程序设计实习

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

继承和派生

(教材P215)

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

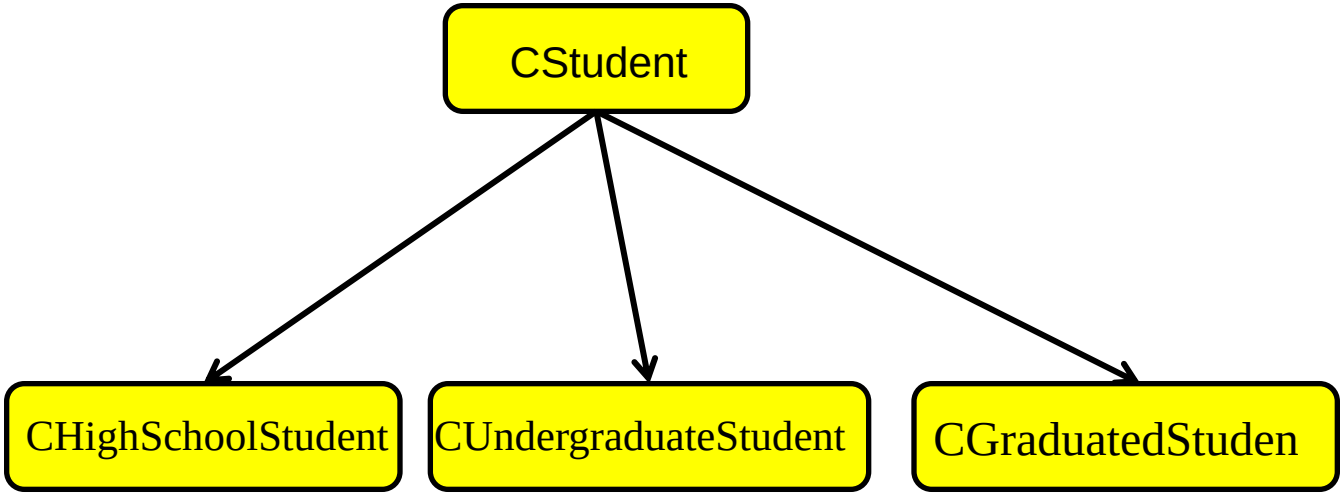
竞赛特长加分

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。

- 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
{

};
```

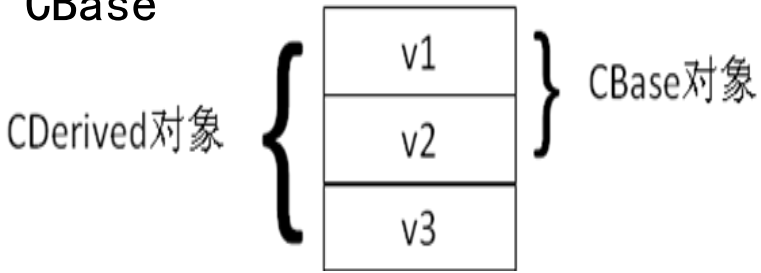
```
class CStudent {
        private:
                string sName;
                int nAge;
        public:
        bool IsThreeGood() { };
        void SetName( const string & name )
        { sName = name; }
        //.....
};
class CUndergraduateStudent: public CStudent {
        private:
                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;
};
```



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

程序设计实习

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

继承关系和复合关系 (P231)

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”。

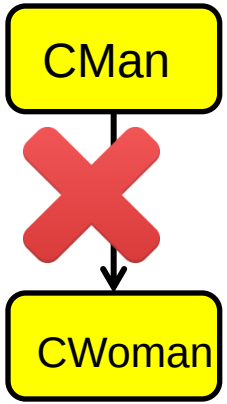
继承关系的使用

- 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

- CWoman类和CMan类有共同之处

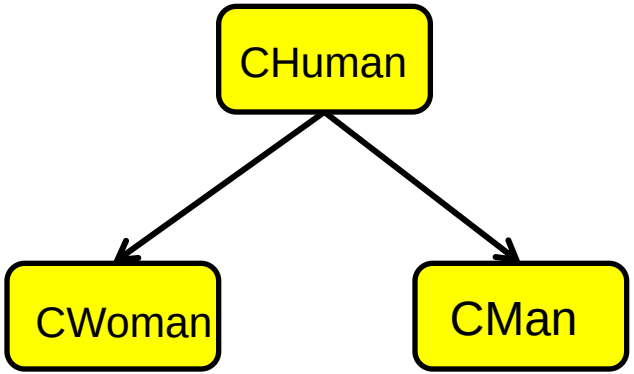
- 就让CWoman类从CMan类派生而来，是否合适？

- 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

