程序设计与算法（三）

C++面向对象程序设计

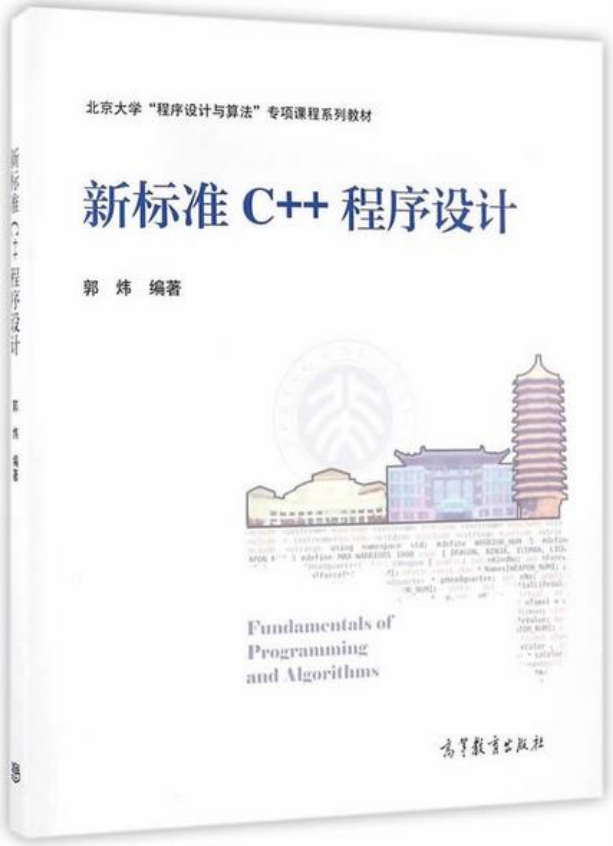
郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著



继承和派生

德国菲森新天鹅堡

继承和派生的概念

- 继承：在定义一个新的类B时，如果该类与某个已有的类A相似(指的是B拥有A的全部特点)，那么就可以把A作为一个**基类**，而把B作为基类的一个**派生类(也称子类)**。

继承和派生的概念

- 派生类是通过对基类进行修改和扩充得到的。在派生类中，可以扩充新的成员变量和成员函数。
- 派生类一经定义后，可以独立使用，不依赖于基类。

继承和派生的概念

- 派生类拥有基类的全部成员函数和成员变量，不论是private、protected、public 。
  - 在派生类的各个成员函数中，不能访问基类中的private成员。

需要继承机制的例子

- 所有的学生都有的共同属性：
  - 姓名
  - 学号
  - 性别
  - 成绩

所有的学生都有的共同方法（成员函数）：

- 是否该留级
- 是否该奖励

需要继承机制的例子

而不同的学生，又有各自不同的属性和方法

研究生

导师

系

大学生

系

中学生

竞赛特长加分

需要继承机制的例子

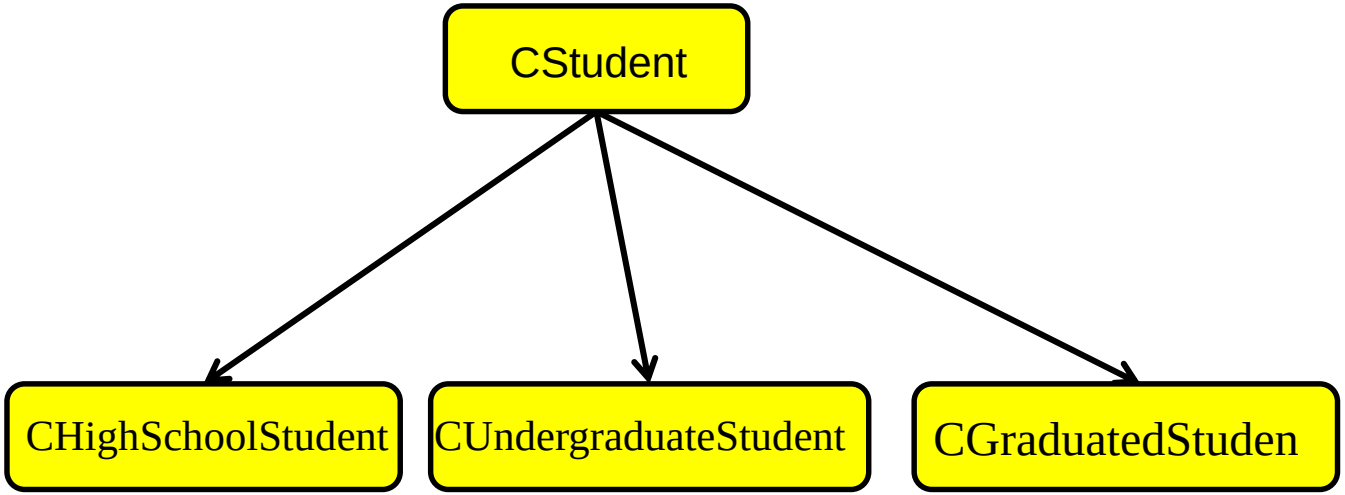
- 如果为每类学生都从头编写一个类，显然会有不少重复的代码，浪费。

需要继承机制的例子

- 如果为每类学生都从头编写一个类，显然会有不少重复的代码，浪费。

- 比较好的做法是编写一个“学生”类，概括了各种学生的共同特点，然后从“学生”类派生出“大学生”类，“中学生”类，“研究生类”。

需要继承机制的例子


CStudent
CHighSchoolStudent
CUndergraduateStudent
CGraduatedStuden

派生类的写法

```
class 派生类名：public 基类名
{

};
```

```
class CStudent {
        private:
                string sName;
                int nAge;
        public:
        bool IsThreeGood() { };
        void SetName( const string & name )
        { sName = name; }
        //.....
};
class CUndergraduateStudent: public CStudent {
        private:
                int nDepartment;
        public:
                bool IsThreeGood() {  .....  };  //覆盖
                bool CanBaoYan() { .... };
}; // 派生类的写法是：  类名: public 基类名
```

```
class CGraduatedStudent:public CStudent {
        private:
                int nDepartment;
                char szMentorName[20];
        public:
                int CountSalary() { ... };
};
```

