetX	private
GetY	private
X,Y	inaccessible
W,H	private

```
#include<iostream.h>
#include<math.h>
int main ( )
{ //通过派生类对象只能访问本类成员
    Rectangle rect; //定义Rectangle类的对象
    rect.InitR(2,3,20,10); //设置矩形的数据
    rect.Move(3,2); //移动矩形位置
    cout<<rect.GetX ( ) <<' , ' //输出矩形的特征参数
        <<rect.GetY ( ) <<' , '
        <<rect.GetH ( ) <<' , '
        <<rect.GetW ( ) <<endl;
    return 0;
}
```

# 说明

关于私有派生，就是要理解，通过私有派生，基类的任何成员在派生类中都是私有的，这样就改变了基类中公有成员（以及保护成员）在派生类中的访问权限。这样一来，派生类中继承来的基类成员均成为它的私有成员，使得通过这个派生类再次派生出新类时，这些成员在新派生类中均成为不可访问成员，使派生类的使用很不方便。所以私有派生在实用中用得不多。

## 2. 公有继承的访问规则

- 基类的`public`成员和`protected`成员被继承到派生类中仍作为派生类的`public`成员和`protected`成员,派生类的其他成员可以直接访问它们。但是,类的外部使用者只能通过派生类的对象访问继承来的`public`成员。
  - 基类的`private`成员在私有派生类中是不可直接访问的。所以无论是派生类成员还是通过派生类的对象,都无法直接访问从基类继承来的`private`成员,但是可以通过基类提供的`public`成员函数间接访问它们。
- `派生类的对象`只能访问基类的`public`成员。

## 例4.2 公有继承举例

```
class Point    //基类Point类的声明
{public:      //公有函数成员
    void InitP(float xx=0, float yy=0)
    {X=xx;Y=yy;}
    void Move(float xOff, float yOff)
    {X+=xOff;Y+=yOff;}
    float GetX ( ) {return X;}
    float GetY ( ) {return Y;}
private:     //私有数据成员
    float X,Y;
protected: //受保护数据成员
    int a,b;
};
```

```
class Rectangle: public Point //派生类声明
```

```
{  
public: //新增公有函数成员
```

```
void InitR(float x, float y, float w, float h)
```

```
{InitP(x,y);W=w;H=h;}//调用基类公有成员函数
```

```
float GetH ( ) const {return H;}
```

```
float GetW ( ) const {return W;}
```

```
private: //新增私有数据成员
```

```
float W,H;
```

```
};
```

a=2;可否?

Move	public
GetX	public
GetY	public
X,Y	inaccessible
a,b	protected

```

#include<iostream.h>
#include<math.h>int main ( )
{ Rectangle rect;
  rect.InitR(2,3,20,10);
  //通过派生类对象访问基类公有成员
  rect.Move(3,2);
  cout<<rect.GetX ( )      <<',';
    << rect.GetY ( )      <<',';
    <<rect.GetH ( ) <<',';
    <<rect.GetW ( ) <<endl;
  return 0;
}

```

rect.a=2;可否?

rect.X=2;可否?



# 说明1

- 一个派生类的基类成员是私有成员的时候，虽然基类的成员已被派生类继承，但C++规定，这个派生类仍不能直接访问基类的私有成员，只能通过基类的公有成员作为接口去访问，这是符合数据封装的思想的。基类的私有成员在派生类中就是“不可访问(inaccessible)成员”。
- 为了能够在派生类中访问基类所有成员，又使数据封装得以实现，就引入了“保护成员”的概念。就是在类的定义中，用protected来说明类成员，而不是用“private”，这样的成员就是类的保护成员。保护成员对于派生类的成员函数而言是公有成员，而对于其他函数就仍是私有成员。

## 说明2

在公有派生的情况下，一个派生类的对象可以作为基类的对象来使用的地方(在公有派生的情况下，每一个派生类的对象都是基类的一个对象----它继承了基类的所有的成员并没有改变其访问权限。想一想，一条黑狗是不是可以当作一条狗来使用呢?).

- 1.派生的对象可以赋给基类的对象。如：(约定derived是从类base公有派生而来的)

```
derived d;
```

```
base b;
```

```
b=d;
```

# 说明2

- 2. 派生类的对象可以初始化基类的引用。如：

```
derive d;  
base &br=d;
```

- 3. 派生类对象的地址可以赋给指向基类的指针，如：

```
derived d;  
base *pb=&d;
```

- 在后两种情况下，通过指针或引用只能访问对象d中所继承的基类成员。

## 例4.3

```
class base{
public: test1(){cout << "test1:base" <<endl;} };
class derived : public base
{
    public: test2(){cout << test2:derived" <<endl;}
};
void main()
{ derived d; base &b = d;
    b.test2();
    b.test1();
}
```

