



# 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>



北京大学  
PEKING UNIVERSITY

信息科学技术学院

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著





## 虚函数和多态



# 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，写函数体时不用。

# 多态的表现形式一

- 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时：
  - (1) 若该指针指向一个基类的对象，那么被调用是基类的虚函数；
  - (2) 若该指针指向一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制就叫做“多态”。

# 多态的表现形式一

```
class CBase {  
    public:  
        virtual void SomeVirtualFunction() { }  
};  
class CDerived:public CBase {  
    public :  
        virtual void SomeVirtualFunction() { }  
};  
int main() {  
    CDerived ODerived;  
    CBase * p = & ODerived;  
    p -> SomeVirtualFunction(); //调用哪个虚函数取决于p指向哪种类型的对象  
    return 0;  
}
```

## 多态的表现形式二

- 派生类的对象可以赋给基类引用
- 通过基类引用调用基类和派生类中的同名虚函数时：
  - (1) 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；
  - (2) 若该引用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。

这种机制也叫做“多态”。

## 多态的表现形式二

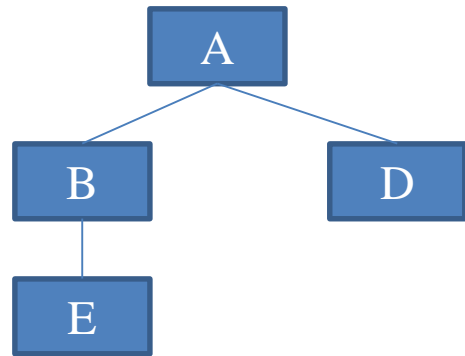
```
class CBase {  
    public:  
        virtual void SomeVirtualFunction() { }  
};  
class CDerived:public CBase {  
    public :  
        virtual void SomeVirtualFunction() { }  
};  
int main() {  
    CDerived ODerived;  
    CBase & r = ODerived;  
    r.SomeVirtualFunction(); //调用哪个虚函数取决于r引用哪种类型的对象  
    return 0;  
}
```



## 多态的简单示例

```
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();  
    pa = pb;  
    pa -> Print();  
    pa = pd;  
    pa -> Print();  
    pa = pe;  
    pa -> Print();  
    return 0;  
}
```

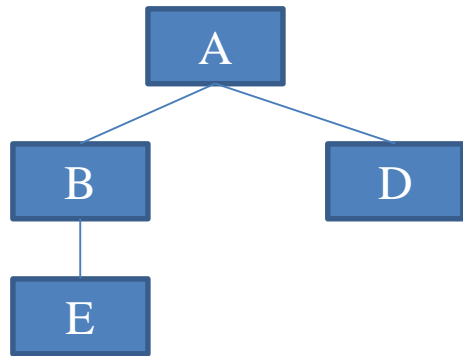


```
int main() {  
    A a; B b;    E e; D d;  
    A * pa = &a; B * pb = &b;  
    D * pd = &d ; E * pe = &e;
```

pa->Print();     // a.Print()被调用， 输出： A::Print

```
    pa = pb;  
    pa -> Print();  
    pa = pd;  
    pa -> Print();  
    pa = pe;  
    pa -> Print();  
    return 0;
```

```
}
```



```
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;

pa -> Print(); //b.Print()被调用, 输出: B::Print

pa = pd;

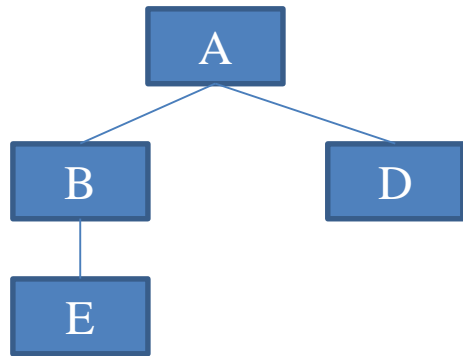
pa -> Print();

pa = pe;

pa -> Print();

return 0;

}