派生类对象的内存空间

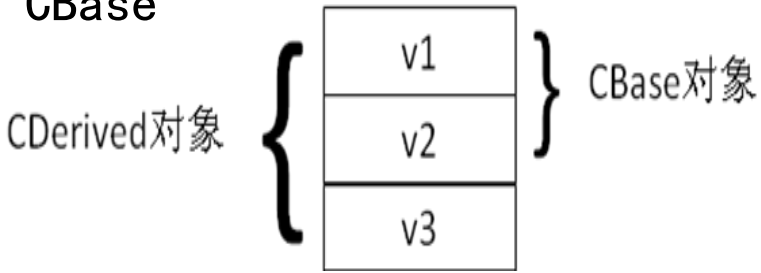
派生类对象的体积，等于基类对象的体积，再加上派生类对象自己的成员变量的体积。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象新增的成员变量之前。

```
class CBase
{
        int v1,v2;
};
class CDerived:public CBase
{
        int v3;
};
```

派生类对象的内存空间

派生类对象的体积，等于基类对象的体积，再加上派生类对象自己的成员变量的体积。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象新增的成员变量之前。

```
class CBase
{
      int v1,v2;
};
class CDerived:public CBase
{
      int v3;
};
```



如果多种事物，有一些共同的特点，又有一些各自不同的特点，如何编写类来代表这些事物比较合适？

A) 为每种事物独立编写一个类，各类之间互相无关

B) 写一个类代表其中一种事物，代表其他事物的类，都从这个类派生出来

C) 概括所有事物的共同特点，写一个基类。然后为每种事物写一个类，都从基类派生而来

D) 一共就写一个类，包含所有事物的所有特点，然后用一个成员变量作为标记来区分不同种类的事物

如果多种事物，有一些共同的特点，又有一些各自不同的特点，如何编写类来代表这些事物比较合适？

A) 为每种事物独立编写一个类，各类之间互相无关

B) 写一个类代表其中一种事物，代表其他事物的类，都从这个类派生出来

C) 概括所有事物的共同特点，写一个基类。然后为每种事物写一个类，都从基类派生而来

D) 一共就写一个类，包含所有事物的所有特点，然后用一个成员变量作为标记来区分不同种类的事物

答案：C

继承实例程序:学籍管理 (P228)

```
#include <iostream>
#include <string>
using namespace std;
class CStudent  {
    private:
        string name;
        string id;  //学号
        char gender; //性别,'F'代表女，'M'代表男
        int age;
    public:
        void PrintInfo();
        void SetInfo( const string & name_,const string & id_,
              int age_,       char gender_ );
        string GetName() {  return name;  }
};
```

```
class CUndergraduateStudent:public CStudent
{//本科生类，继承了CStudent类
    private:
        string department; //学生所属的系的名称
    public:
        void QualifiedForBaoyan() {   //给予保研资格
            cout << “qualified for baoyan” << endl;
        }
        void PrintInfo() {
            CStudent::PrintInfo(); //调用基类的PrintInfo
            cout << “Department:” << department <<endl;
        }
        void SetInfo( const string & name_,const string & id_,
            int age_,char gender_ ,const string &  department_) {
            CStudent::SetInfo(name_,id_,age_,gender_); //调用基类的SetInfo
            department = department_;
        }
};
```

```
void CStudent::PrintInfo()
{
        cout << "Name:" << name << endl;
        cout << "ID:" << id << endl;
        cout << "Age:" << age << endl;
        cout << "Gender:" << gender << endl;
}
void CStudent::SetInfo( const string & name_,const string & id_,
                        int age_,char gender_ )
{
        name = name_;
        id = id_;
        age = age_;
        gender = gender_;
}
```

```
int main()
{

    CUndergraduateStudent s2;
     s2.SetInfo(“Harry Potter ”,  “118829212”,19,‘M’,“Computer Science”);
     cout <<  s2.GetName() << “ ” ;
     s2.QualifiedForBaoyan ();
     s2.PrintInfo ();
     return 0;
}
```

输出结果：

Harry Potter qualified for baoyan
Name:Harry Potter
ID:118829212
Age:19
Gender:M
Department:Computer Science