表4-3 公有继承的访问规则

基类成员	Private成员	public成员	protected成员
内部访问 对象访问	不可访问 不可访问	可访问 可访问	可访问 不可访问

### 3. 保护继承的访问规则

- 基类的`public`成员和`protected`成员被继承到派生类中都作为派生类的`protected`成员,派生类的其他成员可以直接访问它们,但是类的外部使用者不能通过派生类的对象来访问它们。
- 基类的`private`成员在私有派生类中是不可直接访问的,所以无论是派生类成员还是通过派生类的对象,都无法直接访问基类的`private`成员。
- 通过派生类的对象不能访问基类中的任何成员

表4-4 保护继承的访问规则

基类成员	private成员	public成员	protected成员
内部访问 对象访问	不可访问 不可访问	可访问 不可访问	可访问 不可访问

# protected 成员的特点与作用

- 对建立其所在类对象的模块来说（水平访问时），它与 private 成员的性质相同。
- 对于其派生类来说（垂直访问时），它与 public 成员的性质相同。
- 既实现了数据隐藏，又方便继承，实现代码重用。



## 例4.4 保护继承举例

```
class Rectangle: protected Point //派生类声明
{public: //新增外部接口
    void InitR(float x, float y, float w, float h)
    {InitP(x,y);W=w;H=h;} //访问基类公有成员
    void Move(float xOff, float yOff) {Point::Move(xOff,yOff);}
    float GetX ( ) {return Point::GetX ( ) ;}
    float GetY ( ) {return Point::GetY ( ) ;}
    float GetH ( ) {return H;}
    float GetW ( ) {return W;}
private: //新增私有数据
    float W,H;
};
```

InitP	protected
Move	protected
GetX	protected
GetY	protected
X,Y	inaccessible
a,b	protected

# 基类与派生类的关系

- 派生类是基类的具体化
- 派生类是基类定义的延续
- 派生类是基类的组合

# 1.派生类是基类的具体化

- 类的层次通常反映了客观世界中某种真实的模型。  
如，定义输入设备为基类，而键盘、鼠标和数字化板将是派生类。

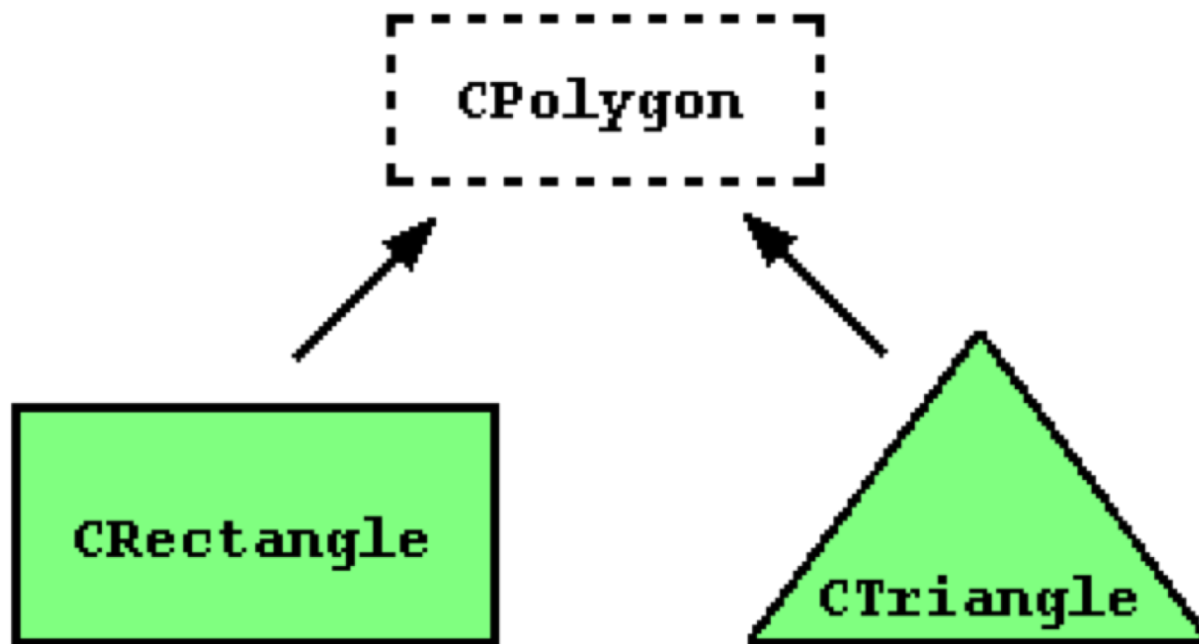
## 2.派生类是基类定义的延续

- 先定义一个抽象基类，该基类中有些操作并未实现。然后定义非抽象的派生类，实现抽象基类中定义的操作。后面要讲述的虚函数就属此类情况。这时，派生类是抽象的基类的实现，即可看成是基类定义的延续。这也是派生类的一种常用的方法。

### 3.派生类是基类的组合

- 在多继承时，一个派生类有多于一个的基类，这时派生类将是所有基类行为的组合。
- 派生类将其本身与基类区别开来的方法是添加数据成员和成员函数。因此，继承的机制将使得在创建新类时，只需说明新类与已有类的区别，从而大量原有的程序代码都可以复用，所以有人称类为“可复用的软件构件”。

