

2018年春

# 程序设计实习 (I): C++程序设计

## 第七讲 多 态

刘家瑛

*liujiaying@pku.edu.cn*



# 课前多吼歪

## □ 本周五合班上课通知

■ 3月30日第5周 周五7-8节 **理教107**

■ 郭老师代课

## □ 见证魔兽3的诞生

■ 在魔兽2和终极版之间



# 上节课内容回顾

- 基本概念: 继承, 基类, 派生类
  - 合理派生
- 派生类的成员组成/可见性
  - private/protected成员的继承性
  - 派生类的成员函数 — 不能访问基类中的private成员
  - 派生类的成员函数 — 访问当前对象的基类的protected成员
- 派生类的构造/析构
  - 构造顺序: 基类→对象成员→派生
  - 析构反之
- 派生类与基类的指针类型转换
  - f(派生类)→y(基类对象): **f: 对象/对象地址; y: 对象/引用/指针**
  - 基类指针→(强制指针类型转换)派生类指针



# 上节课知识点回顾

- 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



# 上节课知识点回顾

- 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



# 上节课知识点回顾

- 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不可以访问基类的成员变量





# 上节课知识点回顾

- 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



# 主要内容

- 多态的基本概念
- 虚函数
- 多态的作用
- 动态联编
- 虚析构函数
- 纯虚函数和抽象类





# 面向对象的程序设计特点

## 几个重要关键词

- 封装 – 类的抽象和设计 / 可见性
- 继承 – 基类派生派生类 / 代码的继承
- 多态 – 函数重载 / ?



# 多态的基本概念

□ 派生类的**指针**可以赋给基类**指针**

□ 通过**基类指针**调用基类和派生类中的同名**虚函数**

(一种特殊的成员函数) 时:

- 若该指针指向一个**基类**的对象, 则被调用是**基类**的虚函数
- 若该指针指向一个**派生类**的对象, 则被调用的是**派生类**的虚函数

这种机制就叫做 **多态**

□ 例如:

```
CBase * p = &ODerived;
```

```
p -> SomeVirtualFunction();
```

//即根据 指针/引用指向的对象类型 来决定调用的函数



- 派生类的对象可以赋给基类引用
- 通过该基类引用调用基类和派生类中的同名虚函数时:
  - 若该引用引用的是一个基类的对象, 那么被调用是基类的虚函数
  - 若引用的是一个派生类的对象, 那么被调用的是派生类的虚函数

这种机制也叫 多态

□ 例如:

```
CBase & r = ODerived;
```

```
r.SomeVirtualFunction();
```

“多态”就是指上述这两种机制



# 虚函数

- 在类的定义中, 前面有 **virtual** 关键字的成员函数就是**虚函数**

```
class base {  
    virtual int get() ;  
};  
int base::get()  
{ }
```

- virtual 关键字 **只用在类定义里的函数声明** 中
- 不能在类外部编写的函数体时使用
- 静态成员函数不能是虚函数

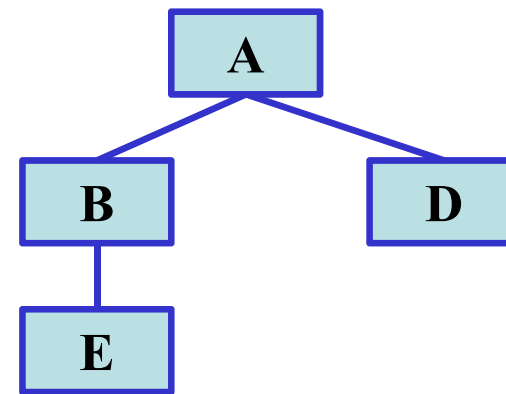


# 多态示例1: 通过指针/引用访问虚函数

```
class A {  
    public :  
        virtual void Print( ) { cout << "A::Print" << endl ; }  
};  
class B: public A {  
    public :  
        virtual void Print( ) { cout << "B::Print" << endl ; }  
};  
class D: public A {  
    public:  
        virtual void Print( ) { cout << "D::Print" << endl ; }  
};  
class E: public B {  
        virtual void Print( ) { cout << "E::Print" << endl ; }  
};
```



```
int main() {  
    A a;  B b;  E e;  D d;  
    A * pa = &a; B * pb = &b;  
    D * pd = &d; E * pe = &e;
```



```
pa->Print();    //a.Print()被调用, 输出: A::Print  
pa = pb;        //派生类指针赋值给基类指针  
pa -> Print();   //b.Print()被调用, 输出: B::Print  
pa = pd;  
pa -> Print();   //d.Print()被调用, 输出: D::Print  
pa = pe;  
pa -> Print();   //e.Print()被调用, 输出: E::Print  
return 0;  
}
```



