



中南大學  
CENTRAL SOUTH UNIVERSITY



# 面向对象程序设计

## Object Oriented Programing

# 第四章 继承与派生


张宝一

地理信息系

zhangbaoyi@csu.edu.cn

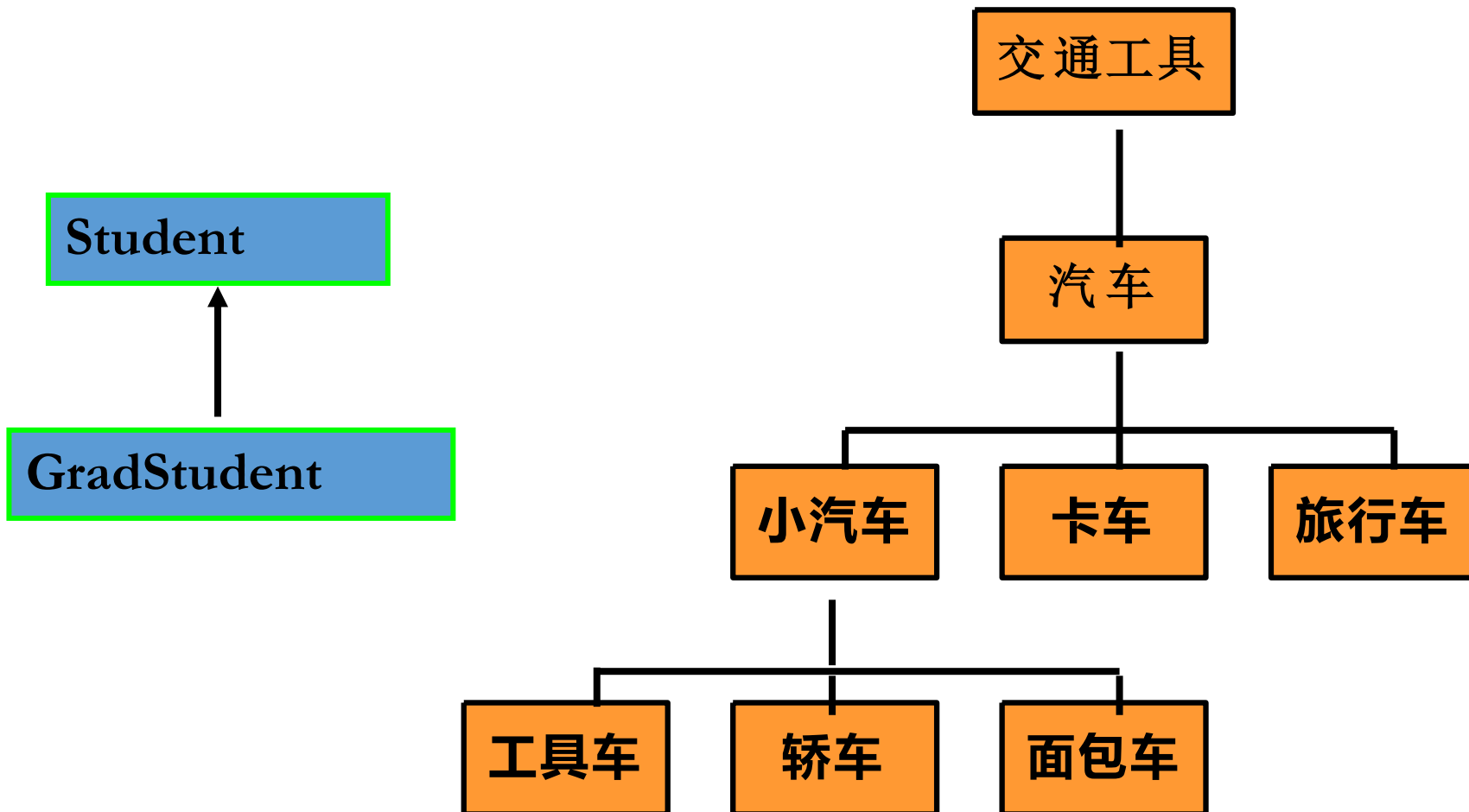


# 第四章：继承与派生

- 继承与派生的基本概念 
- 单继承
- 派生类的访问控制
- 多继承
- 虚基类
- 基类与派生类对象间的赋值规则
- 本章小结

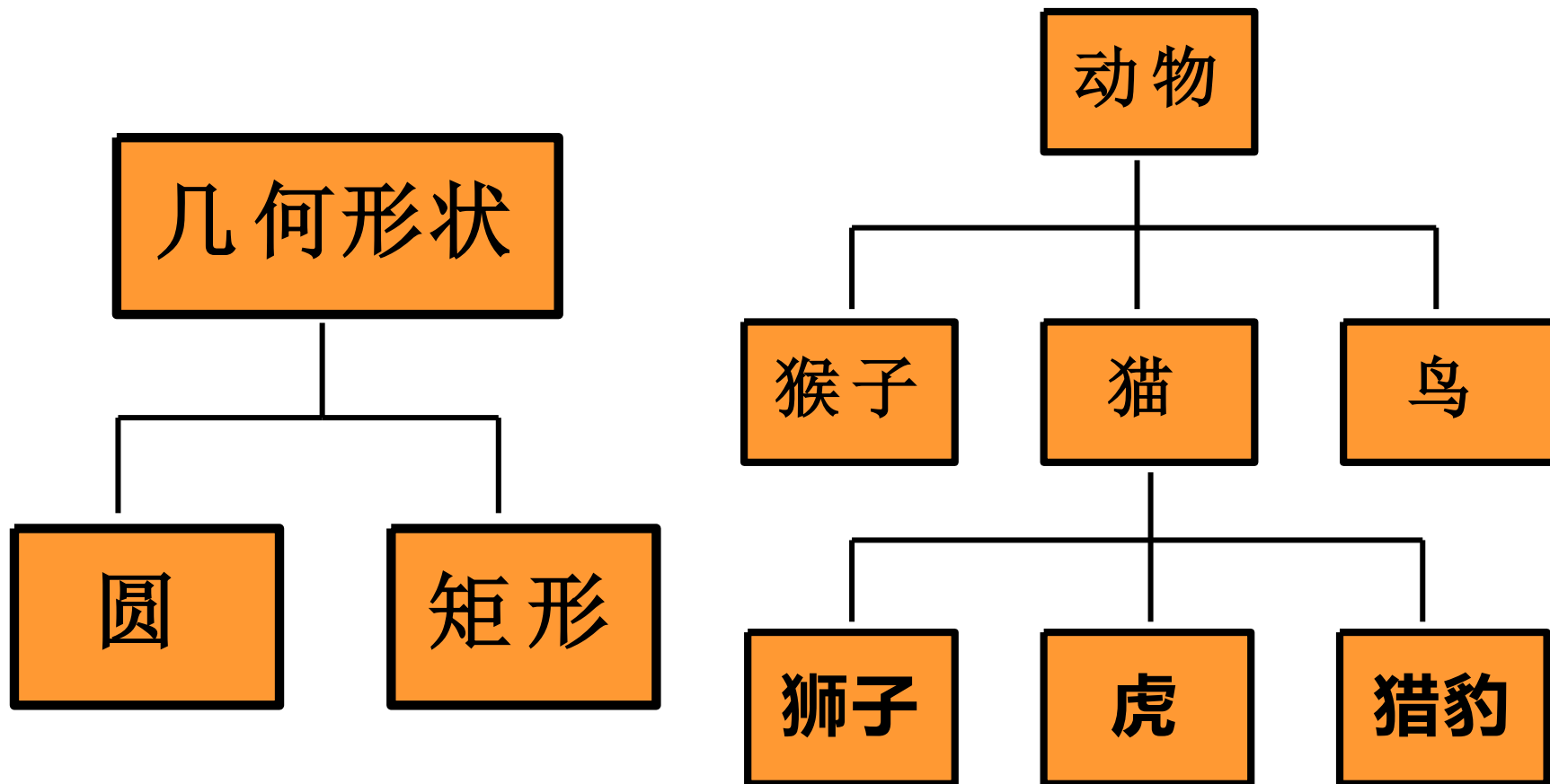
# 4.1 继承和派生的基本概念

## 4.1.1 为什么要使用继承



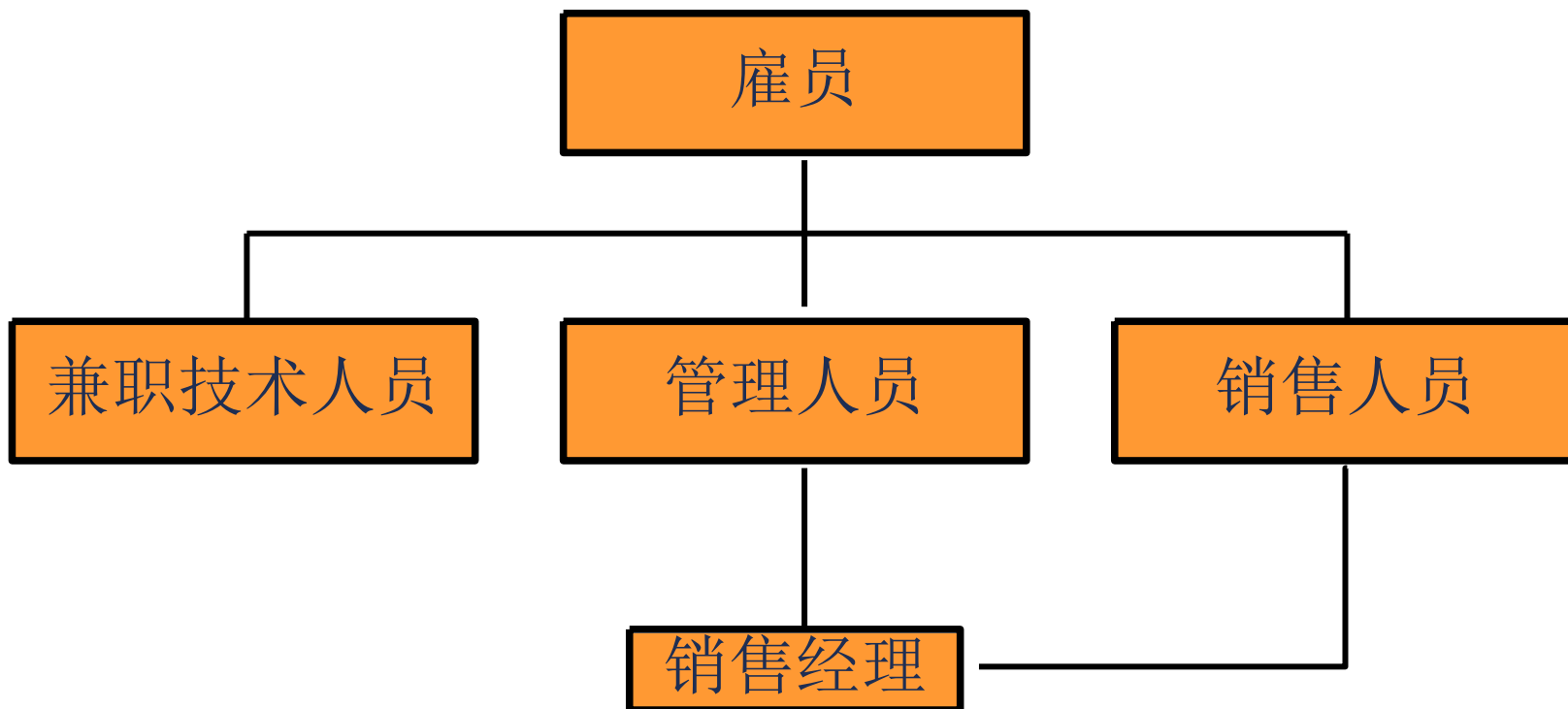
# 4.1 继承和派生的基本概念

## 4.1.1 为什么要使用继承



# 4.1 继承和派生的基本概念

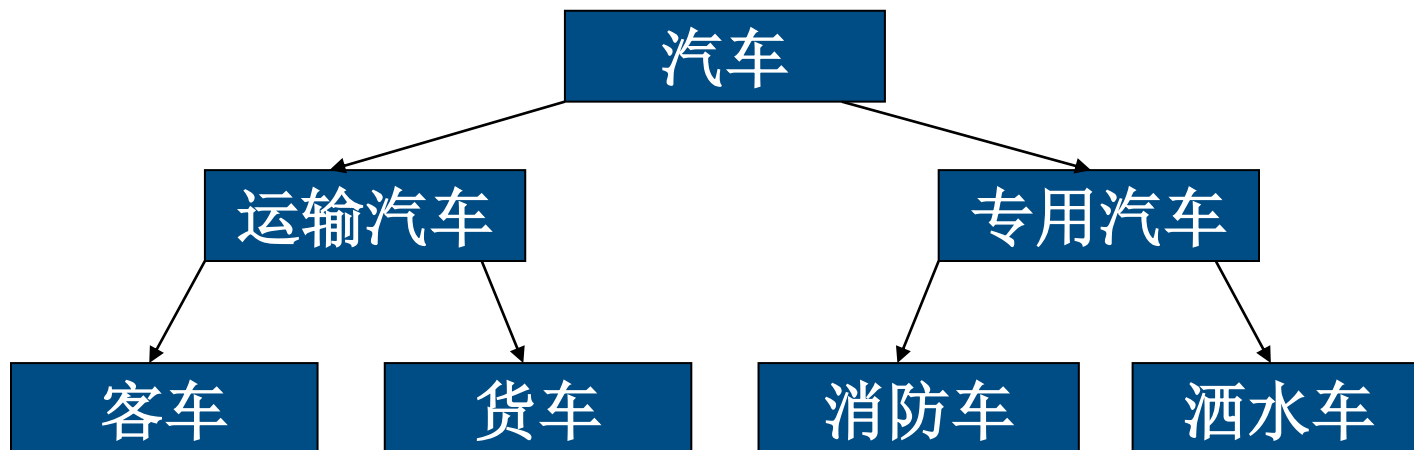
