

# 嵌入式系统工程师



---

# 继承与派生

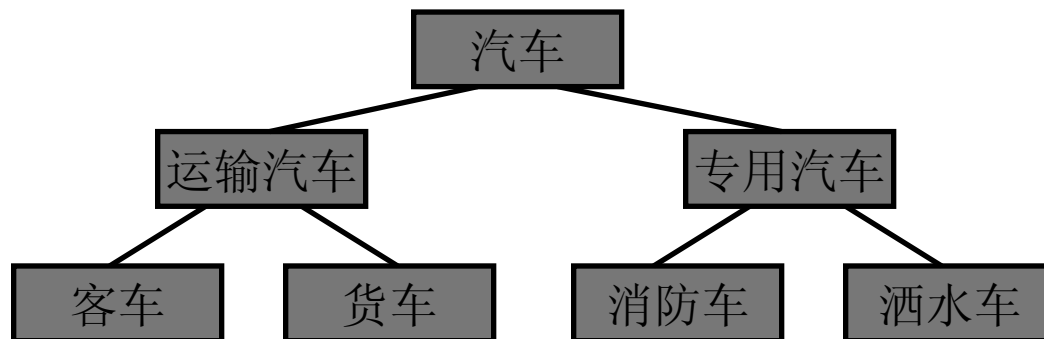
---

- 派生的概念
- 派生的方式
- 衍生类的构造函数与析构函数

- 派生的概念
- 派生的方式
- 派生类的构造与析构函数

## ➤ 为什么要使用继承

- 简单的汽车分类层次图：



- 最高层（基类）：是最普遍、最一般的
- 低层（派生类）：比它的上一层更具体，并且含有高层的特性（继承），同时也与高层有细微的不同
- 继承性是程序设计中一个非常有用的、有力的特性，它可以让程序员在既有类的基础上，通过增加少量代码或修改少量代码的方法得到新的类，从而较好地解决了代码重用的问题

例：使用继承的必要性

person(个人)类

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

employee(职工)类

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

➤ 直接定义employee类，代码重复非常严重

- 派生的概念
- 派生的方式
  - 公有方式派生 (public)
  - 私有方式派生 (private)
  - 保护方式派生 (protected)
  - 多继承
- 派生类的构造与析构函数


►使用继承：将employee定义成person类的派生类. 增加新的数据成员department和salary; 修改print成员函数

一般格式:

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

如:

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





- 在声明派生类时，根据“派生方式”所写的关键字不同可以分为三种派生方式：
  - 用 public 关键字的为公有派生方式
  - 用 private 关键字的为私有派生方式
  - 用 protected 关键字的为保护派生方式
- 从继承源上分：
  - 单继承：指每个派生类只直接继承了一个基类的特征
  - 多继承：指多个基类派生出一个派生类的继承关系，多继承的派生类直接继承了不止一个基类的特征

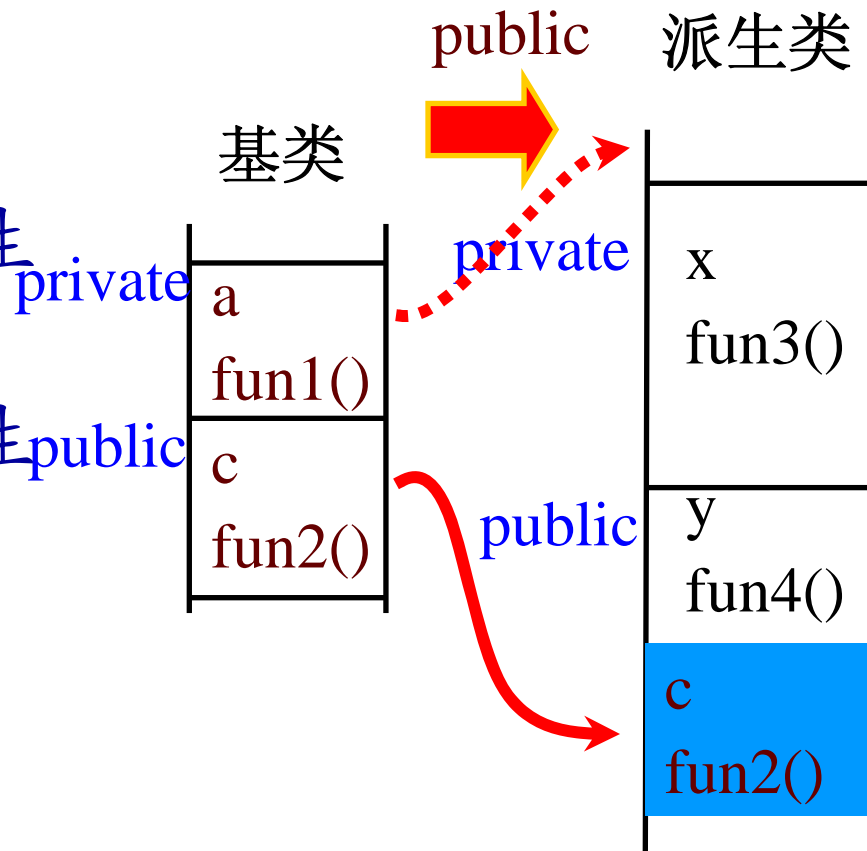
- 派生的概念
- 派生的方式
  - 公有方式派生 (public)
  - 私有方式派生 (private)
  - 保护方式派生 (protected)
  - 多继承
- 衍生类的构造与析构函数