例子：



## 例4.5 :

```
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) { width=a; height=b;}
};
class Rectangle: public Polygon {
public:
    int area () { return width * height; }
};
```

```
class Triangle: public Polygon {  
public:  
    int area () { return width * height / 2; }  
};  
  
int main () {  
    Rectangle rect;  
    Triangle trgl;  
    rect.set_values (4,5);  
    trgl.set_values (4,5);  
    cout << rect.area() << '\n';  
    cout << trgl.area() << '\n';  
    return 0;  
}
```



## 例4.6 :

// constructors and derived classes

```
#include <iostream>
```

```
using namespace std;
```

```
class Mother {
```

```
public:
```

```
    Mother () { cout << "Mother: no parameters\n"; }
```

```
    Mother (int a) { cout << "Mother: int parameter\n"; }
```

```
};
```

```
class Daughter : public Mother {
```

```
public:
```

```
    Daughter (int a) { cout << "Daughter: int parameter\n\n"; }
```

```
};
```

```
class Son : public Mother {  
public:  
    Son (int a) : Mother (a) { cout << "Son: int  
        parameter\n\n"; }  
};  
int main () {  
    Daughter kelly(0);  
    Son bud(0);  
    return 0;  
}
```

## 例4.7 :

```
// multiple inheritance
#include <iostream>
using namespace std;
class Polygon {
protected:
    int width, height;
public:
    Polygon (int a, int b) : width(a), height(b) {}
};
class Output {
public:
    static void print (int i);
};
void Output::print (int i) { cout << i << '\n'; }
```

```
class Rectangle: public Polygon, public Output {
public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area () { return width*height; }
};

class Triangle: public Polygon, public Output {
public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area () { return width*height/2; }
};

int main () {
    Rectangle rect (4,5);
    Triangle trgl (4,5);
    rect.print (rect.area());
    Triangle::print (trgl.area());
    return 0;
}
```

## 4.2 派生类的构造函数和析构函数

- 基类的构造函数和析构函数不能被继承，一般派生类要加入自己的构造函数。
- 派生类构造函数和析构函数的执行顺序
  - 通常情况下,当创建派生类对象时,首先执行基类的构造函数,随后再执行派生类的构造函数;
  - 当撤消派生类对象时,则先执行派生类的析构函数,随后再执行基类的析构函数。

例4.8 基类和派生类的构造函数及析构函数的执行顺序。

```
#include <iostream>
class Base{
public:
    Base(){ cout<<"Constructing base class\n"; }
                //基类的构造函数
    ~Base(){ cout<<"Destructing baes class\n"; }
                //基类的析构函数
};
class Derive:public Base{
public:
    Derive(){cout<<"Constructing derived class\n";}
                //派生类的构造函数
    ~Derive(){cout<<"Destructing derived class\n";}
                //派生类的析构函数
};
int main()
{   Derive op;   return 0; }
```

# 派生类构造函数和析构函数的构造规则

当基类的构造函数没有参数，或没有显式定义构造函数时，派生类可以不向基类传递参数，甚至可以不定义构造函数；当基类含有带参数的构造函数时，派生类必须定义构造函数，以提供把参数传递给基类构造函数的途径。

在C++中,派生类构造函数的一般格式为:

派生类名(参数总表):基类名(参数表)

```
{  
    // 派生类新增成员的初始化语句  
}
```

基类构造函数的参数，通常来源于派生类构造函数参数总表，也可以用常量值

例4.9 当基类含有带参数的构造函数时，派生类构造函数的构造方法。

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

```
class Base {
```

```
    public:
```

```
        Base(int n)                //基类的构造函数
```

```
        {
```

```
            cout<<"Constructing base class\n";
```

```
            i=n;
```

```
        }
```

```
        ~Base()                    //基类的析构函数
```

```
        {    cout<<"Destructing base class\n"; }
```

```
        void showi()
```

```
        {    cout<<i<<endl; }
```

```
    private:
```

```
        int i;
```

```
};
```



```

class Derive :public Base{
public:
    Derive(int n,int m):Base(m) // 定义派生类构造函数时,缀上基类的构造函数
    {    cout<<"Constructing derived class"<<endl;
        j=n;    }
    ~Derive()                //派生类的析构函数
    { cout<<"Destructing derived class"<<endl;    }
    void showj()    { cout<<j<<endl;}
private:
    int j;
};

main()
{ Derive obj(50,60);
  obj.showi();
  obj.showj();
  return 0;
}

```

当派生类中含有内嵌对象成员时,其构造函数的一般形式为:  
派生类名(参数总表):基类名(参数表1),内嵌对象名1(内嵌对象参数表1),...,  
内嵌对象名n(内嵌对象参数表n)

```
{  
    // 派生类新增成员的初始化语句  
}
```

在定义派生类对象时,构造函数的执行顺序如下:

- 调用基类的构造函数;
- 调用内嵌对象成员（子对象类）的构造函数（有多个对象成员时,调用顺序由它们在类中声明的顺序确定）;
- 派生类的构造函数体中的内容

撤消对象时,析构函数的调用顺序与构造函数的调用顺序正好相反。

## 【例4.10】 内嵌对象成员时派生类构造函数和析构函数的执行顺序

```
#include <iostream>
class Base {
    int x;
public :
    Base(int i)                //基类的构造函数
    { x=i;
      cout<<"Constructing base class\n";
    }
    ~Base()                    //基类的析构函数
    { cout<<"Destructing base class\n"; }
    void show()
    { cout<<" x=" <<x<<endl; }
};
```

```
class Derived : public Base
{
    Base d;    //d为基类对象，作为派生类的对象成员
public :
    Derived(int i) : Base(i),d(i) //派生类的构造函数，
    {           //缀上基类构造函数和对象成员构造函数
        cout<<"Constructing derived class\n";
    }
    ~Derived()    //派生类的析构函数
    { cout<<"Destructing derived class\n"; }
};
main()
{
    Derived obj(123);
    obj.show();
    return 0;
}
```

## 需要注意的几点：

- 当基类构造函数不带参数时，派生类可不定义构造函数，但基类构造函数带有参数，则派生类必须定义构造函数。
- 若基类使用缺省构造函数或不带参数的构造函数，则在派生类中定义构造函数时可略去“：基类构造函数名（参数表）”
- 如果派生类的基类也是一个派生类，每个派生类只需负责其直接基类的构造，依次上溯。
- 由于析构函数是不带参数的，在派生类中是否定义析构函数与它所属的基类无关，基类的析构函数不会因为派生类没有析构函数而得不到执行，基类和派生类的析构函数是各自独立的。

# 派生类构造函数使用中应注意的问题

- 使用派生类构造函数时应注意如下几个问题：
  - (1) 派生类构造函数的定义中可以省略对基类构造函数的调用，其条件是在基类中必须有缺省的构造函数或者根本没有定义构造函数。
  - (2) 当基类的构造函数使用一个或多个参数时，则派生类必须定义构造函数，提供将参数传递给基类构造函数途径。
    -

## 基类构造函数带有参数，则派生类必须定义构造函数

```
class base{
    int i;
public :
    base(int n)
    {   cout<<"Constructing base class\n";           i=n;       }
    void showi()
    {   cout<<i<<"\n"; }
};

class derived : public base{
    int j;
public :
    derived(int n):base(n)
    {   cout<<"constructing derived class\n";       j=0;   }
    void showj()
    {   cout <<j<<"\n"; }
};
```

## 【例4.11】 说明派生类构造函数和析构函数的构造规则

```
#include<iostream.h>
class First{
private :
    int a,b;
public :
    First(){a=0;b=0;}
    First(int x,int y)
    {   a=x;b=y;}
    ~First(){ }
    void print()
    {
        cout<<"\n a="<<a<<" b="<<b;
    }
};
```



```
class Second : public First{
private :
    int c,d;
public :
    Second() : First(1,1)
    { c=0; d=0; }
    Second(int x,int y) : First(x+1,y+1)
    { c=x; d=y; }
    Second(int x,int y,int m,int n) : First(m,n)
    { c=x; d=y; }
    ~Second(){ }
    void print() //覆盖基类同名成员函数
    {
        First::print();
        cout<<" c="<<c<<" d="<<d;
    }
};
```

```
class Third : public Second{
private :
    int e;
public :
    Third()
    { e=0; }
    Third(int x,int y,int z) : Second(x,y)
    { e=z; }
    Third(int x,int y,int z,int m,int n) : Second(x,y,m,n)
    { e=z; }
    ~Third(){ }
    void print() //覆盖基类同名成员函数
    {
        Second :: print();
        cout<<" e="<<e;
    }
};
```

```
main()
{
    First obj0;
    obj0.print();
    Second obj1;
    obj1.print();
    Second obj2(10,10,20,20);
    obj2.print();
    Second obj3(10,10);
    obj3.print();
    Third obj4;
    obj4.print();
    Third obj5(10,10,20);
    obj5.print();
    Third obj6(100,100,200,50,50);
    obj6.print();
    return 0;
}
```

```
a=0 b=0
a=1 b=1 c=0 d=0
a=20 b=20 c=10 d=10
a=11 b=11 c=10 d=10
a=1 b=1 c=0 d=0 e=0
a=11 b=11 c=10 d=10 e=20
a=50 b=50 c=100 d=100 e=200
```

## 4.4 多继承

- 单继承
  - 派生类只从一个基类派生。
- 多继承
  - 派生类从多个基类派生。
- 多重派生
  - 由一个基类派生出多个不同的派生类。
- 多层派生
  - 派生类又作为基类，继续派生新的类。