## 4.1.1 为什么要使用继承



# 4.1 继承和派生的基本概念

## 4.1.1 为什么要使用继承

□ 继承与派生问题举例：



- 继承的目的：实现代码重用。
- 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。

# 4.1 继承和派生的基本概念

## 4.1.1 为什么要使用继承

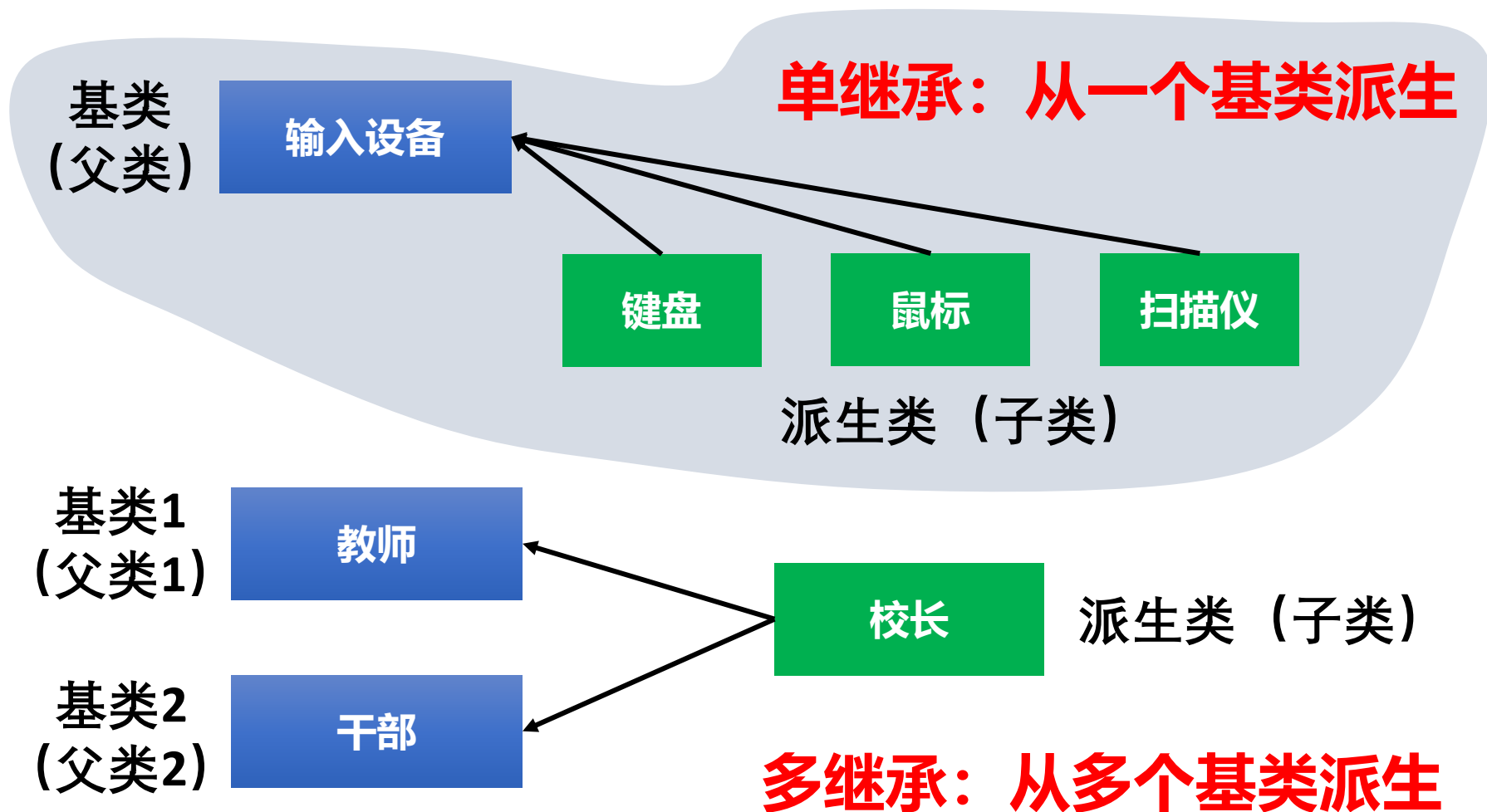
```
class person {  
protected:  
    char name[10];  
    int age;  
    char sex;  
public:  
    void print();  
};
```

```
class employee{  
protected:  
    char name[10];  
    int age;  
    char sex;  
    char department[20];  
    float salary;  
public:  
    void print();  
};
```

➤ **继承机制**：消除冗余，实现代码复用、代码扩充，提高开发效率的一种途径


# 4.1 继承和派生的基本概念

## 4.1.2 继承的种类





# 第四章：继承与派生

- 继承与派生的基本概念
- 单继承 
- 派生类的访问控制
- 多继承
- 虚基类
- 基类与派生类对象间的赋值规则
- 本章小结

## 4.2 单继承

### 4.2.1 单继承： 派生类的声明

//定义一个基类

```
class person{  
protected:  
    char name[10];  
    int age;  
    char sex;  
public:  
    void print();  
};
```

//定义一个派生类

```
class employee :  
    public person {  
protected:  
    char department[20];  
    float salary;  
public:  
    void print1();  
};
```