继承关系
和
复合关系

美国纪念碑谷

类之间的两种关系

- 继承：“是”关系。
  - 基类A，B是基类A的派生类。
  - 逻辑上要求：“一个B对象也是一个A对象”。

类之间的两种关系

- 继承：“是”关系。
  - 基类A，B是基类A的派生类。
  - 逻辑上要求：“一个B对象也是一个A对象”。

- 复合：“有”关系。
  - 类C中“有”成员变量k，k是类D的对象，则C和D是复合关系
  - 一般逻辑上要求：“D对象是C对象的固有属性或组成部分”。

继承关系的使用

- 写了一个 CMan 类代表男人
- 后来又发现需要一个CWoman类来代表女人

继承关系的使用

- 写了一个 CMan 类代表男人
- 后来又发现需要一个CWoman类来代表女人

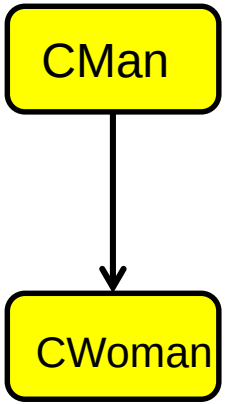
- CWoman类和CMan类有共同之处

继承关系的使用

- 写了一个 CMan 类代表男人
- 后来又发现需要一个CWoman类来代表女人

- CWoman类和CMan类有共同之处

- 就让CWoman类从CMan类派生而来，是否合适？



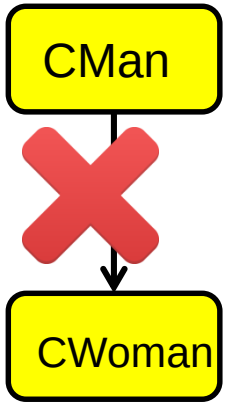
继承关系的使用

- 写了一个 CMan 类代表男人
- 后来又发现需要一个CWoman类来代表女人

- CWoman类和CMan类有共同之处

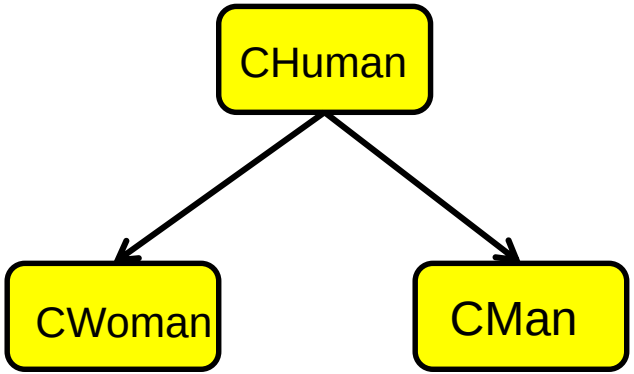
- 就让CWoman类从CMan类派生而来，是否合适？

- 是**不合理的**！因为“一个女人也是一个男人”从逻辑上不成立！



继承关系的使用

好的做法是概括男人和女人共同特点，
写一个 CHuman类，代表“人”，然后CMan和CWoman都从CHuman派生。



复合关系的使用

- 几何形体程序中，需要写“点”类，也需要写“圆”类

复合关系的使用

- 几何形体程序中，需要写“点”类，也需要写“圆”类，

```
class CPoint
{
    double x,y;
};
```

```
class CCircle:public CPoint
{
    double r;
};
```

复合关系的使用

- 几何形体程序中，需要写“点”类，也需要写“圆”类

```
class CPoint
{
    double x,y;
};
```

```
class CCircle:public CPoint
{
    double r;
};
```

复合关系的使用

- 几何形体程序中，需要写“点”类，也需要写“圆”类，两者的关系就是复合关系 ---- 每一个“圆”对象里都包含(有)一个“点”对象，这个“点”对象就是圆心

```
class CPoint
{
    double x,y;
};
```

```
class CCircle
{
    double r;
    CPoint center;
};
```

复合关系的使用

- 几何形体程序中，需要写“点”类，也需要写“圆”类，两者的关系就是复合关系 ---- 每一个“圆”对象里都包含(有)一个“点”对象，这个“点”对象就是圆心

```
class CPoint
{
    double x,y;
    friend class CCircle;
   //便于Ccirle类操作其圆心
};
```

```
class CCircle
{
    double r;
    CPoint center;
};
```

以下哪种派生关系是合理的

A) 从“虫子”类派生出“飞虫”类

B) 从“点”类派生出“圆”类

C) 从“狼”类派生出“狗”类

D) 从“爬行动物”类派生出“哺乳动物”类

以下哪种派生关系是合理的

A) 从“虫子”类派生出“飞虫”类

B) 从“点”类派生出“圆”类

C) 从“狼”类派生出“狗”类

D) 从“爬行动物”类派生出“哺乳动物”类

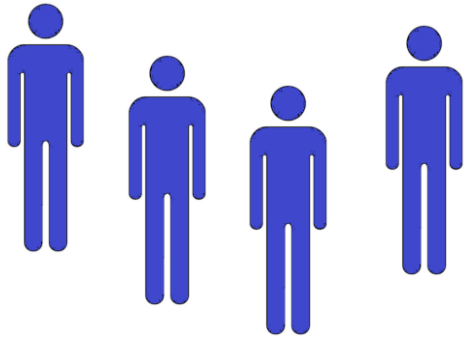
答案：A

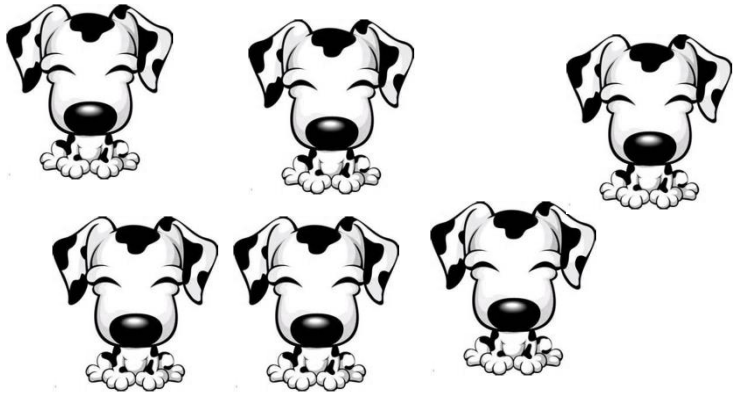
复合关系的使用

- 如果要写一个小区养狗管理程序，

  需要写一个“业主”类，还需要写一个“狗”类。

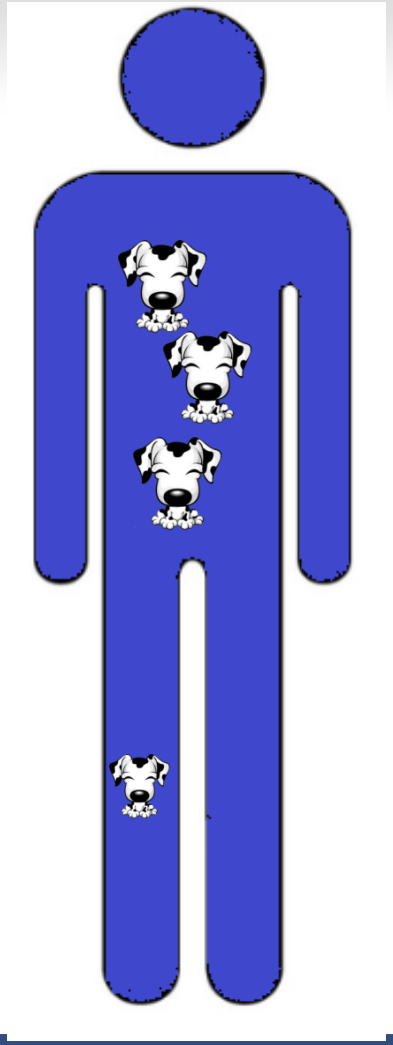
- 而狗是有 “主人” 的，主人当然是业主(假定狗只有一个主人，但一个业主可以有最多10条狗)