```
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;

pa -> Print(); //b.Print()被调用, 输出: B::Print

pa = pd;

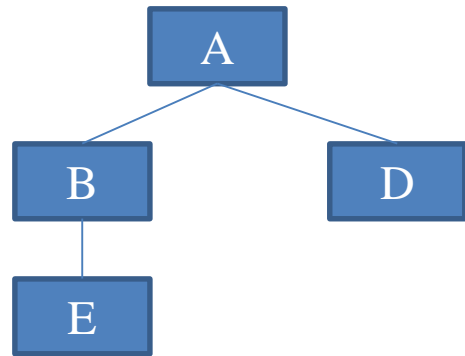
pa -> Print(); //d. Print ()被调用,输出: D::Print

pa = pe;

pa -> Print();

return 0;

}



```
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;

pa -> Print(); //b.Print()被调用, 输出: B::Print

pa = pd;

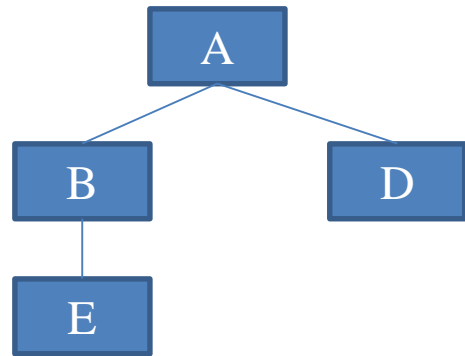
pa -> Print(); //d. Print ()被调用,输出: D::Print

pa = pe;

pa -> Print(); //e.Print () 被调用,输出: E::Print

return 0;

}



# 多态的作用

在面向对象的程序设计中，使用多态，能够增强程序的可扩充性，即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。



## 使用多态的 游戏程序实例





# 游戏《魔法门之英雄无敌》

游戏中有很多种怪物，每种怪物都有一个类与之对应，  
每个怪物就是一个对象。



类: CSoldier



类CPhonex

类: CDragon



类: CAngel

# 游戏《魔法门之英雄无敌》

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《魔法门之英雄无敌》

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

## 基本思路：

- 为每个怪物类编写 `Attack`、`FightBack`和 `Hurted`成员函数。

## 基本思路：

- 为每个怪物类编写 `Attack`、`FightBack`和 `Hurted`成员函数。
- `Attact`函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 `Hurted`函数，以减少被攻击怪物的生命值，同时也调用被攻击怪物的 `FightBack`成员函数，遭受被攻击怪物反击。

## 基本思路：

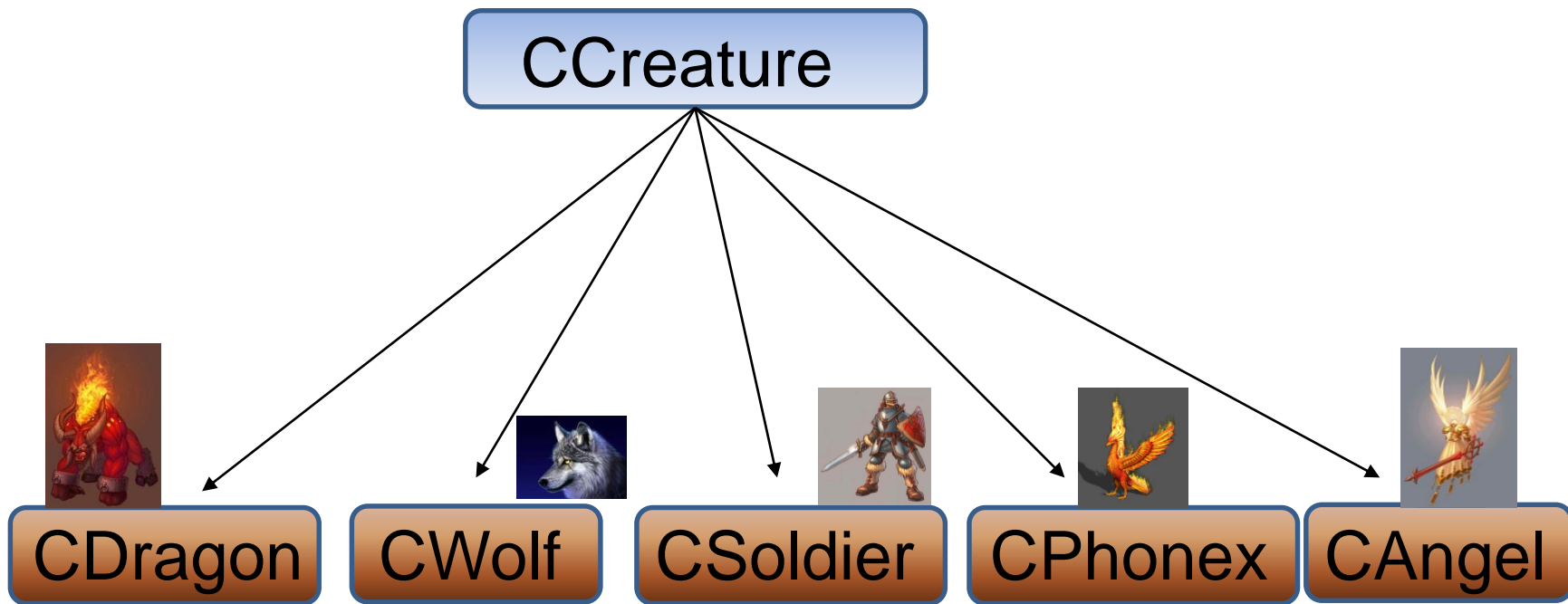
- 为每个怪物类编写 `Attack`、`FightBack`和 `Hurted`成员函数。
- `Attact`函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 `Hurted`函数，以减少被攻击怪物的生命值，同时也调用被攻击怪物的 `FightBack`成员函数，遭受被攻击怪物反击。
- `Hurted`函数减少自身生命值，并表现受伤动作。

## 基本思路：

- 为每个怪物类编写 **Attack**、**FightBack**和 **Hurted**成员函数。
- **Attact**函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的**Hurted**函数，以减少被攻击怪物的生命值，同时也调用被攻击怪物的 **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**成员函数表现反击动作，并调用被反击对象的**Hurted**成员函数，使被反击对象受伤。

## 基本思路：

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。





# 非多态的实现方法