## 4.2 单继承

### 4.2.1 单继承： 派生类的声明

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  
};
```

□ 继承方式：

- (1) **public**： 公有继承
- (2) **private**： 私有继承
- (3) **protected**： 保护继承

## 4.2 单继承

### 4.2.1 单继承： 派生类的声明

由类person派生出类employee可以采用下面的三种格式之一：

(1) 公有继承

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

(2) 私有继承

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

(3) 保护继承

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

如果没有显式  
定义继承方式，  
则系统默认为  
private

## 4.2 单继承

### 4.2.2 派生类是对基类的有效扩充

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（using用法）。

## 4.2 单继承

### 4.2.2 派生类是对基类的有效扩充


//定义一个基类

```
class person{  
protected:  
    char name[10];  
    int age;  
    char sex;  
public:  
    void print();  
};
```

//定义一个派生类

```
class employee :  
    public person {  
protected:  
    char department[20];  
    float salary;  
public:  
    void print1();  
private:  
    using person::print();  
};
```

# 第四章：继承与派生

- 继承与派生的基本概念
- 单继承
- 派生类的访问控制 
- 多继承
- 虚基类
- 基类与派生类对象间的赋值规则
- 本章小结

## 4.3 派生类的访问控制

### 4.3.1 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：

- ❑ **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。
- ❑ **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。

派生类对基类成员的访问规则：

- （1）私有继承的访问规则；
- （2）公有继承的访问规则；
- （3）保护继承的访问规则。



## 4.3 派生类的访问控制

### 4.3.2 基类成员在派生类中的访问属性

#### (1)成员对基类成员访问控制

• 派生性质    基类访问权限    基类成员成为派生类的访问权限

- public          public          public
- public          protected          protected
- public          private          不可见
- protected          public          protected
- protected          protected          protected
- protected          private          不可见
- private          public          private
- private          protected          private
- private          private          不可见

(2)对象访问控制：对象只能访问类中public成员

## 4.3 派生类的访问控制

### 4.3.3 公有继承的访问规则

当类的继承方式为公有继承时，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成员在公有派生类中是不可直接访问的**，派生类成员与派生类的对象，都无法直接访问从基类继承来的private成员，但是可以通过基类的public成员函数间接访问它们。

基类成员	<b>private成员</b>	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

## 例4.2

```
class base{  
    public:  
        void setxy (int m, int n);  
        void showxy( );  
    private:    int x;  
    protected: int y;  
};  
class derive1: public base {  
    public:  
        void setxyz (int m, int n, int l);  
        void showxyz( )  
            {   cout<< "x= " <<x<<endl; //可以?  
                cout<< "y= " <<y<<endl; //可以?  
                cout<< "z= " <<z <<endl;    }  
    private:  
        int z;    };
```

## 4.3 派生类的访问控制

### 4.3.4 私有继承的访问规则

当类的继承方式为私有继承时，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。

基类成员	<b>private成员</b>	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4.3 `#include <iostream>`

`using namespace std;`

`class base{`

`public:`

`void seta (int sa){ a=sa;}`

`void showa( ){cout<< "a= "<<a<<endl;}`

`protected:`

`int a;`

`};`

`class derive1: private base {`

`public:`

`void setab(int sa, int sb){ a=sa; b=sb;}`

`void showab( ) { cout<< "a= "<<a<<endl; //对吗?`

`cout<< "b= "<<b<<endl;`

`protected:`

`int b;`

`};`

```

class derive2: private derive1 {
    public:
        void setabc(int sa, int sb, int sc){ setab(sa, sb); c=sc;}
        void showabc( ) {
            cout<< "a= " <<a<<endl;    //对吗? showa()可以吗?
            cout<< "b= " <<b<<endl;    // showab()可以吗?
            cout<< "c= " <<c <<endl;
        }
        protected: int c;
};

```

```

int main()
{    base op1; op1.seta(11); op1.showa();
        derive1 op2; op2.setab(22,33);    op2.showab();
        derive2 op3; op3.setabc(44,55,66); op3.showabc();
        return 0;
}

```

## 4.3 派生类的访问控制

### 4.3.5 保护继承的访问规则

当类的继承方式为保护继承时，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成员在保护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。

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4.4 `#include<iostream.h>`

**class base{**

**public:**

**int z;**

**void setx (int a) { x=a;}**

**void showx( ) { cout<< “x= ”<<x<<endl; }**

**private: int x;**

**protected: int y;**

**};**

**class derive: protected base {**

**public:**

**void setall (int a, int b, int c , int d);**

**void showall( ) ;**

**private:**

**int m;**

**};**



```
void derive:: setall (int a, int b, int c , int d){  
    x=a; // setx (a);  
    y=b; z=c; m=d;  
}
```

```
void derive:: showall( ) {  
    cout<< "x= " << x << endl; //showx();  
    cout<< "y= " << y << endl;  
    cout<< "z= " << z << endl;  
    cout<< "m= " << m << endl;  
}
```

```
int main(){  
    derive obj;  
    obj.setall(1,2,3,4);  
    obj.showall();  
    return 0;  
}
```

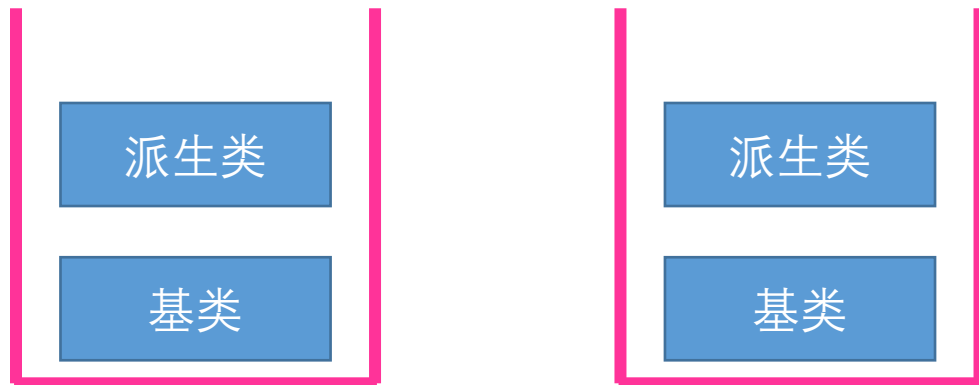
## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

#### □ 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。**



## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

□ 派生类构造函数和析构函数的构造规则：

1. 简单的派生类构造函数和析构函数

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

}

## ■ 例4.5 当基类含有带参数的构造函数时, 派生类构造函数

```
#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;
};

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

## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

□ 派生类构造函数和析构函数的构造规则：

2. 含有内嵌对象的派生类的构造函数  
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

}

## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

- 当基类中声明有缺省形式的构造函数或未声明构造函数时，派生类构造函数的声明中可以省略对基类构造函数的调用。
- 若基类中未声明构造函数，派生类中也可以不声明，全采用缺省形式构造函数。
- 当基类声明有带形参的构造函数时，派生类也应声明带形参的构造函数，提供将参数传递给基类构造函数的途径。

## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

□ 派生类构造函数和析构函数的构造规则：

在定义派生类对象时，构造函数的执行顺序如下：

(1) 调用基类的构造函数；

(2) 调用内嵌对象成员的构造函数（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）；

(3) 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5.6 内嵌对象成员时派生类的构造函数和析构函数

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

```
class Base {
```

```
    int x;
```

```
public:
```

```
    Base(int i) { ..... }           //基类的构造函数
```

```
};
```

```
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
```

```
    .....