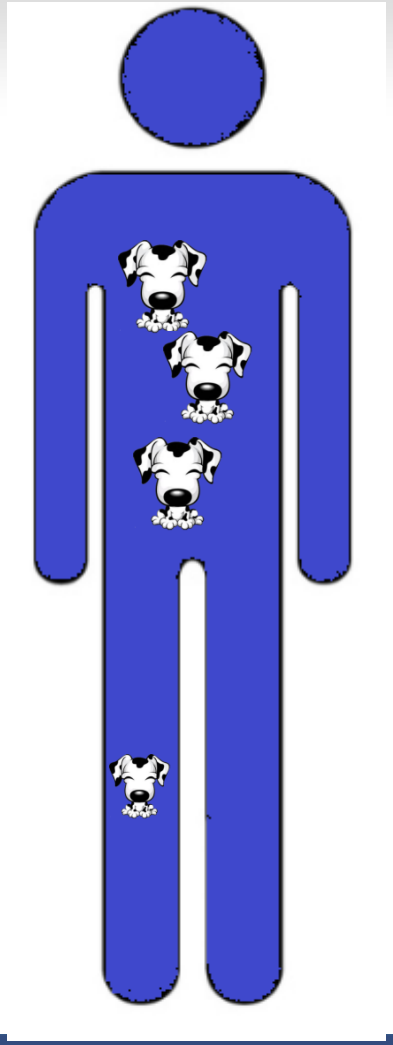


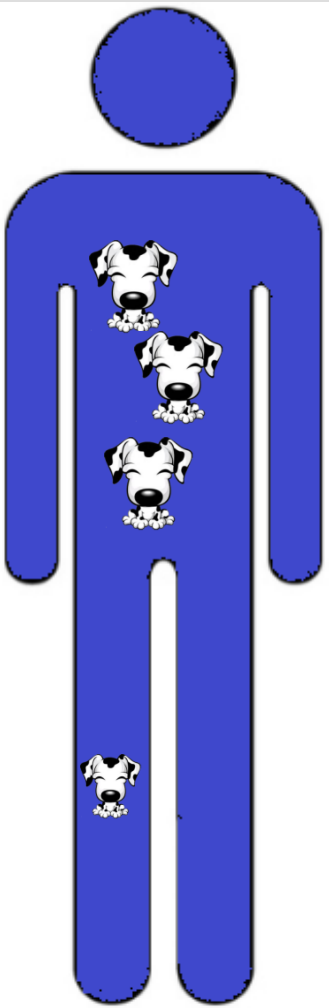


复合关系的使用

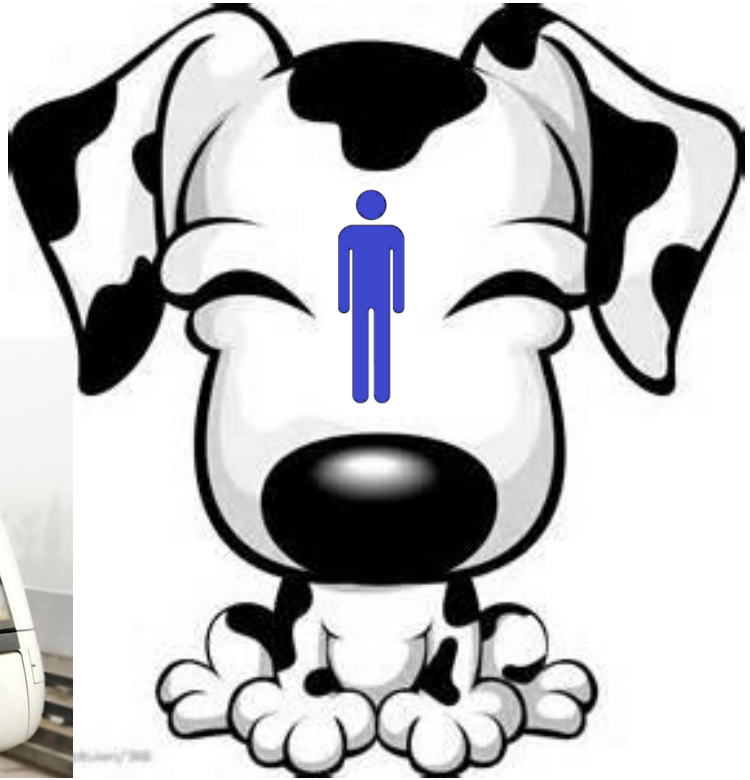
```
class CDog;
class CMaster
{
    CDog dogs[10];
};
class CDog
{
    CMaster m;
};
```