```
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;
    friend class CCircle;
   //便于Ccirle类操作其圆心
};
```

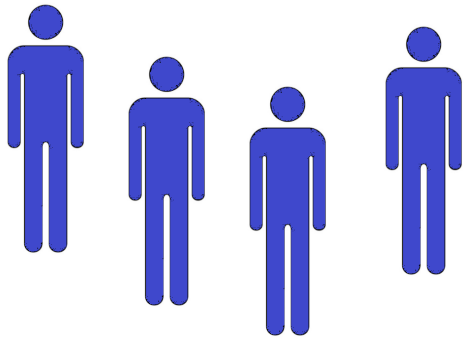
```
class CCircle
{
    double r;
    CPoint center;
};
```

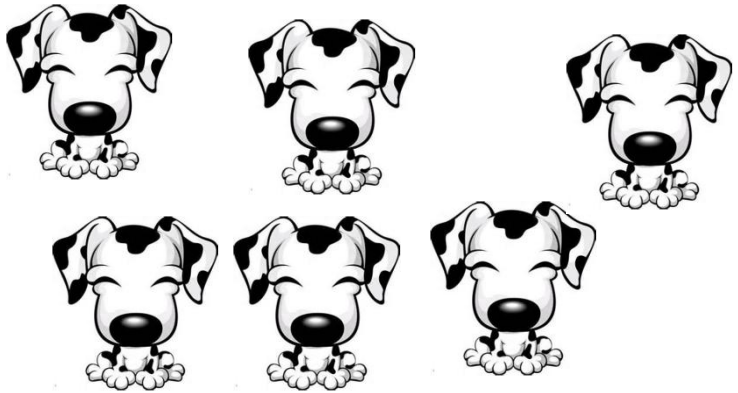
复合关系的使用

- 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。

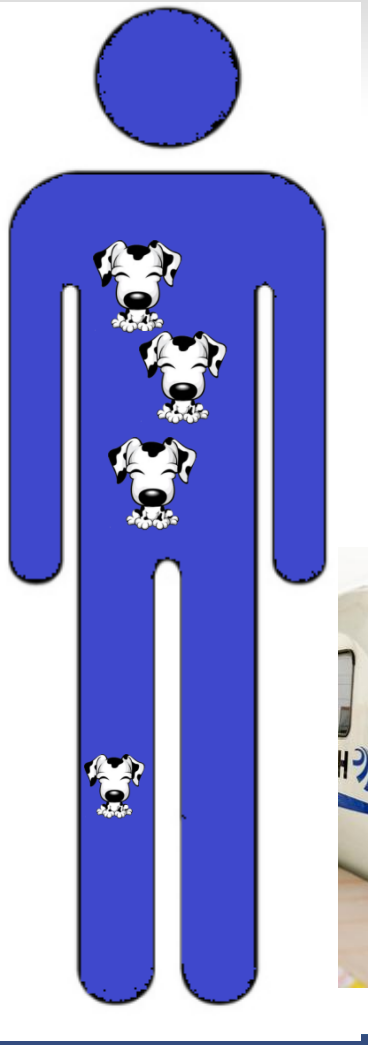
- 而狗是有 “主人” 的，主人当然是业主(假定狗只有一个主人，但一个业主可以有最多10条狗)



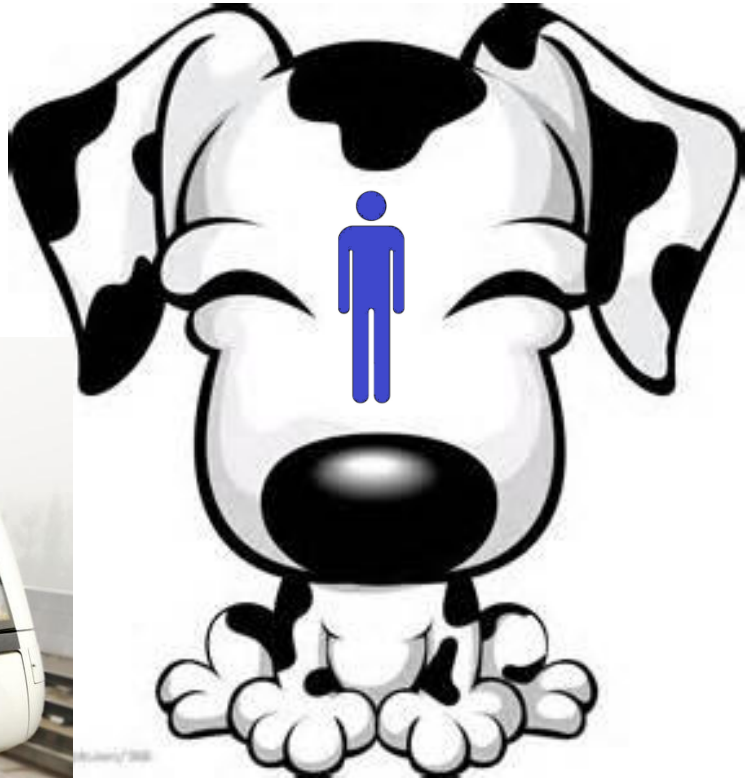


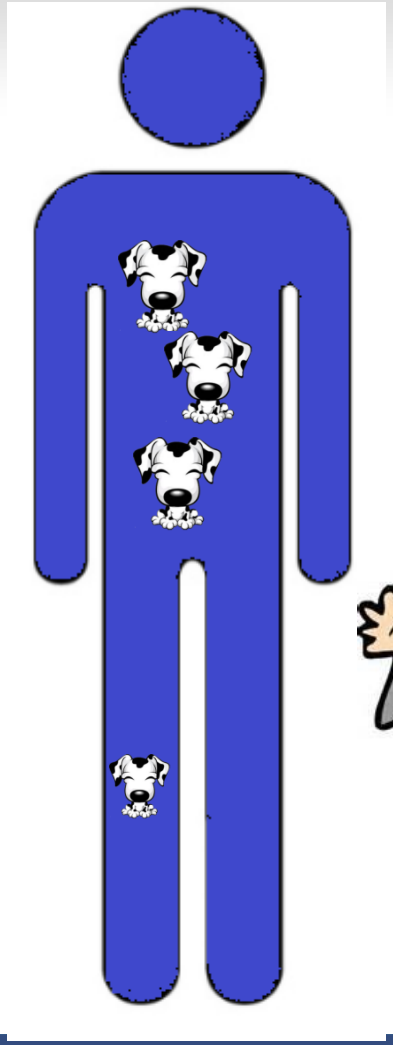
复合关系的使用