## ➤ 公有派生

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

- 基类中的**私有成员** → 派生类中**不可访问**
- 基类中的**公有成员** → 派生类中是**公有**的

### 例4.1公有派生



➤ 派生的概念

➤ 派生的方式

➤ 公有方式派生 (public)

➤ 私有方式派生 (private)

➤ 保护方式派生 (protected)

➤ 多继承

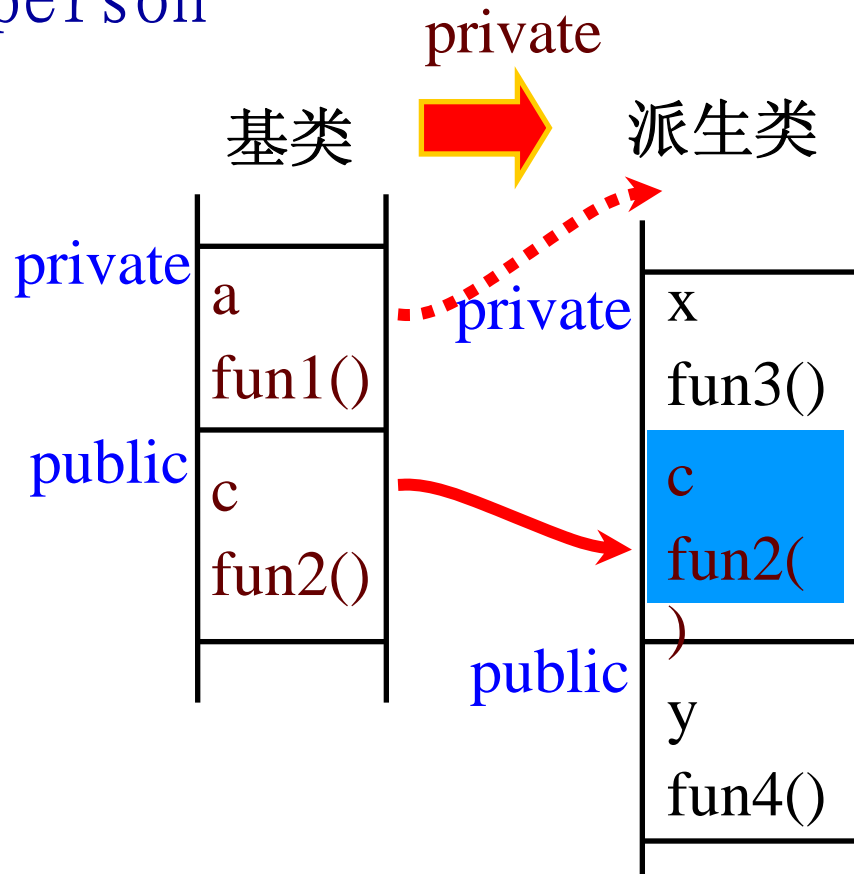
➤ 衍生类的构造与析构函数

## ➤ 私有派生

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

- 基类中的私有成员 → 派生类中不能访问
- 基类中的公有成员 → 派生类中是私有的

### 例4.2私有派生



➤ 派生的概念

➤ 派生的方式

➤ 公有方式派生 (public)

➤ 私有方式派生 (private)

➤ 保护方式派生 (protected)