```
class class C Creature {
    protected:          int nPower ; //代表攻击力
                        int nLifeValue ; //代表生命值
};
class C Dragon:public C Creature {
    public:
        void Attack(C Wolf * pWolf) {
            . . . 表现攻击动作的代码
            pWolf->Hurted( nPower);
            pWolf->FightBack( this);
        }
        void Attack( C Ghost * pGhost) {
            . . . 表现攻击动作的代码
            pGhost->Hurted( nPower);
            pGohst->FightBack( this);
        }
}
```

# 非多态的实现方法

```
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);  
}  
void FightBack( CGhost * pGhost) {  
    . . . . 表现反击动作的代码  
    pGhost->Hurted( nPower / 2 );  
}  
}
```

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

# 非多态的实现方法的缺点



- 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。

# 非多态的实现方法的缺点



- 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两个成员函数：

```
void Attack( CThunderBird * pThunderBird) ;  
void FightBack( CThunderBird * pThunderBird) ;
```

# 非多态的实现方法的缺点



- 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两个成员函数：

```
void Attack( CThunderBird * pThunderBird) ;  
void FightBack( CThunderBird * pThunderBird)
```

- 在怪物种类多的时候，工作量较大有木有！！！！

抓狂啦！



# 多态的实现方法

//基类 C Creature:

```
class C Creature {  
    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 C Creature 的派生类也是这样。