# 多继承的声明

有两个以上基类的派生类声明的一般形式如下:

```
class 派生类名:继承方式1 基类名1,...,继承方式n  
基类名n{  
    // 派生类新增的数据成员和成员函数  
};
```

```
Class z : public x, y{  
    //类z公有继承了类 x , 私有继承了类y  
    //...  
};  
class z : x, public y{  
    //类z私有继承了类x , 公有继承了类y  
    //...  
};  
class z : public x, public y{  
    //类z公有继承了类x和类y  
    //...  
};
```

## 【例4.12】多继承情况下的访问特性

```
#include<iostream.h>
class A      {
    public:
        void setA(int x)
        {    a=x;  }
        void printA()
        {    cout<<"a="<<a<<endl;  }
private:
    int a;
};
class B {
    public:
        void setB(int x)
        {    b=x;  }
        void printB()
        {    cout<<"b="<<b<<endl;    }
private:
    int b;
};
```

```

class C : public A, private B {
    public :
        void setC(int x, int y)
        {
            c=x;
            setB(y);
        }
        void printC()
        {
            printB();
            cout<<"c="<<c<<endl;
        }
    private :
        int c;
};

```

```

void main()
{
    C obj;
    obj.setA(3);
    obj.printA();
    obj.setB(4); //错误
    obj.printB(); //错误
    obj.setC(6, 8);
    obj.printC();
}

```

a=3  
b=8  
c=6



## 对基类成员的访问必须是无二义性

```
class X{
public:
    int f();
};
class Y{
public:
    int f();
    int g();
};
class Z : public X, public Y {
public:
    int g();
    int h();
};
Z obj;
```

`obj.f();` //二义性错误, 不知调用的是类X的f(), 还是类Y的f()。

使用类名限定可以消除二义性, 例如:

```
obj.X::f();    //调用类x的f()
obj.Y::f();    //调用类y的f()
```

# 多继承的构造函数与析构函数

多继承构造函数定义的一般形式如下:

派生类名(参数总表):基类名1(参数表1),基类名2(参数表2),...,基类名n(参数表n)

{

// 派生类新增成员的初始化语句

}

构造函数的执行顺序同单继承：

- 先执行基类构造函数，再执行对象成员的构造函数，最后执行派生类构造函数。
- 必须同时负责该派生类所有基类构造函数的调用。派生类的参数个数必须包含完成所有基类初始化所需的参数个数。
- 处于同一层次各基类构造函数执行顺序，取决于声明派生类时所指定各基类的顺序，与派生类构造函数中所定义的成员初始化列表的各项顺序无关。

```

class Hard{
protected:
    char bodyname[20];
public:
    Hard(char *bdnm);           // 基类Hard构造函数
    //...                       };

class Soft{
protected:
    char os[10];
    char Lang[15];
public:
    Soft(char *o,char *lg);     // 基类Soft的构造函数
    //...                       };

class System:public Hard,public Soft {
private:
    char owner[10];
public:
    System(char *ow,char *bn,char *o,char *lg)
        : Hard(bn),Soft(o,lg);
        //缀上了基类Hard和Soft的构造函数
    //...                       };

```

```

class window {                                //定义窗口类window
    //...
public:
    window(int top,int left,int bottom,int right);
    ~window();
    //... };

class scrollbar{                               //定义滚动条类scrollbar
    //...
public:
    scrollbar(int top,int left,int bottom,int right);
    ~scrollbar();
    //... };

class scrollbarwind : window, scrollbar {      //定义派生类
    //...
public:
    scrollbarwind(int top,int left,int bottom,int right);
    ~scrollbarwing();
    //... };

scrollbarwind::scrollbarwind(int    top,int    left,int    bottom,int
right)    :    window(top,left,bottom,right), scrollbar(top,right-
20,bottom,right)
    {    //...    }

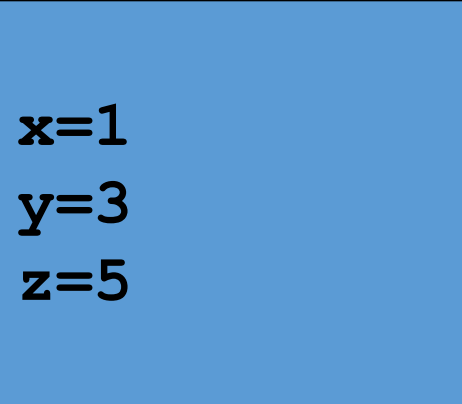
```

## 【例4.13】多继承中的构造函数的定义方法

```
#include<iostream.h>
class Base1 {
public:
    Base1(int sx)                // 基类Base1的构造函数
    {    x=sx;    }
    int getx()
    {    return x;    }
private:
    int x;
};
class Base2 {
public:
    Base2(int sy)                // 基类Base2的构造函数
    {    y=sy;    }
    int gety()
    {    return y;    }
private:
    int y;
};
```

```
class Derived : public Base1, private Base2 {
public:
    Derived(intsx, intsy, intsz) : Base1(sx), Base2(sy)
    {    z=sz;    }
    int getz()
    {    return z; }
    int gety()
    {    return Base2::gety(); }
private:
    int z;
};
```

```
main()  
  
{  
    Derived obj(1,3,5);  
    int ma=obj.getx();  
    cout<<"x="<<ma<<endl;  
    int mb=obj.gety();  
    cout<<"y="<<mb<<endl;  
    int mc=obj.getz();  
    cout<<"z="<<mc<<endl;  
    return 0;  
}
```