➤ 多继承

➤ 派生类的构造与析构函数

# 保护成员的作用

- 私有成员在派生类中是无权直接访问的，只能通过调用基类中的公有成员函数的方式实现
- 一定要直接访问基类中的私有成员，可以把这些成员声明为保护成员 protected。一般格式：

```
class 类名 {  
    [private:]  
        私有成员  
    protected:  
        保护成员  
    public:  
        公有成员  
};
```

1、不涉及派生时，保护成员与私有成员的地位完全一致。

class samp{ 例4.3 保护成员

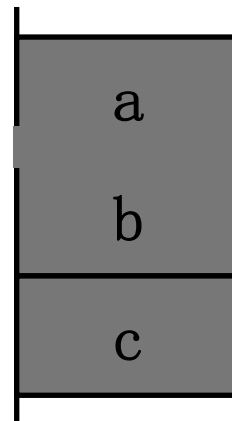
```

    int a;
protected:    int b;
public:  int c;
    samp(int n, int m) {a=n; b=m;}
    int geta() {return a;}
    int getb() {return b;}
};
    
```

**private**

**protected**

**public**



```

int main()
{ samp obj(20, 30);
  obj.a=11;           //Error, 私有成员
  obj.b=22;           //Error, 保护成员
  obj.c=33;           //Ok
  cout<<obj.geta() << ' ' <<obj.getb() <<endl;    //Ok
}
    
```



## 2、以公有派生时：基类中的保护成员 → 在派生类中仍是保护的

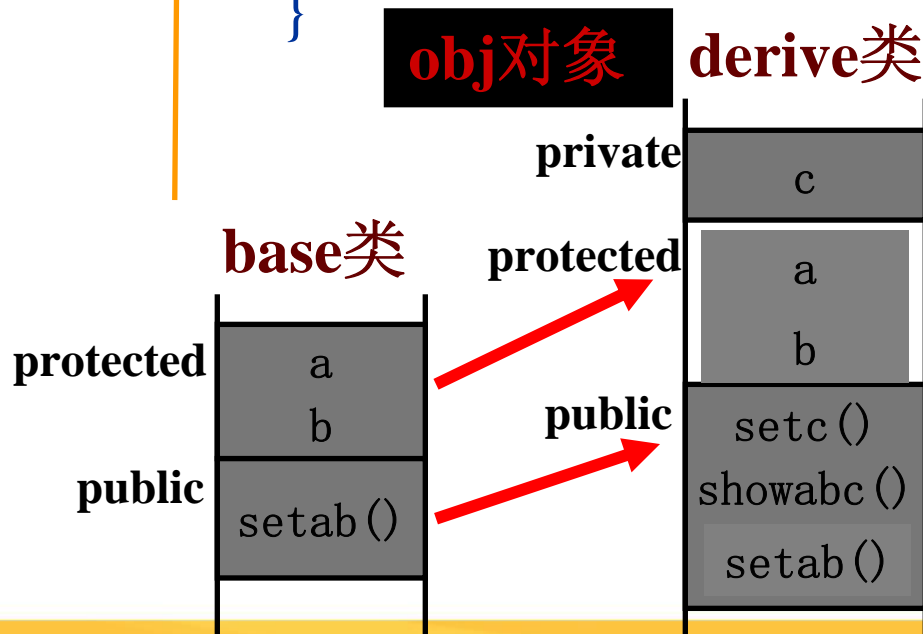
### 例4.4

```

class base{
protected:  int a,b;
public:
    void setab(int n,int m){a=n;b=m;}
};
class derive:public base{
    int c;
public:
    void setc(int n){c=n;}
    void showabc()
    { cout<<a<<endl;
      cout<<b<<endl;
      cout<<c<<endl; }
    };
    
```

```

int main()
{
    derive obj;
    obj.setab(2,4);
    obj.setc(3);
    obj.showabc();
}
    
```