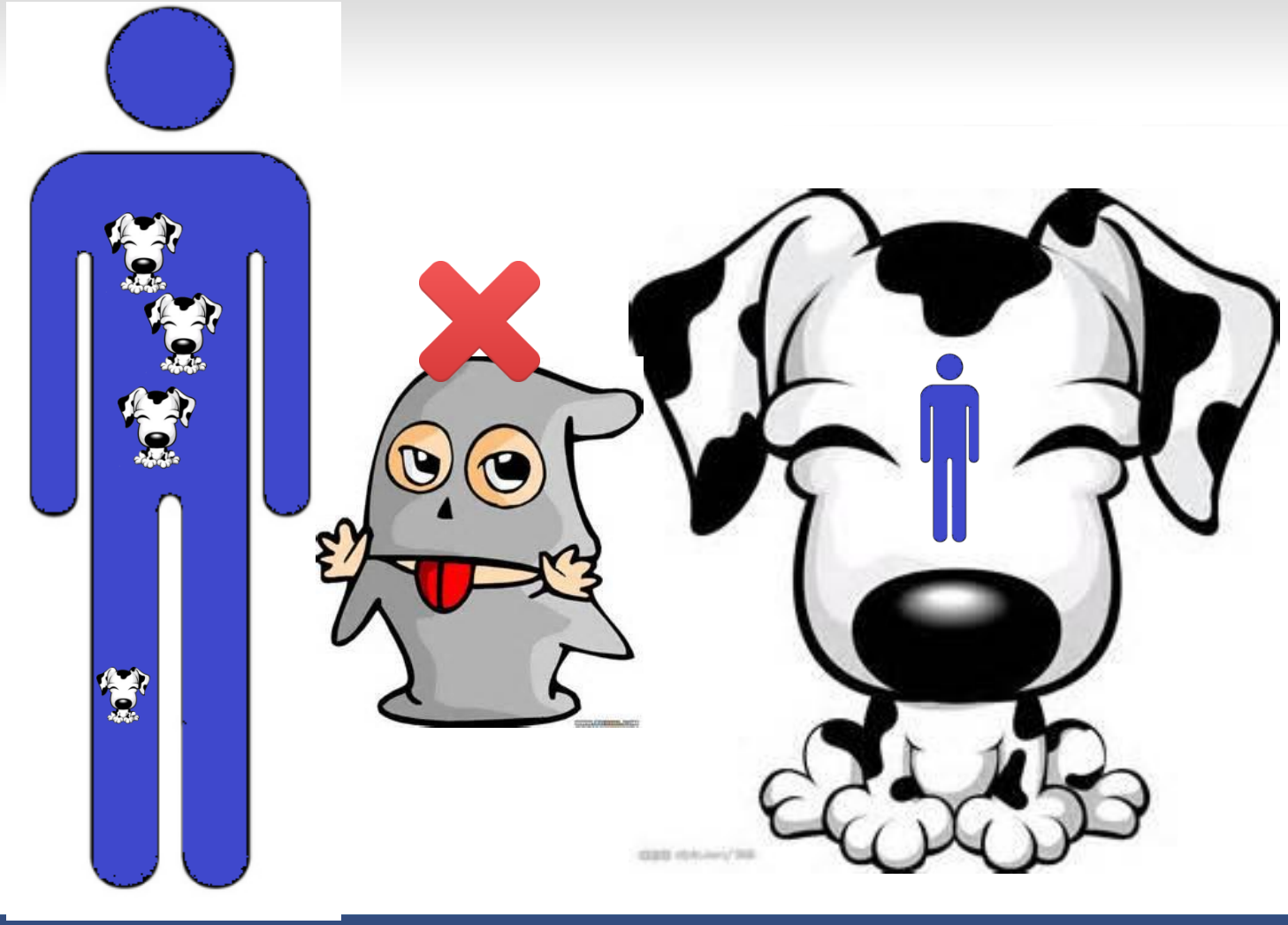







和谐号





复合关系的使用

```
class CDog;
class CMaster
{
    CDog dogs[10];
};
class CDog
{
    CMaster m;
};
```

循环定义是不对的！编译不通过

复合关系的使用

- 另一种写法：

  为“狗”类设一个“业主”类的成员对象；

  为“业主”类设一个“狗”类的对象指针数组。

```
class CDog;
class CMaster  {
    CDog * dogs[10];
};
class CDog   {
    CMaster m;
};
```

复合关系的使用

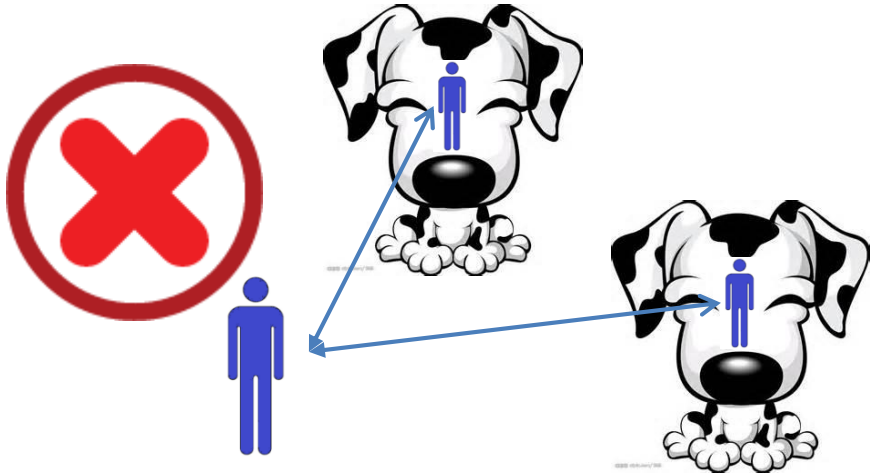
- 另一种写法：

  为“狗”类设一个“业主”类的成员对象；

  为“业主”类设一个“狗”类的对象指针数组。

```
class CDog;
class CMaster  {
    CDog * dogs[10];
};
class CDog   {
    CMaster m;
};
```

狗可以是共同的主人，如何维护不同的狗的相同主人的信息？？？



复合关系的使用

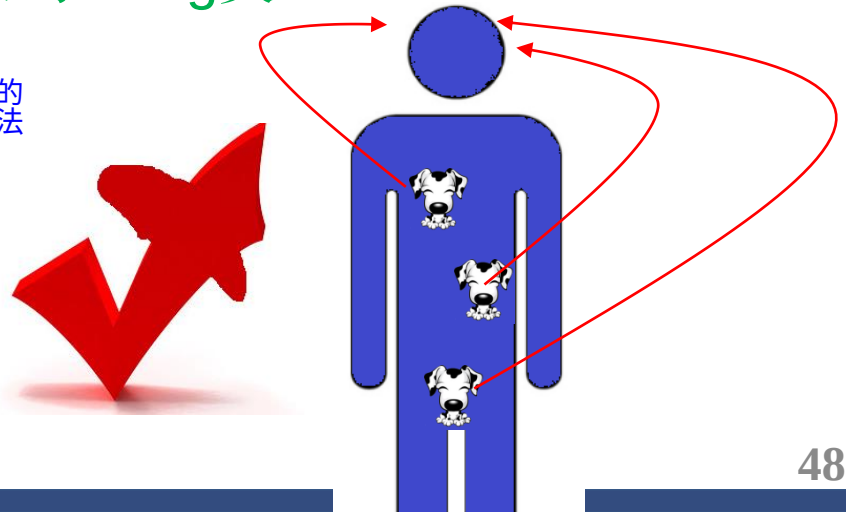
- 凑合的写法：

  为“狗”类设一个“业主”类的对象指针；

  为“业主”类设一个“狗”类的对象数组。

```
class CMaster;   //CMaster必须提前声明，不能先
                 //写CMaster类后写Cdog类
class CDog  {
    CMaster * pm;
};
class CMaster {
    CDog dogs[10];
};
```

狗不是主人的固有属性，有点别扭
这里面狗狗失去了自由，所有的狗的对象都包含在主人的对象里面，没法独立操作。



复合关系的使用

- 正确的写法：

  为“狗”类设一个“业主”类的对象指针；

  为“业主”类设一个“狗”类的对象指针数组。

```
class CMaster;    //CMaster必须提前声明，不能先
                          //写CMaster类后写Cdog类
class CDog  {
    CMaster * pm;
};
class CMaster  {
    CDog * dogs[10];
};
```