# 将上述例子稍加修改

```
class A {  
    public :  
    virtual void Print( ) { cout << "A::Print"<<endl ; }  
};  
class B:public A {  
    public :  
    void Print( ) { cout << "B::Print" <<endl; }  
};  
class D: public A {  
    public:  
    void Print( ) { cout << "D::Print" << endl ; }  
};  
class E: public B {  
    void Print( ) { cout << "E::Print" << endl ; }  
};
```

- main函数不变，所得的结果一样
- 在多层继承的情况下，从**定义virtual**开始的派生类中同名函数均为**虚函数**，无论在这些派生类的同名函数中是否显示加virtual





## 多态示例2:基类普通成员函数中访问虚函数

```
class myclass{
    public:
        void reset();
        void virtual print(void) //virtual可以写在返回值前或后
        {    cout<<"myclass::print\n";    }
};

void myclass::reset() {
    print(); //等效于 this->print(); this是基类指针
}

class derived: public myclass{
    public:
        void virtual print() { cout<<"derived::print\n"; }
};
```



```
int main(){  
    myclass b;  
    derived d;  
    b.reset();  
    d.reset();  
    return 0;  
}
```

**输出:**  
**myclass::print**  
**derived::print**



# 多态的作用

## □ 多态的实质

- 父类定义共同接口，子类不同实现
- 通过父类以相同的方式操作不同子类的行为

## □ 在面向对象的程序设计中使用时多态

- 能够增强程序的扩充性
- 程序需要修改或增加功能的时候，  
需要改动和增加的代码较少



# 多态增强程序可扩充性的例子

□ 游戏《魔法门之英雄无敌》



## 类CSoldier



## 类CDragon



## 类CPhoenix



## 类CAngel



游戏中有很多种怪物(其他的如Wolf, Ghost)

每种怪物都有一个类与之对应, **每个怪物就是一个对象**





怪物能够互相攻击,攻击敌人和被攻击时都有相应的动作, **动作是通过对象的成员函数实现的**



游戏版本升级时，要增加新的怪物——雷鸟。如何编程才能使升级时的代码改动和增加量较小？



新增类：CThunderBird





## □ 不论是否用多态编程, 基本思路都是:

- 为每个怪物类编写 **Attack**, **FightBack** 和 **Hurted** 成员函数
- **Attack** 函数表现攻击动作, 攻击某个怪物
  - 调用被攻击怪物的 **Hurted** 函数, 以减少被攻击怪物的生命值
  - 调用被攻击怪物的 **FightBack** 成员函数, 遭受被攻击怪物反击
- **Hurted** 函数减少自身生命值, 并表现受伤动作
- **FightBack** 函数表现反击动作, 并调用被反击对象的 **Hurted** 成员函数, 使被反击对象受伤



# 非多态的实现方法

```
class CDragon {  
    private:  
        int nPower;    //代表攻击力  
        int nLifeValue; //代表生命值  
    public:  
        void Attack(CWolf * pWolf) { //攻击Wolf的成员函数  
            ... 表现攻击动作的代码  
            pWolf->Hurled(nPower);  
            pWolf->FightBack(this); //指向攻击者自身的this指针  
        }  
        void Attack(CGhost * pGhost) { //攻击Ghost的成员函数  
            ... 表现攻击动作的代码  
            pGhost->Hurled(nPower);  
            pGohst->FightBack(this);  
        }  
}
```



```

void Hurted (int nPower) {
    ... 表现受伤动作的代码
    nLifeValue -= nPower;
}
void FightBack(CWolf * pWolf) {
    ... 表现反击动作的代码
    pWolf ->Hurted(nPower/ 2); //使攻击者受伤,
                                //反攻功力减半
}
void FightBack(CGhost * pGhost) {
    ... 表现反击动作的代码
    pGhost->Hurted(nPower/ 2);
}
}