### 3、私有派生时：基类中的保护成员 → 在派生类中 是私有的。

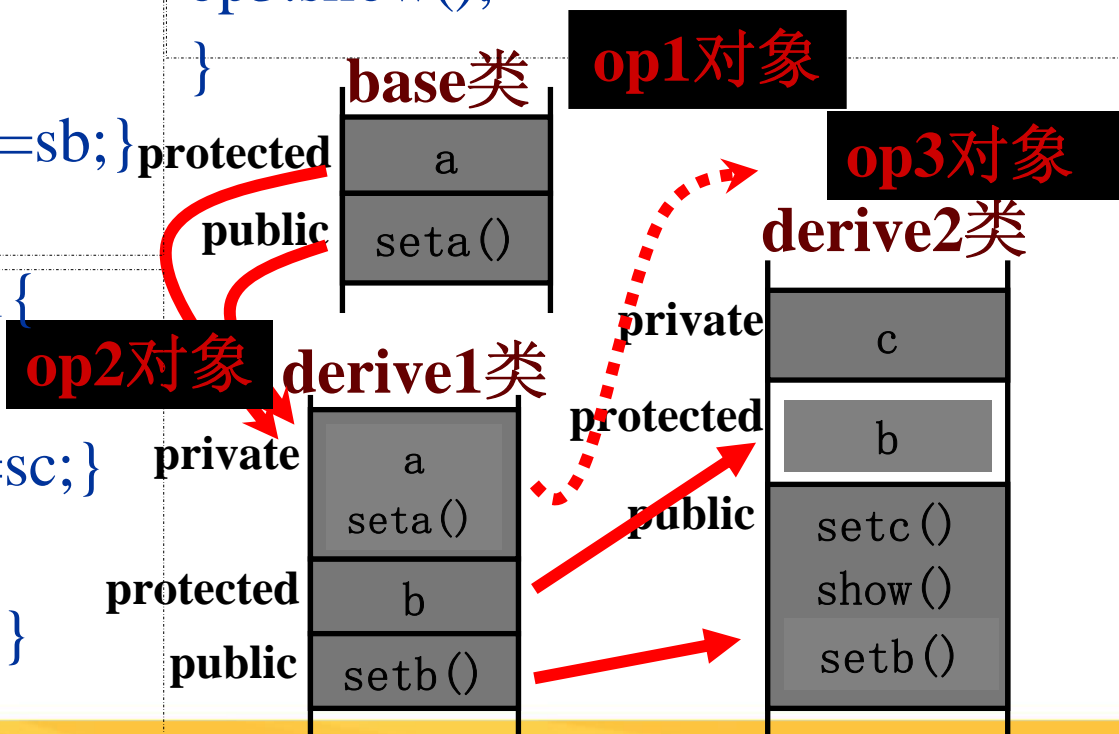
**例4.5**

```
class base{
protected: int a;
public: void seta(int sa){a=sa;}
};
```

```
class derive1:private base{
protected: int b;
public: void setb(int sb){b=sb;}
};
```

```
class derive2:public derive1{
    int c;
public: void setc(int sc){c=sc;}
    void show()
{ cout<<a<<b<<c<<endl; }
};
```

```
int main(){
base op1; op1.seta(1);
derive1 op2; op2.setb(2);
derive2 op3; op3.setc(3);
op3.show();
}
```