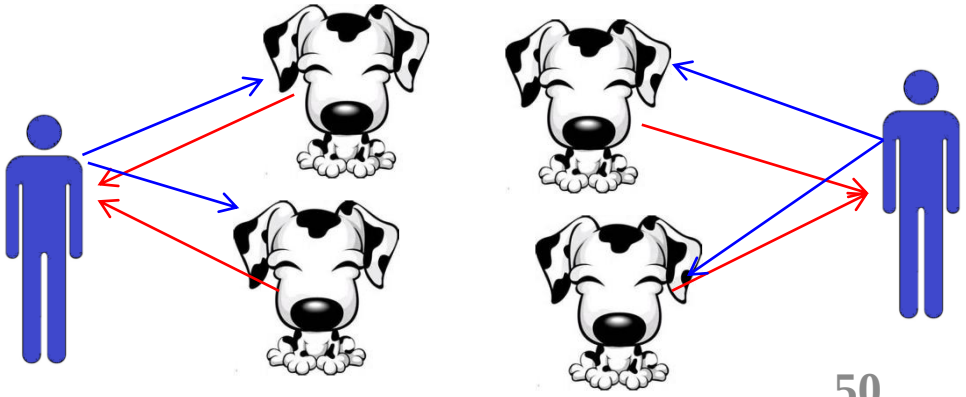
复合关系的使用

- 正确的写法：

  为“狗”类设一个“业主”类的对象指针；

  为“业主”类设一个“狗”类的对象指针数组。

```
class CMaster;    //CMaster必须提前声明，不能先
                  //写CMaster类后写Cdog类
class CDog  {
    CMaster * pm;
};
class CMaster  {
    CDog * dogs[10];
};
```



派生类覆盖基类成员

冰岛杰古沙龙冰河湖

覆盖

- 派生类可以定义一个和基类成员同名的成员，这叫覆盖。在派生类中访问这类成员时，缺省的情况是访问派生类中定义的成员。要在派生类中访问由基类定义的同名成员时，要使用作用域符号::。

基类和派生类有同名成员的情况：

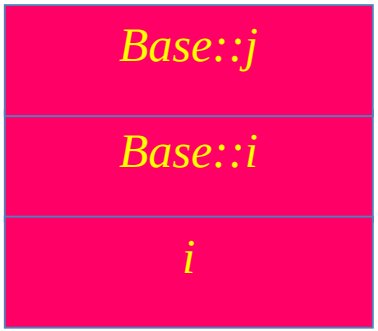
```
class base {
        int j;
public:
        int i;
        void func();
};
```

```
class derived :public base{
public:
        int i;
        void access();
        void func();
};
```

```
void derived::access()  {
        j = 5; //error
        i = 5;    //引用的是派生类的 i
        base::i = 5; //引用的是基类的 i
        func(); //派生类的
        base::func(); //基类的
}
derived  obj;
obj.i = 1;
obj.base::i = 1;
```

➢一般来说，基类和派生类不定义同名成员变量。

Obj 占用的存储空间



派生类和基类有同名同参数表的成员函数，这种现象：

A) 叫重复定义，是不允许的

B) 叫函数的重载

C) 叫覆盖。在派生类中基类的同名函数就没用了

D) 叫覆盖。体现了派生类对从基类继承得到的特点的修改

派生类和基类有同名同参数表的成员函数，这种现象：

A) 叫重复定义，是不允许的

B) 叫函数的重载

C) 叫覆盖。在派生类中基类的同名函数就没用了

D) 叫覆盖。体现了派生类对从基类继承得到的特点的修改

答案：D

以下说法正确的是：

A) 派生类可以和基类有同名成员函数，但是不能有同名成员变量

B) 派生类的成员函数中，可以调用基类的同名同参数表的成员函数

C) 派生类和基类的同名成员函数必须参数表不同，否则就是重复定义

D) 派生类和基类的同名成员变量存放在相同的存储空间

以下说法正确的是：

A) 派生类可以和基类有同名成员函数，但是不能有同名成员变量

B) 派生类的成员函数中，可以调用基类的同名同参数表的成员函数

C) 派生类和基类的同名成员函数必须参数表不同，否则就是重复定义

D) 派生类和基类的同名成员变量存放在相同的存储空间

答案：B

类的保护成员

冰岛维克镇海角

另一种存取权限说明符：protected

- 基类的private成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友员函数
- 基类的public成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友员函数
  - 派生类的成员函数
  - 派生类的友员函数
  - 其他的函数
- 基类的protected成员：可以被下列函数访问
  - 基类的成员函数
  - 基类的友员函数
  - 派生类的成员函数可以访问当前对象的基类的保护成员

保护成员(P233)

```
class  Father {
    private: int nPrivate;    //私有成员
       public:  int nPublic;     //公有成员
       protected: int nProtected;    // 保护成员
};
class Son :public Father{
    void AccessFather () {
        nPublic = 1; // ok;
        nPrivate = 1;  // wrong
        nProtected = 1; // OK, 访问从基类继承的protected成员
        Son f;
              f.nProtected = 1; //wrong , f不是当前对象
    }
};
```

```
int main()
{
	Father f;
	Son s;
		f.nPublic = 1;    // Ok
		s.nPublic = 1;   // Ok
		f.nProtected = 1;   // error
		f.nPrivate = 1;      // error
	s.nProtected = 1;   //error
		s.nPrivate = 1;    // error
	return 0;
}
```

派生类的构造函数

木兰围场泰丰湖

派生类的构造函数

```
class Bug  {
        private :
                int nLegs;      int nColor;
        public:
                int nType;
                Bug (  int legs, int color);
                void PrintBug (){ };
};
class FlyBug: public Bug // FlyBug是Bug的派生类
{
                int nWings;
        public:
                FlyBug( int legs,int color, int wings);
};
```

```
Bug::Bug( int legs, int color)
{
        nLegs = legs;
        nColor = color;
}
//错误的FlyBug构造函数
FlyBug::FlyBug ( int legs,int color, int wings)
{
        nLegs = legs;       // 不能访问
        nColor = color;   // 不能访问
              nType = 1; // ok
        nWings = wings;
}
//正确的FlyBug构造函数:
FlyBug::FlyBug ( int legs, int color, int wings):Bug( legs, color)
{
        nWings = wings;
}
```

```
int main() {
    FlyBug fb ( 2,3,4);
    fb.PrintBug();
    fb.nType = 1;
    fb.nLegs = 2 ;  // error.  nLegs is private
    return 0;
}
```