```
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;
};
```

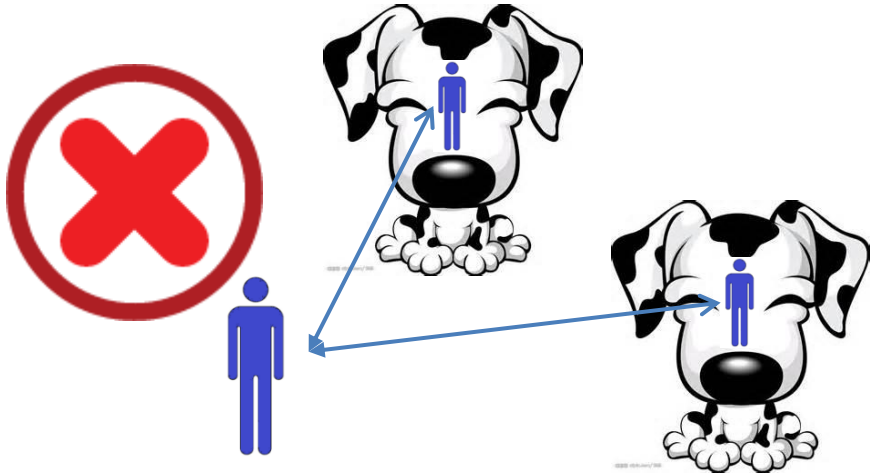
复合关系的使用

- 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;
};
```