# 多态的实现方法

//派生类 CDragon:

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

# 多态的实现方法

```
void CDragon::Attack(CCreature * p)
{
    ...表现攻击动作的代码
    p->Hurted(m_nPower); //多态
    p->FightBack(this); //多态
}

void CDragon::Hurted( int nPower)
{
    ...表现受伤动作的代码
    m_nLifeValue -= nPower;
}

void CDragon::FightBack(CCreature * p)
{
    ...表现反击动作的代码
    p->Hurted(m_nPower/2); //多态
}
```



# 多态实现方法的优势



- 如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird……

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， 已有的类可以原封不动， 没压力啊！！！
```

# 多态实现方法的优势



- 如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird……

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， 已有的类可以原封不动， 没压力啊！！
```



# 原理

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

```
void CDragon::Attack(CCreature * p)
{
    p->Hurled(m_nPower); //多态
    p->FightBack(this); //多态
}
```



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜 刘家瑛

## 更多 多态程序实例



# 几何形体处理程序

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

# 几何形体处理程序

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



# 几何形体处理程序

## 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>
#include <math.h>
using namespace std;
class CShape
{
    public:
        virtual double Area() = 0; //纯虚函数
        virtual void PrintInfo() = 0;
};
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];  
int MyCompare(const void * s1, const void * s2);
```

```
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();  
                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();  
return 0;  
}
```

```
int MyCompare(const void * s1, const void * s2)
{
    double a1,a2;
    CShape * * p1 ; // s1,s2 是 void * , 不可写 “* s1” 来取得s1指向的内容
    CShape * * p2;
    p1 = ( CShape * * ) s1; //s1,s2指向pShapes数组中的元素，数组元素的类型是CShape *
    p2 = ( CShape * * ) s2; // 故 p1,p2都是指向指针的指针，类型为 CShape **
    a1 = (*p1)->Area(); // * p1 的类型是 Cshape * ,是基类指针，故此句为多态
    a2 = (*p2)->Area();
    if( a1 < a2 )
        return -1;
    else if ( a2 < a1 )
        return 1;
    else
        return 0;
}
```

```
    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();  
return 0;  
}
```



如果添加新的几何形体，比如五边形，则只需要从CShape派生出CPentagon, 以及在main中的switch语句中增加一个case，其余部分不变有木有！



用基类指针数组存放指向各种派生类对象的指针，然后遍历该数组，就能对各个派生类对象做各种操作，是很常用的做法

# 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;  
}
```

# 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()



# 多态的又一例子

```
class Base {
public:
    void fun1() { this->fun2(); } //this是基类指针, fun2是虚函数, 所以是多态
    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()

# 多态的又一例子

```
class Base {  
public:  
    void fun1() { this->fun2(); } //this是基类指针, fun2是虚函数, 所以是多态  
    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()



在非构造函数, 非析构函数的成员函数中调用虚函数, 是多态!!!

# 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，不是多态。编译时即可确定，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。

```
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(); };
};
```

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

😊 派生类中和基类中虚函数同名同参数表的函数，不加virtual也自动成为虚函数

```
class grandson:public son{    public:
    void hello(){cout<<"hello from grandson"<<endl;};
    void bye() { cout << "bye from grandson"<<endl; }
    grandson(){cout<<"constructing grandson"<<endl;};
    ~grandson(){cout<<"destructing grandson"<<endl;};
};
```

结果:  
hello from son  
constructing grandson  
hello from grandson  
destructing grandson  
bye from myclass