FlyBug fb (2,3,4);

- 在创建派生类的对象时，需要调用基类的构造函数：初始化派生类对象中从基类继承的成员。在执行一个派生类的构造函数之前，总是先执行基类的构造函数。
- 调用基类构造函数的两种方式
  - 显式方式：在派生类的构造函数中，为基类的构造函数提供参数.

    derived::derived(arg_derived-list):base(arg_base-list)
  - 隐式方式：在派生类的构造函数中，省略基类构造函数时，派生类的构造函数则自动调用基类的默认构造函数.
- 派生类的析构函数被执行时，执行完派生类的析构函数后，自动调用基类的析构函数。

```
class Base {
        public:
                int n;
                Base(int i):n(i)
                { cout << "Base " << n << " constructed" << endl;}
                ~Base()
                { cout << "Base " << n << " destructed" << endl; }
};
class Derived:public Base  {
        public:
                Derived(int i):Base(i)
                {  cout << "Derived constructed" << endl;  }
                ~Derived()
                {  cout << "Derived destructed" << endl;}

};
int main()  {  Derived Obj(3);  return 0;   }
```

输出结果:

```
Base 3 constructed
Derived constructed
Derived destructed
Base 3 destructed
```

包含成员对象的派生类的构造函数写法

```
class Bug  {
        private :
                int nLegs;     int nColor;
        public:
                int nType;
                Bug (  int legs, int color);
                void PrintBug (){ };
};
```

包含成员对象的派生类的构造函数写法

```
class Skill  {
   public:
              Skill(int n) { }
};
class FlyBug: public Bug   {
            int nWings;
            Skill  sk1, sk2;
   public:
            FlyBug( int legs, int color, int wings);
};
FlyBug::FlyBug( int legs, int color, int wings):
            Bug(legs,color),sk1(5),sk2(color) ,nWings(wings) {
}
```

封闭派生类对象的构造函数执行顺序

在创建派生类的对象时：

封闭派生类对象的构造函数执行顺序

在创建派生类的对象时：

1）先执行基类的构造函数，用以初始化派生类对象中从基类继承的成员；

封闭派生类对象的构造函数执行顺序

在创建派生类的对象时：

1）先执行基类的构造函数，用以初始化派生类对象中从基类继承的成员；
2）再执行成员对象类的构造函数，用以初始化派生类对象中成员对象。

封闭派生类对象的构造函数的执行顺序

在创建派生类的对象时：

1）先执行基类的构造函数，用以初始化派生类对象中从基类继承的成员；
2）再执行成员对象类的构造函数，用以初始化派生类对象中成员对象。
3）最后执行派生类自己的构造函数

封闭派生类对象消亡时析构函数的执行顺序

在派生类对象消亡时：

封闭派生类对象消亡时析构函数的执行顺序

在派生类对象消亡时：

1）先执行派生类自己的析构函数

封闭派生类对象消亡时析构函数的执行顺序

在派生类对象消亡时：

1）先执行派生类自己的析构函数
2）再依次执行各成员对象类的析构函数

封闭派生类对象消亡时析构函数的执行顺序

在派生类对象消亡时：

1）先执行派生类自己的析构函数
2）再依次执行各成员对象类的析构函数
3）最后执行基类的析构函数

析构函数的调用顺序与构造函数的调用顺序相反。

public继承的赋值兼容规则

美国鹅颈湾

public继承的赋值兼容规则

```
class base {     };
class derived : public base {    };
base b;
derived d;
```

public继承的赋值兼容规则

```
class base {     };
class derived : public base {    };
base b;
derived d;
```

1） 派生类的对象可以赋值给基类对象

```
b = d;
```

public继承的赋值兼容规则

```
class base {     };
class derived : public base {    };
base b;
derived d;
```

1） 派生类的对象可以赋值给基类对象

```
b = d;
```

2） 派生类对象可以初始化基类引用

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

public继承的赋值兼容规则

```
class base {     };
class derived : public base {    };
base b;
derived d;
```

1） 派生类的对象可以赋值给基类对象

```
b = d;
```

2） 派生类对象可以初始化基类引用

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

3） 派生类对象的地址可以赋值给基类指针

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

public继承的赋值兼容规则

```
class base {     };
class derived : public base {    };
base b;
derived d;
```

1） 派生类的对象可以赋值给基类对象

```
b = d;
```

2） 派生类对象可以初始化基类引用

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

3） 派生类对象的地址可以赋值给基类指针

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

- 如果派生方式是 private或protected，则上述三条不可行。

protected继承和private继承 （P240）

```
class base {
};
class derived : protected base {
};
base b;
derived d;
```

- protected继承时，基类的public成员和protected成员成为派生类的protected成员。
- private继承时，基类的public成员成为派生类的private成员，基类的protected成员成为派生类的不可访问成员。
- protected和private继承不是“是”的关系。

基类与派生类的指针强制转换（P238）

- 公有派生的情况下,派生类对象的指针可以直接赋值给基类指针

  Base * ptrBase = &objDerived;

  ptrBase指向的是一个Derived类的对象；

  *ptrBase可以看作一个Base类的对象，访问它的public成员直接通过ptrBase即可，但不能通过ptrBase访问objDerived对象中属于Derived类而不属于Base类的成员

- 即便基类指针指向的是一个派生类的对象，也不能通过基类指针访问基类没有，而派生类中有的成员。
- 通过强制指针类型转换，可以把ptrBase转换成Derived类的指针

  Base * ptrBase = &objDerived;

  Derived *ptrDerived = (Derived *) ptrBase;