```

## 4.3 派生类的访问控制

### 4.3.6 派生类的构造函数和析构函数

#### 派生类构造函数的规则

如果基类拥有构造函数但没有默认构造函数，那么派生类的构造函数必须显式地调用基类的某个构造函数。

**建议：**为每一基类设计一个默认构造函数。

#### 继承机制下的析构函数

在类的层次结构当中，构造函数按基类到派生类的次序执行，析构函数则按照派生类到基类的次序执行，因此，析构函数的执行次序和构造函数的执行次序是相反的。

## 4.3 派生类的访问控制

### 4.3.7 派生类访问基类的同名成员

#### □ 同名函数访问:

C++允许派生类成员与基类成员同名字，称为派生类成员覆盖了基类的同名成员。在派生类中使用这个名字意味着访问在派生类中说明的成员，为了在派生类中使用基类的同名成员，必须在该成员名字前加上基类名和作用域标识符。

```
class X { public: int f( ); };  
class Y : public X {  
public:  
    int f( ); int g( );  
};
```

```
void Y::g( ) { f( ); } //访问函数 Y::f( )  
void main {  
    Y obj;  
    obj.f( ); //访问函数 Y::f( )  
    obj.X::f( ); //访问函数 X::f( )  
}
```

## 4.3 派生类的访问控制

### 4.3.7 派生类访问基类的同名成员

#### □ 访问声明: **using**:

对于**私有继承**，基类的公有成员变为派生类的私有成员。外界不能利用派生类的对象直接调用，**只能通过调用派生类的成员函数(内含调用基类成员函数的语句)来间接调用。**


```
class A {  
    public:  
        void show(){  
            cout<<x;  
        }  
        A(int a):x(a){}  
    private: int x;  
};
```

```
class B: private A {  
    public:  
        B(int x1, int y1):A(x1){y=y1;}  
        using A::show();  
    private: int y; };  
  
B obj(2, 5); obj.show(); // 可以吗?
```

## 4.3 派生类的访问控制

- (1) 下列对派生类的描述中，()是错误的。
  - A. 一个派生类可以作为另一个派生类的基类
  - B. 派生类至少应有一个基类
  - C. 基类中成员的访问权限被派生类继承后都不改变
  - D. 派生类的成员除了自己定义的成员外，还包含了它的基类成员
- (2) 主函数可以访问派生类的对象中它的哪一类基类成员？()
  - A. 公有继承的基类的公有成员
  - B. 公有继承的基类的保护成员
  - C. 公有继承的基类的私有成员
  - D. 保护继承的基类的公有成员

# 第四章：继承与派生

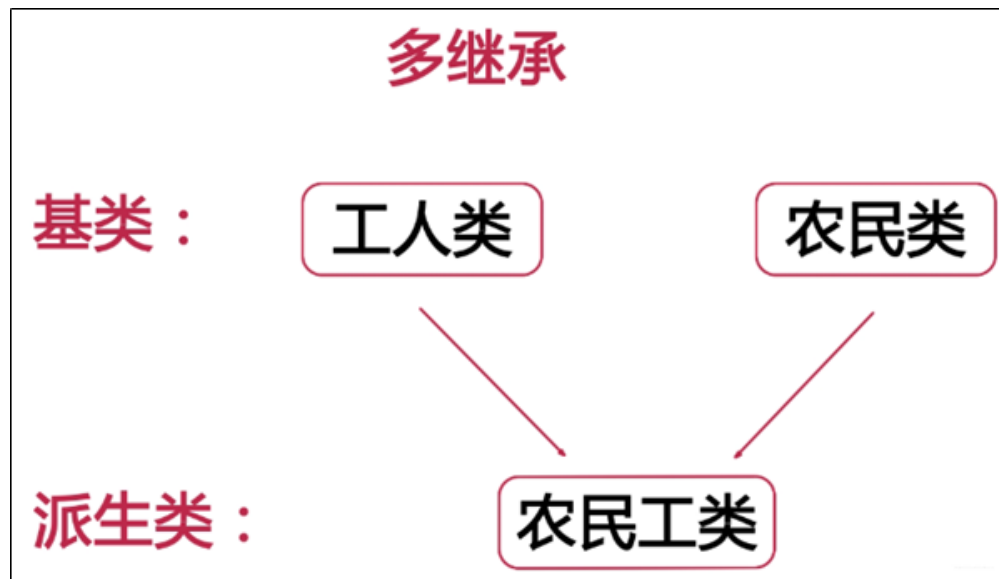
- 继承与派生的基本概念
- 单继承
- 派生类的访问控制
- 多继承 
- 虚基类
- 基类与派生类对象间的赋值规则
- 本章小结

## 4.4 多继承

### 4.4.1 多继承的含义

派生类只有一个基类，这种派生方法称为**单基派生**或**单继承**。

当一个派生类具有多个基类时，这种派生方法称为**多基派生**或**多继承**。



## 4.4 多继承

### 4.4.1 多继承的含义

- 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。



## 4.4 多继承

### 4.4.2 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2> 基类名2 , ...,  
                <继承方式n> 基类名n  
{  
    // 派生类新增的数据成员和成员函数  
};
```

## 4.4 多继承

### □ 多继承举例:

```
class A{  
    public:  
        void setA(int);  
        void showA();  
    private:  
        int a;  
};  
class B{  
    public:  
        void setB(int);  
        void showB();  
    private: int b;  
};
```

```
class C : public A, private B{  
    public:  
        void setC(int, int, int);  
        void showC();  
    private:  
        int c;  
};  
  