# 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
```

- 编译出错是因为 fun2() 是Base的私有成员。即使运行到此时实际上调用的应该是Derived的公有成员 fun2()也不行，因为语法检查是不考虑运行结果的。
- 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()。



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜 刘家瑛

## 多态的实现原理



## 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 ----- 这叫“**动态联编**”。“**动态联编**”底是怎么实现的呢？

## 提示：请看下面例子程序：

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

程序运行输出结果： 8, 12  
或： 12,16

(也可能是其他，有对齐问题)



## 提示：请看下面例子程序：

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

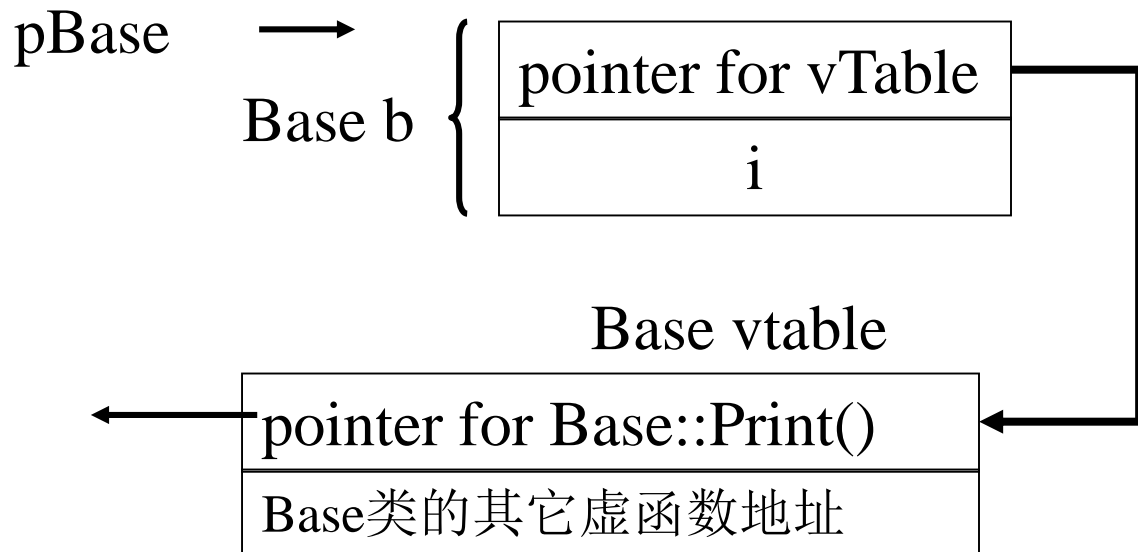
程序运行输出结果： 8, 12



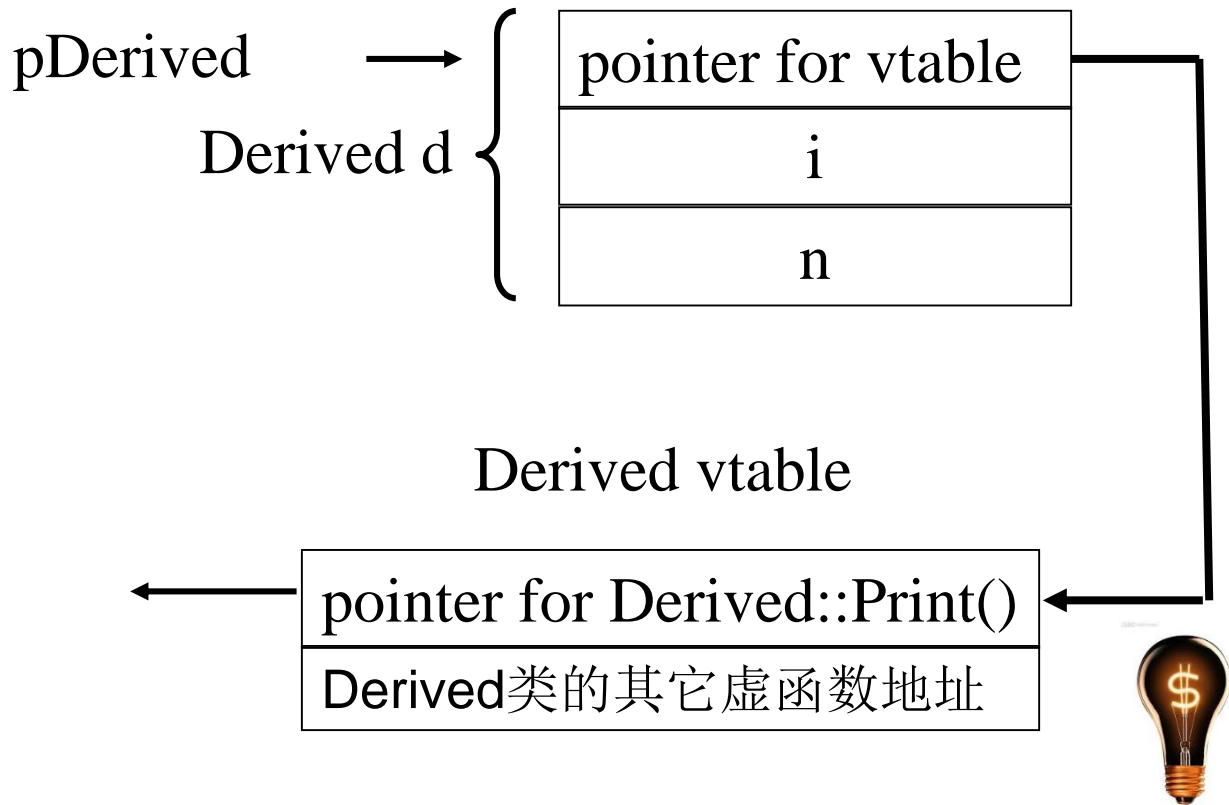
为什么都多了4个字节？

# 多态实现的关键 —— 虚函数表

每一个有虚函数的类（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的。**



# 多态实现的关键 —— 虚函数表