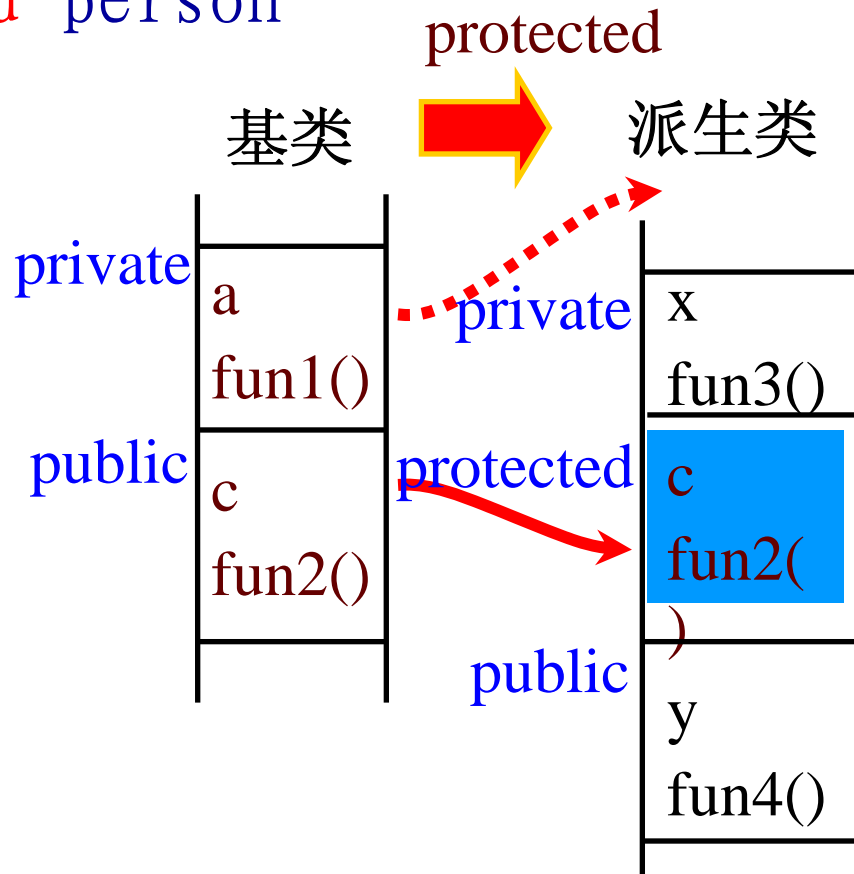


## ➤ 保护派生

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

- 基类中的私有成员 → 派生类中不能访问
- 基类中的公有成员 → 派生类中是保护的

例：4\_6保护派生



# 派生后访问权限总结

公有派生		私有派生		保护派生	
基类属性	派生类权限	基类属性	派生类权限	基类属性	派生类权限
私有	不能访问	私有	不能访问	私有	不能访问
保护	保护	保护	私有	保护	保护
公有	公有	公有	私有	公有	保护

## ➤ 派生的概念

## ➤ 派生的方式

- 公有方式派生 (public)

- 私有方式派生 (private)

- 保护方式派生 (protected)

- 多继承

## ➤ 派生类的构造与析构函数

- 多继承可以看作是单继承的扩展。所谓多继承是指派生类具有多个基类，派生类与每个基类之间的关系仍可看作是一个单继承。
- 多继承下派生类的定义格式如下：  
class <派生类名>: <继承方式1><基类名1>, <继承方式2><基类名2>, ...  
{  
    <派生类类体>  
};  
其中，<继承方式1>, <继承方式2>, ...是三种继承方式：public、private、protected之一

## ➤ 例4\_7\_double\_subclass.cpp

### ➤ 注意:

➤ 当使用多继承时要注意避免发生二义性，如下:

```
class A{  
public:  
    void print();  
    //.....  
};
```

```
class B{  
public:  
    void print();  
    //.....  
};
```

```
class C:public A,public B  
{  
    .....  
};  
在定义如下时:  
C c1;  
c1.print(); //error
```

- 解决方法是: `c1. A::print (); //调用A的`  
或者 `c1. B::print (); //调用B的`
- 在C公有继承A和B, 当A和B都公有继承D时, 这使得A和B都具有D的公有部分, 在C中使用D的公有部分时, 编译器就会不确定是调用A的还是调用B的而出现错误



- 派生的概念
- 派生的方式
- 派生类的构造与析构函数

- 基类都有构造函数和析构函数，或是显式定义、或是隐式定义
- 在派生类中：
  - 创建派生类对象时，如何调用基类的构造函数对基类数据初始化？
  - 撤消派生类对象时，如何调用基类的析构函数对基类对象的数据成员进行善后处理？

# 1 派生类构造函数和析构函数的 执行顺序

➤ 派生类构造函数的执行顺序:

盖房子



基类的构造函数



派生类的构造函数

➤ 派生类析构函数的执行顺序:

拆房子

基类的析构函数



派生类的析构函数



例4.8

## 2 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(或全部默认值), 或没有显示定义构造函数时, 那么派生类可以不向基类传递参数, 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
- 派生类构造函数的一般格式:

派生类构造函数名(参数表0): 基类构造函数名(参数表1)