**x=1  
y=3  
z=5**

## 4.5 虚基类

### 1.为什么要引入虚基类

当某一个类的多个直接基类是从另一个共同基类派生而来时，这些直接基类中从上一级基类继承来的成员就拥有相同的名称。在派生类的对象中，这些同名成员在内存中同时拥有多个拷贝。如何进行分辨呢？

一种方法就是使用作用域标示符来唯一表示它们。

另一种方法是定义虚基类，使派生类中只保留一份拷贝。



## 例4.14 虚基类的引例

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

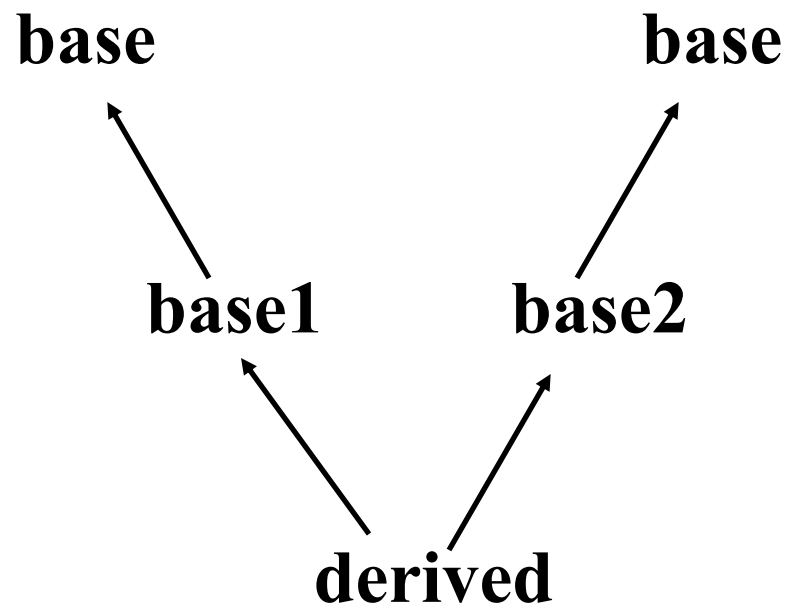
```
class base {  
    public:  
        base(){ a=5; cout<<"base a="<<a<<endl; }  
    protected:  
        int a;  
};  
class base1:public base{  
    public:  
        base1()  
        { a=a+10; cout<<"base1 a="<<a<<endl; }  
};  
class base2:public base{  
    public:  
        base2(){a=a+20; cout<<"base2 a="<<a<<endl;}  
};
```

```
class derived:public base1,public base2{
    public:
        derived()
        {   cout<<"base1::a="<<base1::a<<endl;
            cout<<"base2::a="<<base2::a<<endl;
        }
};

main()
{   derived obj;
    return 0;
}
```

程序运行结果如下：

```
base a=5
base1 a=15
base a=5
base2 a=25
base1::a=15
base2::a=25
```



非虚基类的类层次图

# 虚基类的概念

```
class derived:public base1,public base2{  
    public:  
        derived()  
        {   cout<<"base1::a="<<a<<endl;  
        }  
}
```

可使用虚基类解决上例访问类base成员a的二义性问题。

虚基类的声明：

位置：定义派生类时声明。

其语法形式如下：

```
class 派生类名:virtual 继承方式 类名{  
    //...  
}
```

## 例4-15 虚基类的使用。

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

```
class base {
```

```
    public:
```

```
        base( ) { a=5; cout<<"base a="<<a<<endl;}
```

```
protected:
```

```
    int a;
```

```
};
```

```
class base1: virtual public base{
```

```
    public:
```

```
        base1( ) { a=a+10; cout<<"base1 a="<<a<<endl;}
```

```
};
```

```
class base2: virtual public base{
```

```
    public:
```

```
        base2( ) { a=a+20; cout<<"base2 a="<<a<<endl;}
```

```
};
```

```
class derived:public base1,public base2 {  
    public:  
        derived( ) { cout<<"derived a="<<a<<endl;}  
};  
main( )  
{ derived obj; return 0;  
}
```