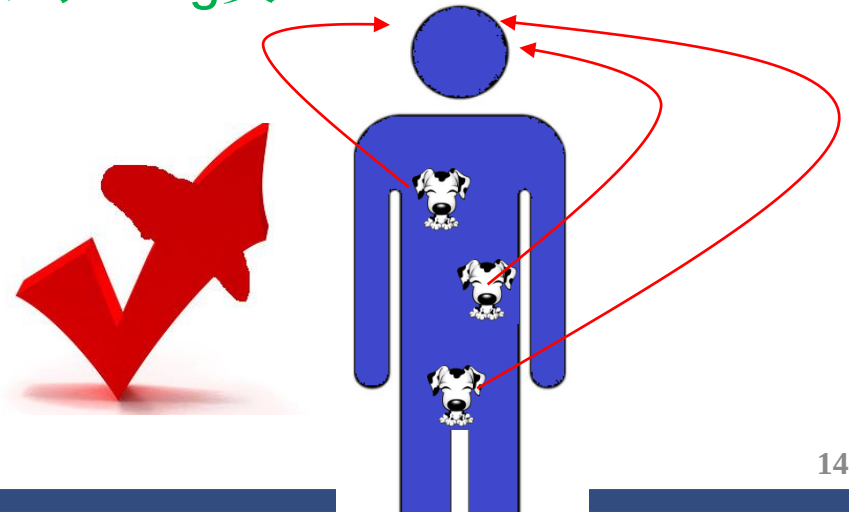
复合关系的使用

- 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];
};
```



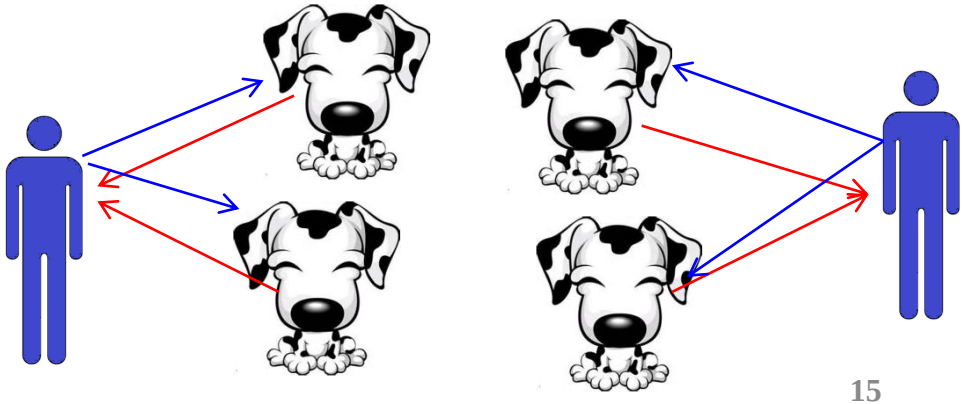
复合关系的使用

- 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];
};
```



基类/派生类同名成员与Protected关键字

郭 炜　刘家瑛

北京大学

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();
};
```

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;
```

Obj占用的存储空间

Note: 一般来说，基类和派生类不定义同名成员变量

访问范围说明符

- 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

访问范围说明符: protected

- 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

```
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

郭 炜 刘家瑛

北京大学

派生类的构造函数

- 派生类对象 包含 基类对象
- 执行派生类构造函数之前，先执行基类的构造函数
- 派生类交代基类初始化，具体形式：

```
构造函数名(形参表): 基类名(基类构造函数实参表)
{
}
```

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;
}
```

表达式中可以出现:
FlyBug构造函数的参数

```
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:

    派生类的构造函数中 → 基类的构造函数提供参数

    derived::derived(arg_derived-list):base(arg_base-list)
  - 隐式方式:

    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, 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

```
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;
}
```

表达式中可以出现:
FlyBug构造函数的
参数, 常量

- 创建 派生类的对象 时，执行 派生类的构造函数 之前:
  - 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
  - 调用 成员对象类 的构造函数

  →初始化派生类对象中成员对象
- 执行完 派生类的析构函数 后:
  - 调用 成员对象类 的析构函数
  - 调用 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

程序设计实习

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

public继承的赋值兼容规则

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;
```

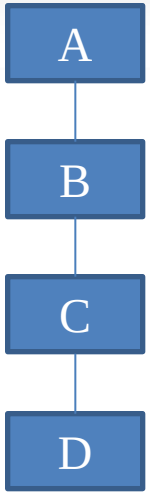
3） 派生类对象的地址可以赋值给基类指针

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

- 如果派生方式是 private或protected，则上述三条不可行。

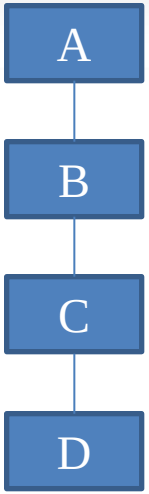
直接基类和间接基类

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