```

有n种怪物, CDragon 中就会有**n个Attack 成员函数**,  
以及 **n个FightBack 成员函数**. 对于其他类也如此



- 如果游戏版本升级，增加了新的怪物雷鸟CThunderBird，则程序改动较大，**所有的类都需要增加：**

**void Attack( CThunderBird \* pThunderBird) ;**

**void FightBack( CThunderBird \* pThunderBird) ;**

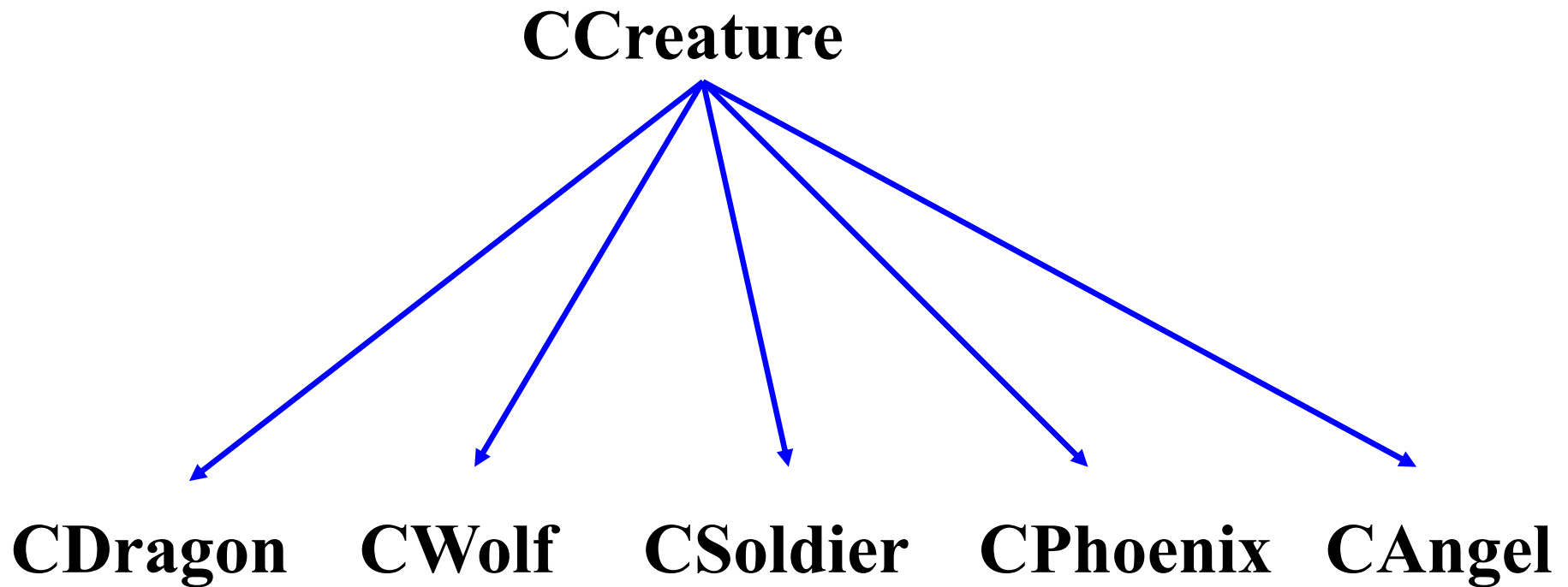
在怪物种类多的时候，工作量较大

- 非多态实现中，代码更精简的做法是将Cdragon, CWolf等类的共同特点抽取出来 → C Creature类
- 然后再从C Creature类派生出CDragon, CWolf等类
- 但是由于每种怪物进行攻击, 反击和受伤时的表现动作不同, Cdragon, CWolf这些类还是要实现各自的Hurted成员函数, 以及一系列Attack, FightBack成员函数
- **所以只要没有利用多态机制, 那么即便引入基类C Creature, 对程序的可扩充性也无帮助**



# 使用多态的实现方法

- 设置基类 C Creature, 并且使 C Dragon, C Wolf 等其他类都从 C Creature 派生而来



## 基类 C Creature:

```
class C Creature { // "怪物" 类
    protected :
        int m_nLifeValue, m_nPower;
    public:
        virtual void Attack( C Creature * pCreature);
        virtual void Hurted( int nPower);
        virtual void FightBack( C Creature * pCreature);
};
```

- 基类只有一个Attack 成员函数
- 也只有一个FightBack成员函数
- 所有CCreature 的派生类也是这样





## 派生类 CDragon:

```
class CDragon : public CCreature {  
    public:  
        virtual void Attack( CCreature * pCreature);  
        virtual void Hurted( int nPower);  
        virtual void FightBack( CCreature * pCreature);  
};
```





```
void CDragon::Attack(CCreature * p)
{
    p->Hurted(m_nPower);
    p->FightBack(this);
}
void CDragon::Hurted( int nPower)
{
    m_nLifeValue -= nPower;
}
void CDragon::FightBack(CCreature * p)
{
    p->Hurted(m_nPower/2);
}
```