C obj;  
obj.setA(); // right or wrong?  
obj.setB(); // right or wrong?
```

## 4.4 多继承

### □ 说明:

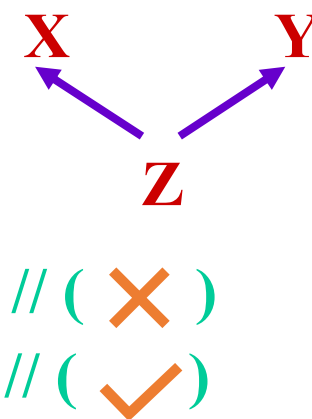
1. 缺省的继承方式是**private**

```
class Z: X, public Y { .....  
};
```

2. 对基类成员的访问必须是无二义的

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

```
int main()  
{  
    Z obj;  
    obj.f();  
    obj.X::f();  
}
```



## 4.4 多继承

### 4.4.3 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

}

## 4.4 多继承

### 4.4.3 多继承的构造函数与析构函数

**多继承构造函数**的执行顺序与单继承下的构造函数执行顺序相同，也是**先执行基类的构造函数，再执行对象成员的构造函数**，处理最后执行派生类的构造函数。

处于同一层的各个基类的构造函数的执行顺序，取决于**声明派生类时所指定的各个基类的顺序，与派生类构造函数中所定义的成员初始化列表中的顺序并没有关系。**

## 例5.1 多继承中构造函数和析构函数的调用顺序

```
#include <iostream>
```

```
using namespace std;
```

```
class Base1{
```

```
    public:
```

```
        Base1(int sx) { x=sx; cout<<"base1 x="<<x<<endl; }
```

```
        ~Base1() { cout<<"base1 Destructor called."<<endl; }
```

```
    private: int x;
```

```
};
```

```
class Base2{
```

```
    public:
```

```
        Base2(int sy) { y=sy; cout<<"base2 y="<<y<<endl; }
```

```
        ~Base2() { cout<<"base2 Destructor called."<<endl; }
```

```
    private: int y;
```

```
};
```


```
class Derived: private Base2, public Base1{
public:
    Derived(int sx, int sy, int sz) : Base1(sx), Base2(sy)
    {   z=sz;
        cout<<"derived::z="<<z<<endl;
    }
    ~Derived () {
        cout<<"derived Destructor called."<<endl; }
private: int z;
};

int main()
{   Derived obj(2,4,6);
    return 0;
}
```

G:\ClassTest\x64\Debug\ClassTest.exe

```
base1 x=2
base2 y=4
derived::z=6
derived Destructor called.
base2 Destructor called.
base1 Destructor called.
```

# 第四章：继承与派生

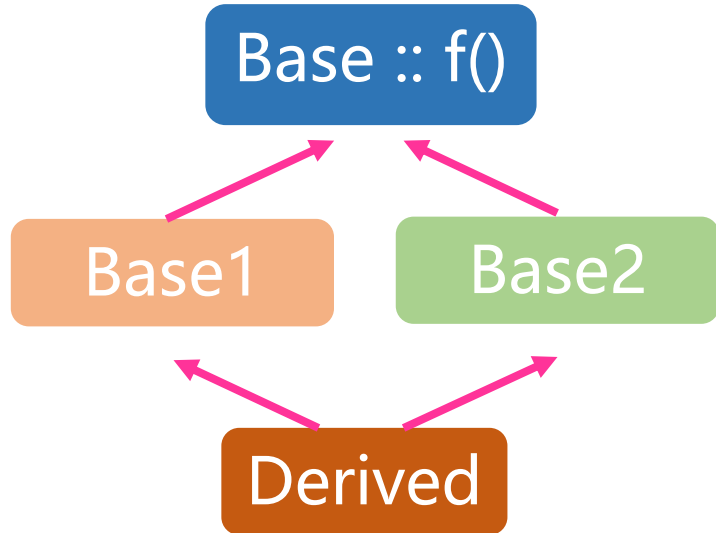
- 继承与派生的基本概念
- 单继承
- 派生类的访问控制
- 多继承
- 虚基类 
- 基类与派生类对象间的赋值规则
- 本章小结



## 4.5 虚基类

### 4.5.1 为什么要引入虚基类

当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有相同的名称。那么在派生类对象中，就存在如何对这些具有相同名称的成员进行分辨的问题。



非虚基类的类层次图

```
Derived d;  
d.f();
```

那么这个f()函数，到底是来源于

Base，还是Base1，或是Base2？

## 4.5 虚基类

### □ 虚基类的引例：

```
class B
{
    public:
        int b;
};
class B1 : public B
{
    private:
        int b1;
};
class B2 : public B
{
    private:
        int b2;
};
```

```
class C : public B1, public B2
{
    public:
        int f ( ) ;
    private:
        int d;
};
```

## 4.5 虚基类

### □ 说明:

下面的访问是二义性的:

C c;

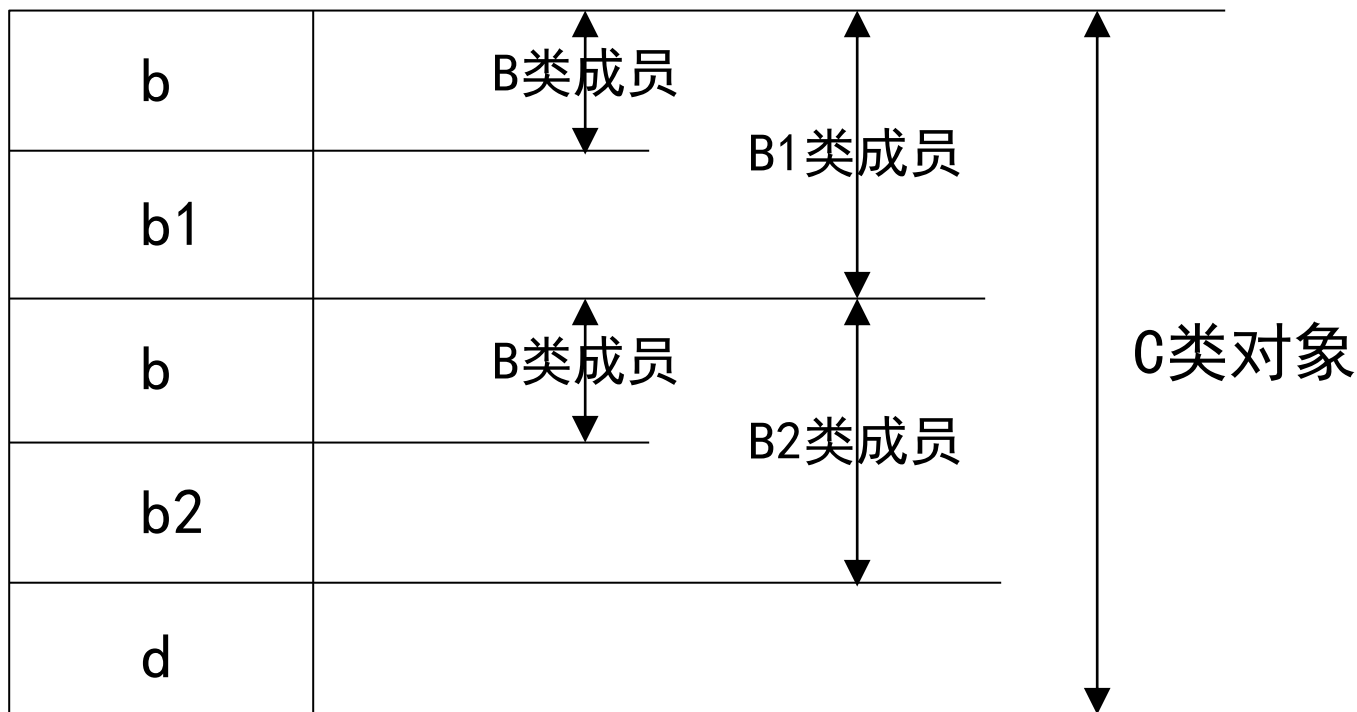
c.b

c.B::b

下面是正确的:

c.B1::b

c.B2::b



## 4.5 虚基类

### 4.5.2 虚基类的声明

如果在上例中公共基类**B**只存在一个拷贝，那么对**b**的访问就不存在二义性。为了达到这一目的，可将**B**说明为虚基类。虚基类的声明是在派生类的声明过程进行，其语法形式如下：