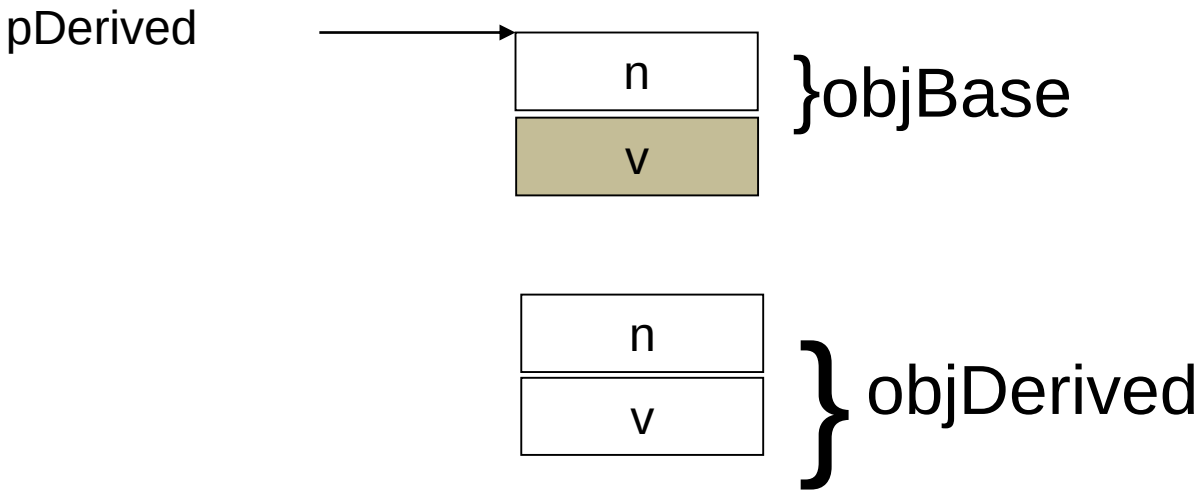
  程序员要保证ptrBase指向的是一个Derived类的对象，否则很容易会出错。

```
#include <iostream>
using namespace std;
class Base {
        protected:
                int n;
        public:
                Base(int i):n(i){
                        cout << "Base " << n <<
                                " constructed" << endl;    }
                ~Base() {
                        cout << "Base " << n <<
                                " destructed" << endl;
                }
                void Print() { cout << "Base:n=" << n << endl;}
};
```

```
class Derived:public Base  {
       public:
              int v;
              Derived(int i):Base(i),v(2 * i) {
                  cout << "Derived constructed" << endl;
              }
              ~Derived()    {
                     cout << "Derived destructed" << endl;
              }
              void Func() { } ;
              void Print() {
                     cout << "Derived:v=" << v << endl;
                     cout << "Derived:n=" << n << endl;
              }
};
```

```
int main()  {
        Base objBase(5);
        Derived objDerived(3);
        Base * pBase = & objDerived ;
        //pBase->Func(); //err;Base类没有Func()成员函数
        //pBase->v = 5; //err; Base类没有v成员变量
        pBase->Print();
        //Derived * pDerived = & objBase; //error
        Derived * pDerived = (Derived *)(& objBase);
        pDerived->Print();  //慎用，可能出现不可预期的错误
        pDerived->v = 128; //往别人的空间里写入数据，会有问题
        objDerived.Print();
        return 0;
}
```

Derived * pDerived = (Derived *)(& objBase);



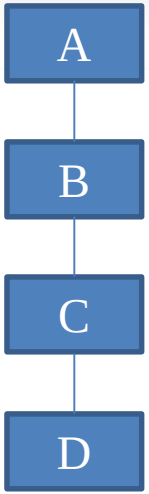
输出结果：

Base 5 constructed
Base 3 constructed
Derived constructed
Base:n=3
Derived:v=1245104 //pDerived->n 位于别人的空间里
Derived:n=5
Derived:v=6
Derived:n=3
Derived destructed
Base 3 destructed
Base 5 destructed

直接基类和间接基类

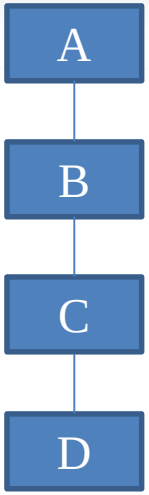
美国胡佛水坝

直接基类与间接基类



- 类A派生类B，类B派生类C，类C派生类D，……
  - 类A是类B的直接基类
  - 类B是类C的直接基类，类A是类C的间接基类
  - 类C是类D的直接基类，类A、B是类D的间接基类

直接基类与间接基类



- 在声明派生类时，只需要列出它的直接基类
  - 派生类沿着类的层次自动向上继承它的间接基类
  - 派生类的成员包括
    - 派生类自己定义的成员
    - 直接基类中的所有成员
    - 所有间接基类的全部成员

```
#include <iostream>
using namespace std;
class Base {
        public:
                int n;
                Base(int i):n(i)  {
                        cout << "Base " << n << " constructed"
                        << endl;
                }
                ~Base() {
                        cout << "Base " << n << " destructed"
                        << endl;
                }

};
```

```
class Derived:public Base
{
        public:
                Derived(int i):Base(i) {
                        cout << "Derived constructed" << endl;
                }
                ~Derived() {
                        cout << "Derived destructed" << endl;
                }

};
```

```
class MoreDerived:public Derived {
public:
	MoreDerived():Derived(4) {
		cout << "More Derived constructed" << endl;
	}
	~MoreDerived() {
		cout << "More Derived destructed" << endl;
	}
};
int main()
{
	MoreDerived Obj;
	return 0;
}
```

输出结果：

Base 4 constructed
Derived constructed
More Derived constructed
More Derived destructed
Derived destructed
Base 4 destructed

以下说法正确的是：

A) 派生类对象生成时，派生类的构造函数先于基类的构造函数执行

B) 派生类对象消亡时，基类的析构函数先于派生类的析构函数执行

C) 如果基类有无参构造函数，则派生类的构造函数就可以不带初始化列表

D) 在派生类的构造函数中部可以访问基类的成员变量

以下说法正确的是：

A) 派生类对象生成时，派生类的构造函数先于基类的构造函数执行

B) 派生类对象消亡时，基类的析构函数先于派生类的析构函数执行

C) 如果基类有无参构造函数，则派生类的构造函数就可以不带初始化列表

D) 在派生类的构造函数中部可以访问基类的成员变量

答案：C