那么当增加新怪物雷鸟的时候, 只需要编写新类  
CThunderBird, 不需要在已有的类里专门为新怪物增加:

**void Attack( CThunderBird \* pThunderBird);**

**void FightBack( CThunderBird \* pThunderBird);**

成员函数



具体使用这些类的代码：

**CDragon Dragon;**

**CWolf Wolf;**

**CGhost Ghost;**

**CThunderBird Bird;**

**Dragon.Attack( & Wolf );**       **//(1)**

**Dragon.Attack( & Ghost );**       **//(2)**

**Dragon.Attack( & Bird );**       **//(3)**

□ 根据多态的规则，上面的(1), (2), (3)进入到**CDragon::Attack**函数后，能分别调用：

**CWolf::Hurled**

**CGhost::Hurled**

**CBird::Hurled**



# 使用多态的另一例子:

几何形体处理程序：输入若干个几何形体的参数，要求按面积排序输出。输出时要指明形状

## Input:

第一行是几何形体数目 $n$  (不超过100)，下面有 $n$ 行，每行以一个字母 $c$ 开头

若  $c$  是 'R'，则代表一个矩形，本行后面跟着两个整数，分别是矩形的宽和高

若  $c$  是 'C'，则代表一个圆，本行后面跟着一个整数代表其半径

若  $c$  是 'T'，则代表一个三角形，本行后面跟着三个整数，代表三条边的长度



# 使用多态的另一例子:

□ 几何形体处理程序:

- 输入若干个几何形体的参数, 要求按面积排序输出
- 输出时要指明形状

## Output:

按面积从小到大依次输出每个几何形体的种类及面积

每行一个几何形体, 输出格式为:

形体名称:面积



# 使用多态的另一例子:

## Sample Input:

3

R 3 5

C 9

T 3 4 5

## Sample Output:

Triangle: 6

Rectangle:15

Circle: 254.34



```
#include <iostream>
#include <stdlib.h> //快速排序函数
#include <math.h> //数学函数
using namespace std;
class Cshape {
    public:
        virtual double Area(){ };
        virtual void PrintInfo(){ };
};
class CRectangle: public Cshape {
    public:
        int w, h;
        virtual double Area();
        virtual void PrintInfo();
};
```





```
class CCircle: public CShape  
{  
    public:  
        int r;  
        virtual double Area();  
        virtual void PrintInfo();  
};  
class CTriangle: public CShape  
{  
    public:  
        int a, b, c;  
        virtual double Area();  
        virtual void PrintInfo();  
};
```



```
double CRectangle::Area() {  
    return w * h;  
}  
  
void CRectangle::PrintInfo() {  
    cout << "Rectangle:" << Area() << endl;  
}  
  
double CCircle::Area() {  
    return 3.14 * r * r;  
}  
  
void CCircle::PrintInfo() {  
    cout << "Circle:" << Area() << endl;  
}  
  
double CTriangle::Area() {  
    double p = (a + b + c) / 2.0;  
    return sqrt(p * (p - a)*(p - b)*(p - c));  
}  
  
void CTriangle::PrintInfo() {  
    cout << "Triangle:" << Area() << endl; }
```



**CShape \* pShapes[100];** //用来存放各种几何形体,  
//假设不超过100

基类指针也能指向各派生类对象  
每输入一个几何形体,  
就动态分配一个相应类的对象,  
将该对象的指针存入pShapes数组



```
CShape * pShapes[100]; //用来存放各种几何形体,假设不超过100
int MyCompare(const void * s1, const void * s2){
    CShape * * p1 ; //s1是指向指针的指针,其指向的指针为Cshape*
    CShape * * p2;
    p1 = ( CShape * * ) s1;
    p2 = ( CShape * * ) s2;
    double a1 = (*p1)->Area(); //p1指向几何形体对象的指针
    double a2 = (*p2)->Area(); //p2指向对象
    if( a1 < a2 )
        return -1;
    else if ( a2 < a1 )
        return 1;
    else
        return 0;
}
```



```
int main()
{
    int i; int n;
    CRectangle * pr; CCircle * pc; CTriangle * pt;
    cin >> n;
    for( i = 0; i < n; i ++ ) {
        char c;
        cin >> c;
        switch(c) {
            case 'R':
                pr = new CRectangle(); //new返回的是对象指针
                cin >> pr->w >> pr->h;
                pShapes[i] = pr;
                break;
```



```
case 'C':
```

```
    pc = new CCircle();
```

```
    cin >> pc->r;
```

```
    pShapes[i] = pc;
```

```
    break;
```

```
case 'T':
```

```
    pt = new CTriangle();
```

```
    cin >> pt->a >> pt->b >> pt->c;
```

```
    pShapes[i] = pt;
```

```
    break;
```

```
}
```

```
}
```

```
qsort(pShapes, n, sizeof( CShape*), MyCompare);
```

```
for( i = 0; i <n; i ++)
```

```
    pShapes[i]->PrintInfo(); //多态
```

```
return 0;
```