```
class 派生类名 : virtual 继承方式 基类名
{
    //...添加派生类的成员函数和数据成员
}
```

## 4.5 虚基类

### □ 虚基类的声明和使用：

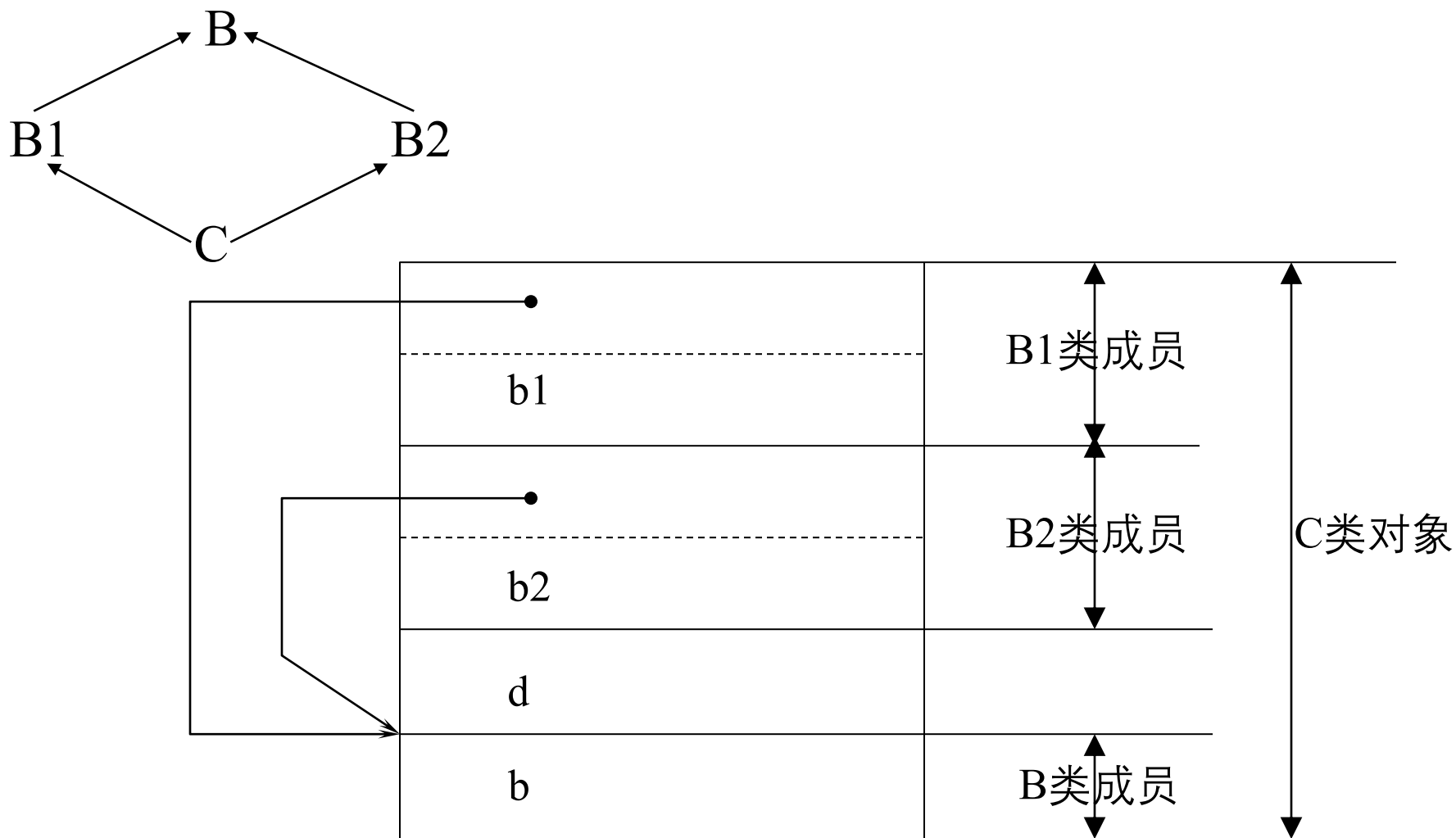
```
class B{ public: int b;};  
class B1 : virtual public B {public : int b1;};  
class B2 : virtual public B {public : int b2;};  
class C : public B1, public B2{public : float  
    d;}
```

下面的访问是正确的：

```
C    cobj;  
cobj.b;
```

## 4.5 虚基类

□ 说明:



## 4.5 虚基类

### 4.5.3 虚基类的初始化

在使用虚基类机制时应该注意以下几点：

- 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式的构造函数, 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;
- 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;

## 4.5 虚基类

### 4.5.3 虚基类的初始化

在使用虚基类机制时应该注意以下几点：

- 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；
- 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；
- 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；
- 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；



## 4.5 虚基类

### 4.5.3 虚基类的初始化

在使用虚基类机制时应该注意以下几点:

- 若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;
- 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;
- 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;
- 关键字virtual 与继承方式关键字(public, private)的先后顺序无关紧要;

**class derived: virtual public base**

**class derived: public virtual base**

- 一个基类在作为某些派生类虚基类的同时, 又可以作为另一些派生类的非虚基类;

## 4.5 虚基类

### 4.5.3 虚基类的初始化