```
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，并调用虚函数的指令。

```
#include <iostream>
using namespace std;
class A    {
    public: virtual void Func() { cout << "A::Func" << endl; }
};
class B:public A {
    public: virtual void Func() { cout << "B::Func" << endl; }
};
int main() {
    A a;
    A * pa = new B();
    pa->Func();
    //64位程序指针为8字节
    long long * p1 = (long long * ) & a;
    long long * p2 = (long long * ) pa;
    * p2 = * p1;
    pa->Func();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class A {
    public: virtual void Func() { cout << "A::Func" << endl; }
};
class B:public A {
    public: virtual void Func() { cout << "B::Func" << endl; }
};
int main() {
    A a;
    A * pa = new B();
    pa->Func();
    //64位程序指针为8字节
    long long * p1 = (long long * ) & a;
    long long * p2 = (long long * ) pa;
    * p2 = * p1;
    pa->Func();
    return 0;
}
```

B::Func

```
#include <iostream>
using namespace std;
class A    {
    public: virtual void Func() { cout << "A::Func" << endl; }
};
class B:public A {
    public: virtual void Func() { cout << "B::Func" << endl; }
};
int main() {
    A a;
    A * pa = new B();
    pa->Func();
    //64位程序指针为8字节
    long long * p1 = (long long * ) & a;
    long long * p2 = (long long * ) pa;
    * p2 = * p1;
    pa->Func();
    return 0;
}
```