# 附录：访问机制对比

□ 覆盖：所访问的对象由**指针**  
(或引用)的类型所决定

- 用**派生类指针**访问这类成员时，访问的是**派生类中定义的(同名)成员函数**
- 用**基类指针**访问这类成员时，访问的是**基类的(同名)成员函数**，即使它指向的是一个派生类对象



□ 虚函数：所访问的对象由**指针**  
(或引用)所指向的对象类型所决定

- 若该指针(或引用)指向一个基类的对象，那么被调用是**基类的虚函数**
- 如果该指针(或引用)指向一个派生类的对象，那么被调用的是**派生类的虚函数**





## 面向对象编程的三个基本概念：

- 数据抽象
- 继承
- 动态绑定



## 面向对象编程的三个基本概念：

- 数据抽象  $\longleftrightarrow$  用 **类** 进行数据抽象，封装
- 继承  $\longleftrightarrow$  **派生类继承** 基类成员
- 动态绑定  $\longleftrightarrow$  **多态**，编译器能够在运行时决定是使用基类中定义的函数还是派生类中定义的函数



# 动态联编

一条函数调用语句在编译时无法确定调用哪个函数，  
运行到该语句时才确定调用哪个函数，  
这种机制叫动态联编



# 为什么需要动态联编？

```
class A { public: virtual void Get(); };  
class B : public A { public: virtual void Get(); };  
void MyFunction( A * pa ) {  
    pa->Get();  
}
```

□ pa->Get() 调用的是 A::Get() 还是 B::Get()?

在编译时无法确定, 因为不知道MyFunction被调用时,  
形参会对应于一个 A 对象还是 B 对象

□ 所以只能等程序运行到 pa->Get() 了,  
才能决定到底调用哪个Get()



# 多态的又一例子

```
class Base {  
public:  
    void fun1() { fun2(); }  
    virtual void fun2() { cout << "Base::fun2()" << endl; }  
};  
class Derived:public Base {  
public:  
    virtual void fun2() { cout << "Derived:fun2()" << endl; }  
};  
int main() {  
    Derived d;  
    Base * pBase = & d;  
    pBase->fun1();  
    return 0;  
}
```

输出: Derived:fun2()



□ 调用的次序是: **Base::fun1() -> Derived::fun2();**

因为

```
void fun1() {  
    fun2();  
}
```

□ 相当于

```
void fun1() {  
    this->fun2();  
}
```

□ 编译这个函数的代码的时候, 由于fun2()是虚函数, this是基类指针, 所以是动态联编

□ 程序运行到fun1函数中时, **this指针指向的是d**, 所以经过动态联编, 调用的是Derived::fun2()



## 稍加修改

```
class Base {  
public:  
    virtual void fun1() { cout << "Base::fun1()" << endl; fun2(); }  
    void fun2() { cout << "Base::fun2()" << endl; }  
};  
class Derived: public Base {  
    virtual void fun1() { cout << "Derived::fun1()" << endl; fun2(); }  
    void fun2() { cout << "Derived:fun2()" << endl; }  
};  
int main() {  
    Derived d;  
    Base * pBase = & d;  
    pBase -> fun1();  
    return 0;  
}
```

输出: Derived:fun1()  
Derived:fun2()





## 思考

“多态”的关键在于**通过基类指针或引用**调用一个虚函数时,编译时不确定到底调用的是基类还是派生类的函数,运行时才确定.  
这到底是怎么实现的呢?