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



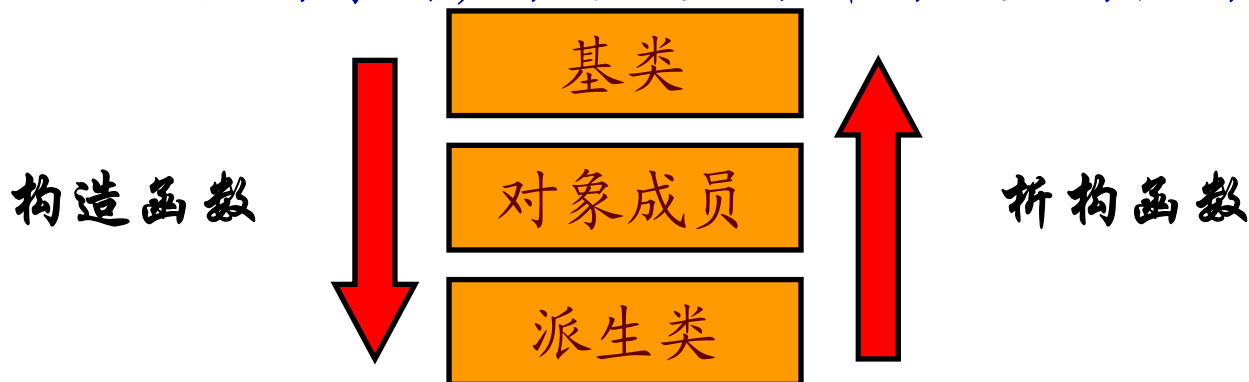
参数表1是参数表0的子集

- 当派生类中含有对象成员时，其构造函数的一般形式为：

派生类构造函数名(参数表0): 基类构造函数名(参数表1),  
对象成员名1(参数表2), ...,  
对象成员名n(参数表n+1)

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

在定义派生类对象时，构造函数与析构函数的执行顺序为：



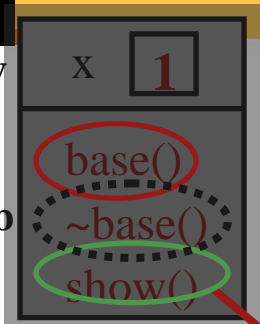
## base类

```
class base{
    int x;
public:
    base(int i)
    { x=i;
      cout<<"构造base类, x="
        <<x<<endl; }
    ~base()
    { cout<<"析构base类, x="
      <<x<<endl;}
    void show()
    { cout<<"x="<<x<<endl;}
};

class derive:public base{
    base d;
    int y;
public:
    derive(int i,int j,int k):base(i), d(j)
```

prv

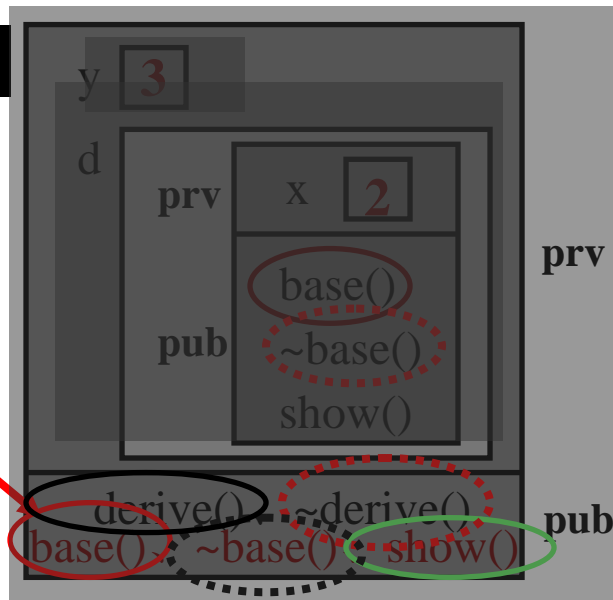
pub



```
{ y=k;
  cout<<"构造derived类, y="<<y<<endl;}
~derive()
{ cout<<"析构derived类, y="<<y<<endl;}
};

int main()
{ derive obj(1,2,3);
  obj.show();
}
```

## derive类



输出结果:

构造base类, x=1  
构造base类, x=2  
构造derived类, y=3  
x=1  
析构derived类, y=3  
析构base类, x=2  
析构base类, x=1

例4.9

## ➤ 说明:

- 如果派生类的基类也是一个派生类，则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，在派生类中是否要定义析构函数与它所属的基类无关，故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，它们各自是独立的

➤ 本章主要讲了派生类的三种派生方式:

➤ 公有派生 (public)

➤ 私有派生 (private)

➤ 保护派生 (protected)

在这三种派生方式中比较常见得是公有派生 (public) 方式, 需要大家牢记的是根据派生方式的不同派生类对基类的成员有不同访问权限

➤ 派生类中构造函数与析构函数定义方式

练习





值得信赖的教育品牌

Tel: 400-705-9680 , Email: edu@sunplusapp.com , BBS: bbs.sunplusedu.com