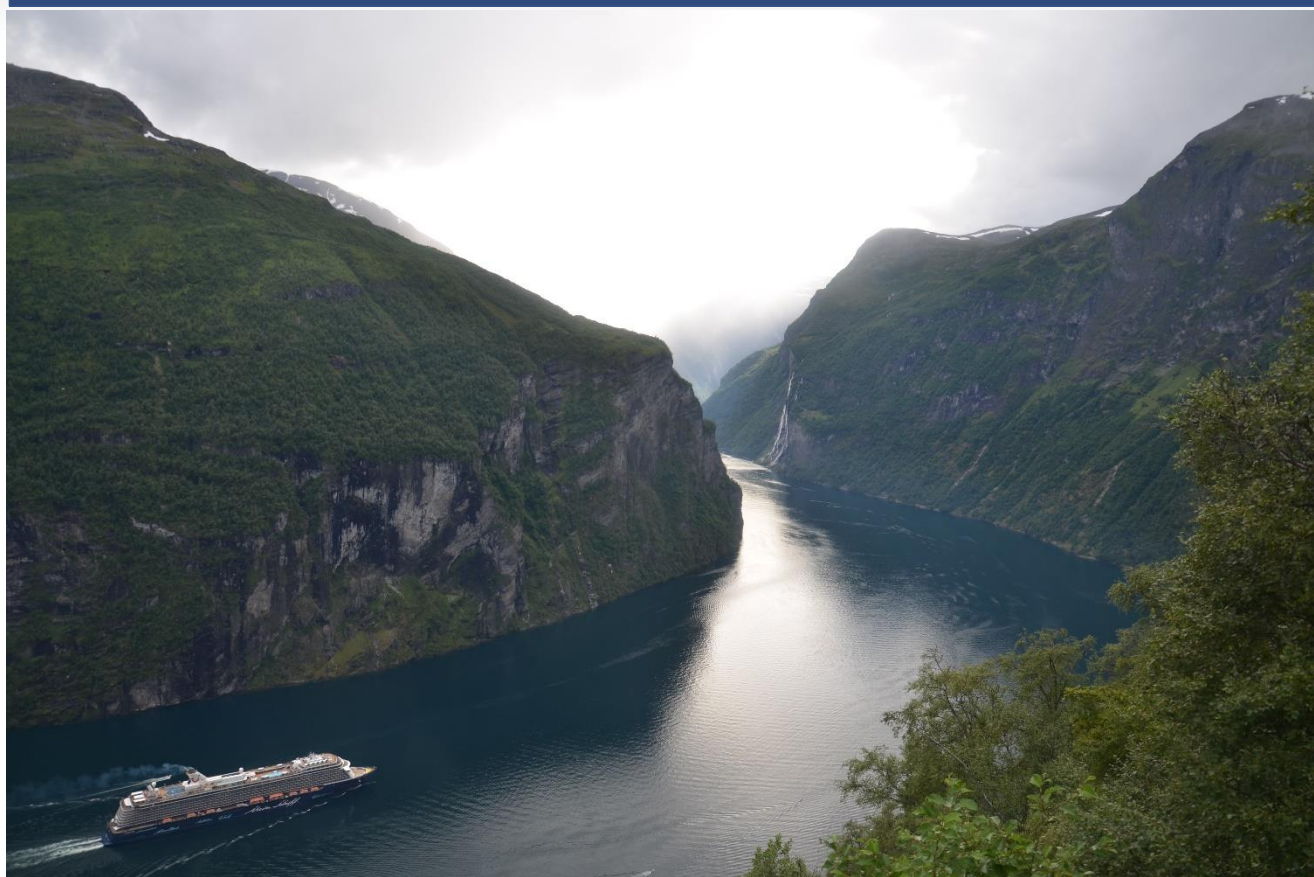
B::Func  
A::Func



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜

## 虚析构函数



挪威厄朗盖尔峡湾

# 虚析构函数

- 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。
- 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virtual不进行声明
  - 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。或者，一个类打算作为基类使用，也应该将析构函数定义成虚函数。
- 注意：不允许以虚函数作为构造函数



```
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: bye from son 没有执行grandson::~~grandson()!!!

```
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: bye from grandson  
 bye from son

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



北京大学  
PEKING UNIVERSITY

信息科学技术学院 郭炜

## 纯虚函数和抽象类



挪威大西洋之路

# 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"; }  
};
```

# 纯虚函数和抽象类

- 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, 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

# 纯虚函数和抽象类

- 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，不能创建独立的抽象类的对象
- 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, `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:  
B:f()