提示：请看下面例子程序：

```
class Base1 {  
    public:  
        int i;  
        virtual void Print() { cout << "Base1:Print" ; }  
};  
class Derived : public Base1{  
    public:  
        int n;  
        virtual void Print() { cout <<"Drived:Print"<< endl; }  
};  
int main() {  
    Derived d;  
    cout << sizeof( Base1) << ", " << sizeof( Derived ) ;  
    return 0;  
}
```

程序运行输出结果: 8, 12



- 为什么Base 对象的大小是8个字节而不是4个字节?
- 为什么Derived 对象的大小是12个字节而不是8个字节?
- 多出来的4个字节做什么用呢?
- 和多态的实现有什么关系?

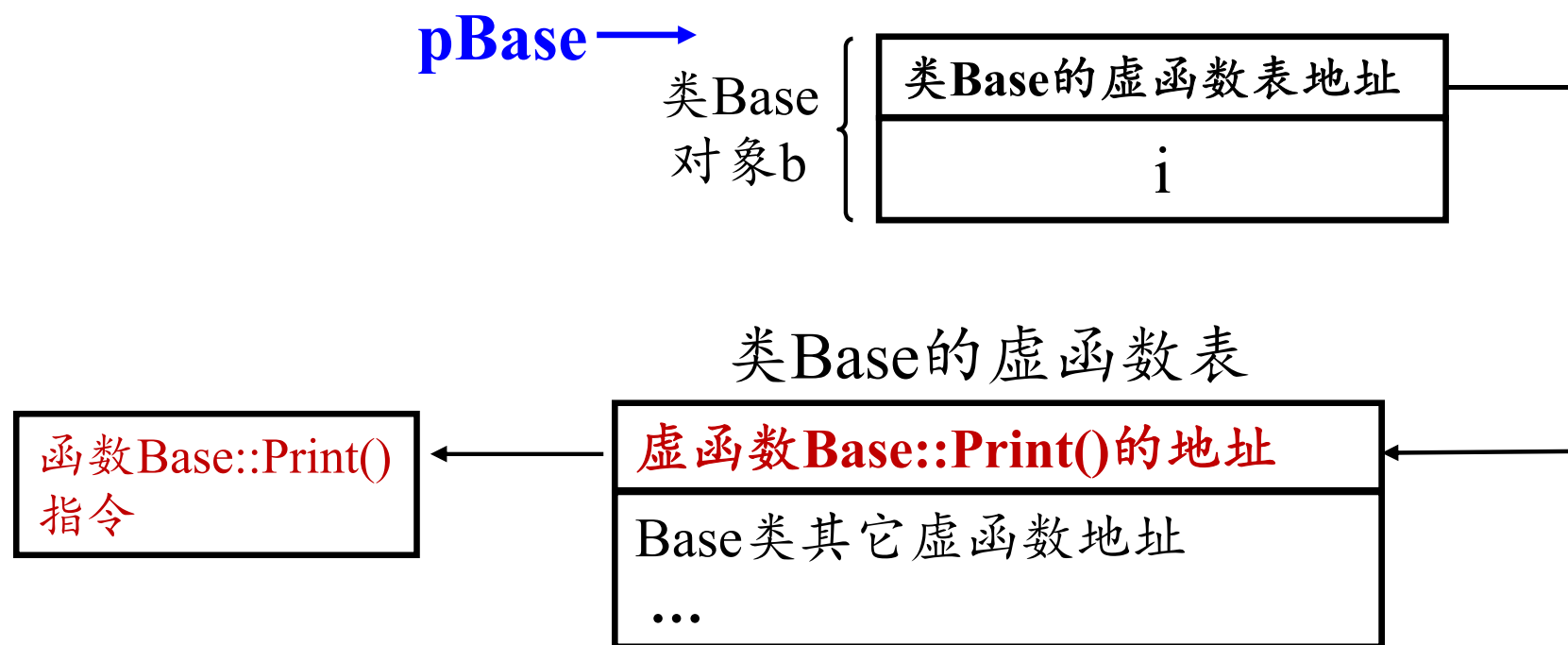


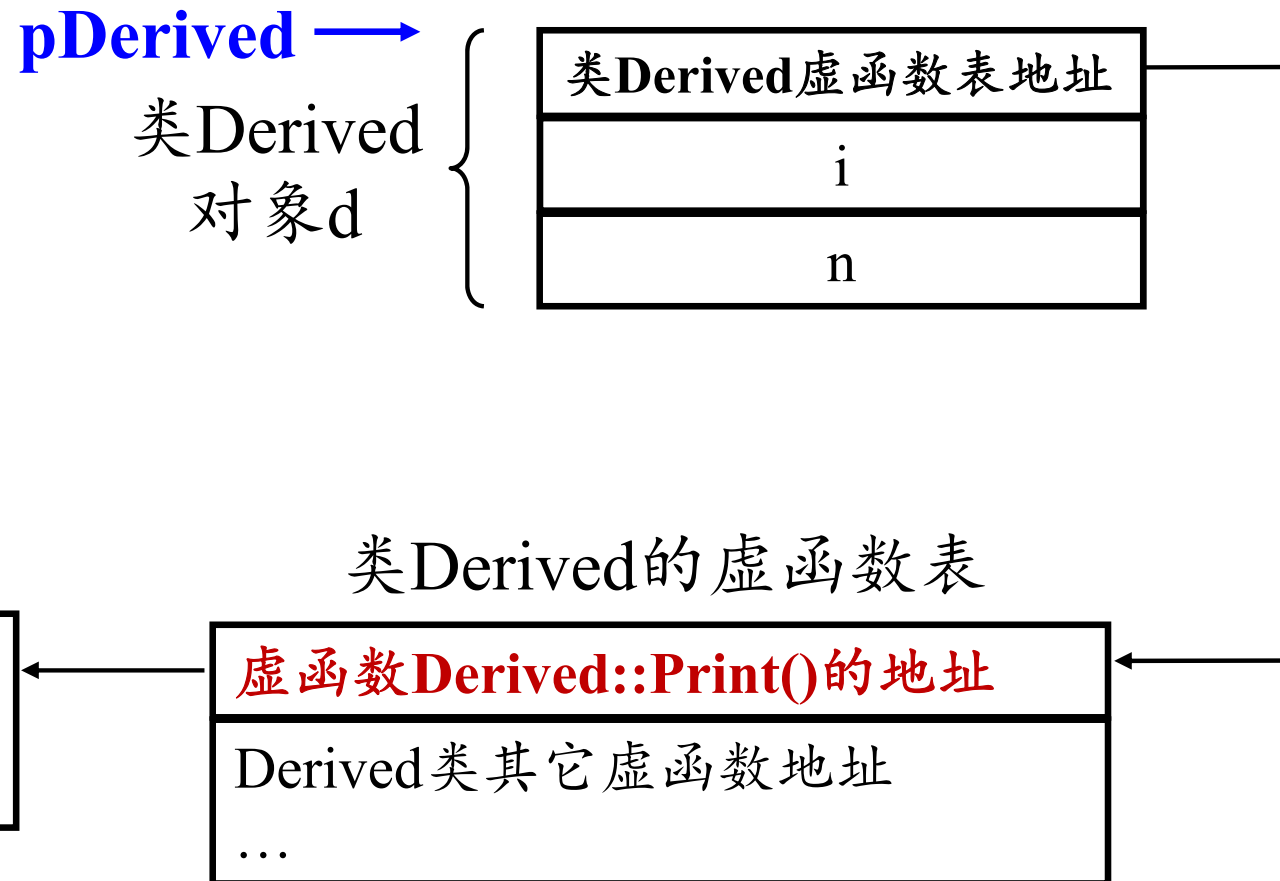
# 多态的实现: 虚函数表

□ 每个有虚函数的类(或有虚函数的类的派生类)都有一个**虚函数表**, 该类的**任何对象**中都放着虚函数表的指针

□ 虚函数表中列出了该类的虚函数地址

多出来的4个字节就是用来放虚函数表的地址





```
*pBase = &pDerived;  
pBase->Print();
```



□ 多态的函数调用语句被编译成一系列根据**基类指针所指向的(或基类引用所引用的)对象**中存放的**虚函数表的地址**

→ 在虚函数表中查找虚函数地址, 并调用虚函数





# 虚函数的访问权限

```
class Base {  
    private:  
        virtual void fun2() { cout << "Base::fun2()" << endl; }  
};  
class Derived:public Base {  
    public:  
        virtual void fun2() { cout << "Derived:fun2()" << endl; }  
};  
Derived d;  
Base * pBase = & d;  
pBase -> fun2(); // 编译出错
```

访问权限检查是根据指针类型来的

考虑运行结果的

- 如果将Base中的 private换成public, 即使Derived中的fun2()是private的, 编译依然能通过, 也能正确调用Derived::fun2()