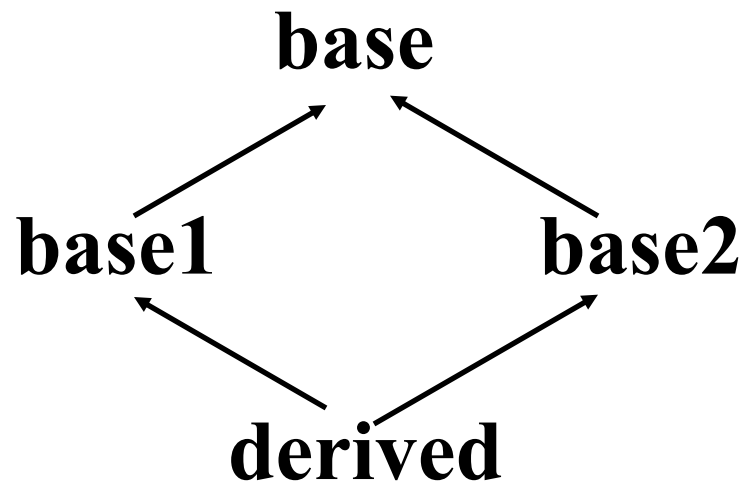
程序运行结果如下：：

base a=5

base1 a=15

base2 a=35

derived a=35



虚基类的类层次图

# 虚基类的初始化

虚基类的初始化与一般的多继承的初始化在语法上是一样的，但构造函数的调用顺序不同。在使用虚基类机制时应该注意以下几点：

- (1) 如果在虚基类中定义有带形参的构造函数,并且没有定义缺省形参的构造函数,则整个继承结构中,所有直接或间接的派生类都必须在构造函数的成员初始化表中列出对虚基类构造函数的调用,以初始化在虚基类中定义的数据成员。
- (2) 建立一个对象时,如果这个对象中含有从虚基类继承来的成员,则虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。该派生类的其他基类对虚基类构造函数的调用都自动被忽略。



- (3) 若同一层次中同时包含虚基类和非虚基类,应先调用**虚基类**的构造函数,再调用**非虚基类**的构造函数,最后调用**派生类**构造函数;
- (4) 对于**多个虚基类**,构造函数的执行顺序仍然是**先左后右,自上而下**;
- (5) 对于**非虚基类**,构造函数的执行顺序仍是**先左后右,自上而下**;
- (6) 若虚基类由非虚基类派生而来,则仍然先调用基类构造函数,再调用派生类的构造函数。

```
class X:public Y,virtual public Z
```

```
{//...};
```

```
X one;
```

定义类X的对象one后,将产生如下的调用次序

```
Z0;  
Y0;  
X0;
```

## 【例4.16】含有虚基类的派生类构造函数的执行顺序

```
#include<iostream.h>
class base {
public:
    base(int sa)
    {
        a=sa;
        cout<<"Constructing base"<<endl;    }
private:
    int a;
};
class base1:virtual public base{
public:
    base1(int sa,int sb):base(sa)
    {
        b=sb;
        cout<<"Constructing baes1"<<endl;
    }
private:
    int b;
};
class base2:virtual public base{
public:
    base2(int sa ,int sc):base(sa)
    {
        c=sc;
        cout<<"Constructing baes2"<<endl;    }
private:
    int c;
};
```

## 虚基类的构造函数

- 为了初始化基类的子对象，派生类的构造函数要调用基类的构造函数。对于虚基类来讲，由于派生类的对象中只有一个虚基类子对象。为保证虚基类子对象只被初始化一次，这次虚基类构造函数必须只被调用一次。由于继承结构的层次可能很深，规定将在建立对象时所指定的类称为**最（远）派生类**。

- C++规定，虚基类子对象是由最派生类的构造函数通过调用虚基类的构造函数进行初始化的。如果一个派生类有一个直接或间接的虚基类，那么派生类的构造函数的成员初始列表中必须列出对虚基类构造函数的调用，如果未被列出，则表示使用该虚基类的缺省构造函数来初始化派生类对象中的虚基类子对象。

- 从虚基类直接或间接继承的派生类中的构造函数的成员初始化列表中都要列出这个虚基类构造函数的调用。但是，只有用于建立对象的那个最派生类的构造函数调用虚基类的构造函数，而该派生类的基类中所列出的对这个虚基类的构造函数调用在执行中被忽略，这样便保证了对虚基类的子对象只初始化一次。
- C++又规定，在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用，则虚基类的构造函数先于非虚基类的构造函数的执行。

```
class derived:public base1 , public base2 {
public:
    derived(int sa , int sb, int sc, int sd) :
        base(sa) , base1(sa, sb) , base2(sa, sc)
    {
        d=sd;
        cout<<"Constructing derived"<<endl;
    }
private:
    int d;
};
main()
{
    derived obj(2,4,6,8);
    return 0;
}
```

**Constructing base**  
**Constructing base1**  
**Constructing base2**  
**Constructing derived**

有关虚基类的另两点说明：

(1). 关键字`virtual`与继承方式关键字(`public`或`private`)的先后顺序无关紧要，它只说明是“虚拟继承”。下面二个虚继承方法是等价

```
class derived: virtual public base{  
    //...  
};  
  
class derived: public virtual base{  
    //...  
};
```

(2). 一个基类在作为某些派生类虚基类的同时，又作为另一些派生类的非虚基类，这种情况是允许的。下例说明了这个问题。

```
class B {  
    //...  
};  
class X:virtual public B {  
    //...  
};  
class Y:virtual public B {  
    //...  
};  
class Z:public B {  
    //...  
};  
class AA:public X,public Y,public Z {  
    //...  
};
```