```
#include <iostream.h>
class B0    //声明基类B0
{ public:   //外部接口
    B0(int n)
    {
        nV=n;
        cout<<"Member of D1"<<endl;
    }
    int nV;
    void fun ( ) {cout<<"Member of B0"<<endl;}
};
```

## 4.5 虚基类

### 4.5.3 虚基类的初始化

```
class B1: virtual public B0    {
    public:
        B1(int a) : B0(a)    //给出对虚基类的构造函数的调用
        { cout<<"Member of B1"<<endl; }
        int nV1;
};

class B2: virtual public B0    {
    public:
        B2(int a) : B0(a)    //给出对虚基类的构造函数的调用
        { cout<<"Member of B2"<<endl; }
        int nV2;
};
```

## 4.5 虚基类

### 4.5.3 虚基类的初始化

```
class D1: public B1, public B2    //
    若有多个虚基类，按声明次序{
public:
    D1(int a=0) : B0(a), B1(a), B2(a)

    //给出对虚基类的构造函数的调用
    {cout<<"Member of D1"<<endl;}
    int nVd;
    void fund ( ) {cout<<"Member of
D1"<<endl;}
};
void main ( )    {
    D1 d1;
    d1.nV=2;
    d1.fun ( ) ;}
```

输出: Member of B0 //调用  
B0构造函数

Member of B1  
//调用B1构造函数

Member of B2  
//调用B2构造函数


Member of D1  
//调用D1构造函数

Member of D1  
//调用D1对象的fun()函数

## 4.5 虚基类

- (1) 关于多继承二义性的描述，( )是错误的。
  - A. 派生类的多个基类中存在同名成员时，派生类对这个成员访问可能出现二义性
  - B. 如果一个派生类是从具有两个同名间接基类的两个直接基类派生来的，则派生类对该公共基类的访问可能出现二义性
  - C. 解决二义性最常用的方法是使用作用域运算符对成员进行限定
  - D. 派生类和它的单个基类中出现同名函数时，将可能出现二义性
- (2) 多继承派生类建立对象时，( )被最先调用。
  - A. 派生类自己的构造函数
  - B. 虚基类的构造函数
  - C. 非虚基类的构造函数
  - D. 派生类中子对象类的构造函数

# 第四章：继承与派生

- 继承与派生的基本概念
- 单继承
- 派生类的访问控制
- 多继承
- 虚基类
- 基类与派生类对象间的赋值规则 
- 本章小结

# 4.6 基类与派生类对象赋值规则

## 4.6.1 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。

**[基类对象/指针]=[派生类对象/指针];**

如下面声明的两个类:

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

根据赋值兼容规则, 有:

(1) 可以用派生类对象给基类对象赋值:

```
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引用:

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

## 4.6 基类与派生类对象赋值规则

### 4.6.1 赋值兼容规则

(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;
```



## 4.6 基类与派生类对象赋值规则

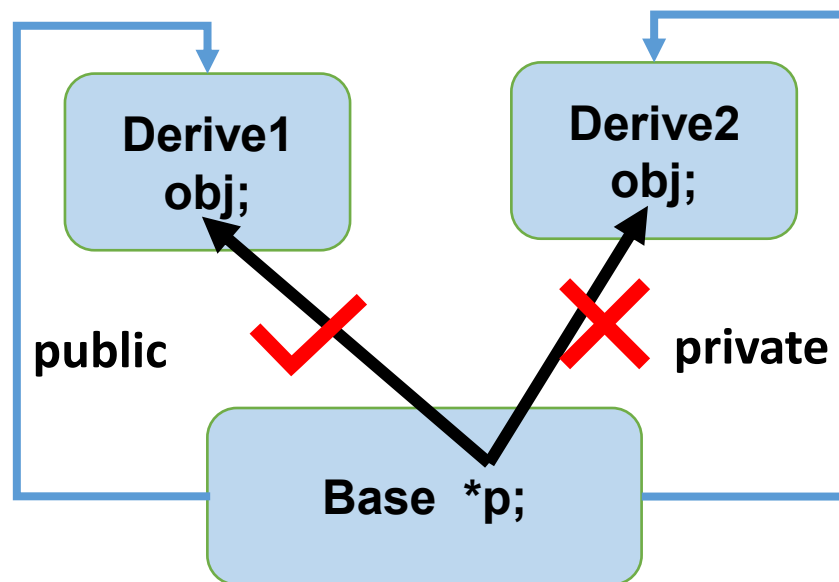
### 4.6.2 基类对象指针给派生类对象赋值规则

(1) 声明为指向基类对象的指针可以指向它的**公有派生对象**，但不允许指向**私有派生的对象**。

```
class Base { // ..... };
```

```
class Derive: private Base { // ..... };
```

```
void main() {  
    Base op1, *ptr;  
    Derive op2;  
    ptr=&op1;  
    ptr=&op2; //对吗?  
    // ... ..  
}
```



## 4.6 基类与派生类对象赋值规则

### 4.6.2 基类对象指针给派生类对象赋值规则

(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。

```
class Base { // ..... };
class Derive: public Base { // ..... };
void main() {
    Base obj1;
    Derive obj2, *ptr;
    ptr=&obj2;
    ptr=&obj1;    //对吗?
    // ... ..
}
```

□ 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;
```

## 4.6 基类与派生类对象赋值规则

### 4.6.2 基类对象指针给派生类对象赋值规则


(3) 声明为指向基类对象的指针，当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。

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

```
void main() {  
    A op1, *ptr;  
    B op2;  
    ptr=&op1;  
    ptr->print1();  
    ptr=&op2;  
    ptr->print1();  
    ptr->print2();  
}
```

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

# 第四章：继承与派生

- 继承与派生的基本概念
- 单继承
- 派生类的访问控制
- 多继承
- 虚基类
- 基类与派生类对象间的赋值规则
- 本章小结 

# 本章小结

## 重点:

- 类的继承（继承的主要作用）
- 单继承
- 派生类的访问控制
- 派生类的构造函数和析构函数
- 多继承的构造函数和析构函数

## 难点:

- 虚基类
- 基类和派生类对象的赋值规则