# 构造函数和析构函数中调用虚函数

- 在构造函数和析构函数中调用虚函数时：
  - 调用的函数是**自己的类或基类**中定义的函数,不会等到运行时才决定调用自己的还是派生类的函数
- 在普通成员函数中调用虚函数,则是**动态联编,是多态**

```
class myclass{  
    public:  
        virtual void hello(){ cout<<"hello from myclass"<<endl; };  
        virtual void bye(){ cout<<"bye from myclass"<<endl; };  
};
```



```
class son:public myclass{
```

```
    public:
```

```
        void hello(){    cout<<"hello from son"<<endl;    };
```

```
        son(){    hello();    };
```

```
        ~son(){    bye();    };
```

```
};
```

```
class grandson:public son{
```

```
    public:
```

```
        void hello(){ cout<<"hello from grandson"<<endl; };
```

```
        grandson(){ cout<<"constructing grandson"<<endl; };
```

```
        ~grandson(){ cout<<"destructing grandson"<<endl; };
```

```
};
```



```

int main(){
    grandson gson;
    son *pson;
    pson=&gson;
    pson->hello(); //void grandson::hello()
    return 0;
}

```

输出结果:

hello from son // gson先创建son, son()中静态连接了son::hello()  
 constructing grandson // gson的创建  
 hello from grandson // pson->hello()动态连接到grandson::hello()  
 destructing grandson // gson的创建的析构  
 bye from myclass // gson中son部分的析构, ~son()中静态  
 //连接了myclass::bye()



# 虚析构函数(1)

- 通过基类的指针删除派生类对象时
- 通常情况下**只调用基类的析构函数**
  - 但是删除一个派生类的对象时, 应该先调用派生类的析构函数, 然后调用基类的析构函数



```
class son{
    public:
        ~son() { cout<<"bye from son"<<endl; };
};
class grandson : public son{
    public:
        ~grandson(){ cout<<"bye from grandson"<<endl; };
};
int main(){
    son *pson;
    pson=new grandson;
    delete pson;
    return 0;
}
```

输出结果: bye from son

没有执行grandson::~~grandson()!!!





## 虚析构函数(2)

- 解决办法: 把基类的析构函数声明为 **virtual**
  - 派生类的析构函数可以不声明为 **virtual**
  - 通过基类的指针删除派生类对象时, 首先调用派生类的析构函数, 然后调用基类的析构函数
- 一般来说, 一个类如果定义了虚函数, 则应该将析构函数也定义成虚函数
- 注意: 不允许以虚函数作为构造函数



```
class son{
    public:
        virtual ~son() {cout<<"bye from son"<<endl;};
};
class grandson:public son{
    public:
        ~grandson(){cout<<"bye from grandson"<<endl;};
};
int main() {
    son *pson;
    pson= new grandson();
    delete pson;
    return 0;
}
```

输出结果: bye from grandson  
bye from son

执行grandson::~~grandson(), 引起执行son::~~son() ! ! !

# 纯虚函数和抽象类

## □ 纯虚函数：没有函数体的虚函数

```
class A {  
    private: int a;  
    public:  
        virtual void Print( ) = 0 ; //纯虚函数  
        void fun() { cout << "fun"; }  
};
```

## □ 包含纯虚函数的类叫抽象类

- 抽象类只能作为基类来派生新类使用, 不能创建抽象类的对象
- 抽象类的指针和引用可以指向由抽象类派生出来的类的对象

A a ; // 错误, A 是抽象类, 不能创建对象

A \* pa ; // ok, 可以定义抽象类的指针和引用

pa = new A ; //错误, A 是抽象类, 不能创建对象



- 在抽象类的**成员函数**内可以调用纯虚函数
- 但是在**构造函数**或**析构函数**内部**不能**调用纯虚函数
- 如果一个类从抽象类派生而来, 那么当且仅当它实现了基类中的**所有纯虚函数**, 它才能成为非抽象类



```

class A {
    public:
        virtual void f() = 0; //纯虚函数
        void g() { this->f(); } //ok
        A(){ } //f(); // 错误
};

class B:public A{
    public:
        void f(){ cout<<"B:f()"<<endl; }
};

int main(){
    B b;
    b.g();
    return 0;
}

```

输出结果:  
B:f()



# 总结

## □ 虚函数和多态

- 虚函数的调用规则：根据指向的对象确定

## □ 多态的作用：可扩展

## □ 多态的实现：虚函数表

## □ 虚函数的应用

- 虚函数的访问权限：基类虚函数应为public
- 纯虚函数和抽象类：抽象类不能生成对象
- 成员函数中调用虚函数：只有在普通成员函数中调用虚函数才是动态联编
- 虚析构函数：定义虚函数的类析构函数应为virtual；不允许以虚函数作为构造函数