## 4.6 赋值兼容规则

所谓赋值兼容规则是指在需要基类对象的任何地方都可以使用公有派生类的对象来替代。这样,公有派生类实际上就具备了基类的所有特性,凡基类能解决的问题,公有派生类也能解决。

- 一个公有派生类的对象在使用上可以被当作基类的对象,反之则禁止。具体表现在:
  - 派生类的对象可以被赋值给基类对象。
  - 派生类的对象可以初始化基类的引用。
  - 指向基类的指针也可以指向派生类。
- 通过基类对象名、指针只能使用从基类继承的成员

例如,下面声明的两个类:

```
class Base{  
    ...  
};  
class Derived:public Base{  
    ...  
};
```

根据赋值兼容规则, 在基类Base的对象可以使用的任何地方, 都可以用派生类derive的对象来替代, **但只能使用从基类继承来的成员**。以下几种情况是合法的:

(1) 可以用派生类对象给基类对象赋值。例如:

```
Base b;
```

```
Derived d;
```

```
b=d;
```

这样赋值的效果是,对象b中所有数据成员都将具有对象d中对应数据成员的值。

(2) 可以用派生类对象来初始化基类的引用。例如:

```
Derived d;
```

```
Base &br=d;
```

(3) 可以把派生类对象的地址赋值给指向基类的指针。例如:

```
Derived d;
```

```
Base *bptr=&d;
```

这种形式的转换,是在实际应用程序中最常见到的。

(4) 可以把指向派生类对象的指针赋值给指向基类对象的指针。例如:

```
Derived *dptr;
```

```
Base *bptr=dptr;
```

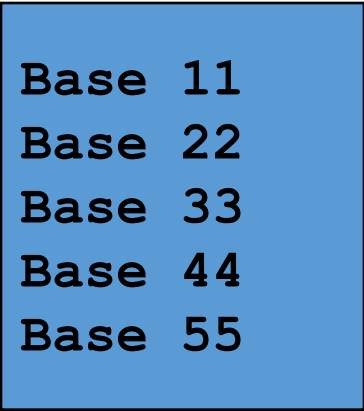
### 例4.17 赋值兼容规则实例。

```
#include <iostream.h>

class base{
public:
    int i;
    base(int x) {i=x;}
    void show()
    {cout<<"base "<<i<<endl;}
};

class derive:public base{
public:
    derive(int x):base(x) { };
    void show()
    {cout<<"derive "<<i<<endl;}
};
```

```
void main()
{
    base b1(11); b1.show();
    derive d1(22);
    b1=d1; b1.show();
    derive d2(33);
    base &b2=d2; b2.show();
    derive d3(44);
    base *b3=&d3; b3->show();
    derive *d4=new derive(55);
    base *b4=d4; b4->show();
    delete d4;
}
```



<b>Base</b>	<b>11</b>
<b>Base</b>	<b>22</b>
<b>Base</b>	<b>33</b>
<b>Base</b>	<b>44</b>
<b>Base</b>	<b>55</b>

说明：

- (1). 声明为指向基类对象的指针可以指向它的公有派生的对象，但不允许指向它的私有派生的对象。
- (2). 允许将一个声明为指向基类的指针指向其公有派生类的对象，但是不能将一个声明为指向派生类对象的指针指向其基类的一个对象。
- (3). 声明为指向基类对象的指针，当其指向公有派生类对象时，只能用它来直接访问派生类中从基类继承来的成员，而不能直接访问公有派生类中定义的成员。

若想访问其公有派生类的特定成员，可以将基类指针用显示类型转换为派生类指针。

```
(1) class base{
    //...
};
class derive : private base
{
    //...
};
void main()
{
    base op1,*ptr;
    derive op2;
    ptr=&op1;
    ptr=&op2; //错误, 不允许指向它的私有派生类对象
    //...
}
```



```
(2) #include<iostream.h>
class Base{
    //...
};
class Derived : public Base{
    //...
};
void main()
{
    Base obj1;
    Derived obj2,*ptr;
    ptr=&obj2;
    ptr=&obj1;// 错误，将派生类指针指向基类对象
    //...
}
```

```
class A {  
    //...  
public :  
    void print1();  
};  
class B : public A {  
    //...  
public :  
    print2();  
};  
void main()  
{
```

**`((B*)ptr)-> print2();`**

```
    A op1,*ptr;           // 定义基类A的对象op1和基类指针ptr  
    B op2;               // 定义派生类B的对象op2  
    ptr=&op1;            // 将指针ptr指向基类对象op1  
    ptr->print1();        // 调用基类函数print1()  
    ptr=&op2;            // 将指针ptr指向派生类对象op2  
    ptr->print1();        // 调用对象op2从其基类继承来的成员函数print1()  
    ptr->print2();        // 错误，基类指针ptr不能访问派生类中定义的  
                          // 成员函数print2()
```

谢谢大家！