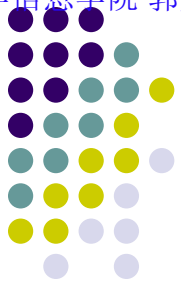
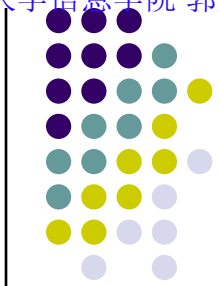


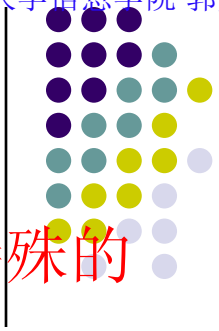
新标准C++程序设计

北京大学信息学院　郭 炜

GWPL@PKU.EDU.CN



多态



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:

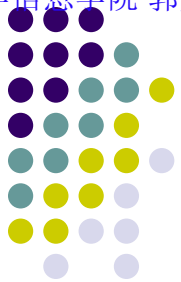
（1）若该指针指向一个基类的对象，那么被调用是基类的虚函数;

（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”。

➢例如:

```
CBase * p = &ODerived;
p -> SomeVirtualFunction();
```



➢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:

（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;

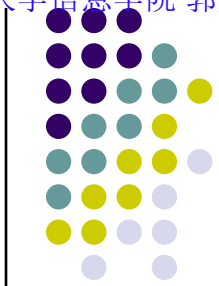
（2）如果引用的是一个派生类的对象，那么被调用的是派生类的虚函数。

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。



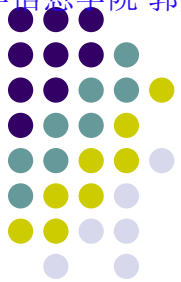
虚函数

➢在类的定义中，前面有 virtual 关键字的成员函数就是虚函数。

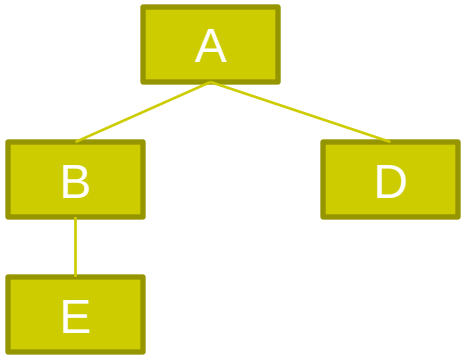
```
class base {
        virtual int get() ;
};
int base::get()
{}
```

➢virtual 关键字只用在类定义里的函数声明中，写函数体时不用。

多态的例子

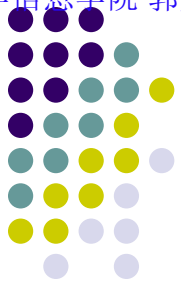


```
class A {
        public :
        virtual  void Print(  ) {   cout << "A::Print"<<endl ; }
};
class B: public A {
        public :
        virtual  void Print(  ) {   cout << "B::Print" <<endl; }
};
class D: public A {
        public:
        virtual  void Print(  ) {   cout << "D::Print" << endl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



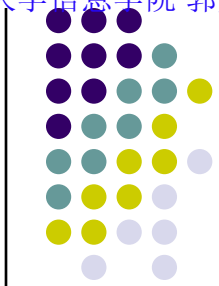

```
int main()  {
   A  a; B  b;     E e; D d;
   A * pa = &a;  B * pb = &b;
   D * pd = &d ; E * pe = &e;

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



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： CAngel

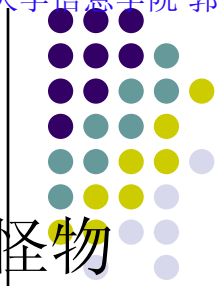
类CPhonex

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。

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



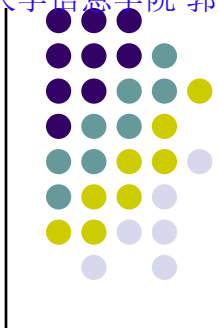
编程基本思路都是：

✓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。

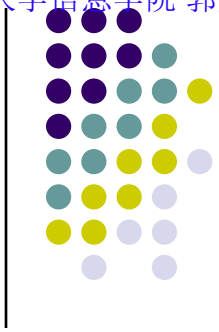
✓Hurted函数减少自身生命值，并表现受伤动作。

✓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。



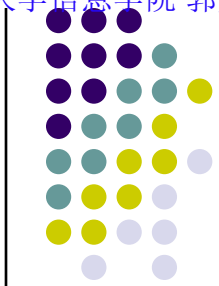
非多态的实现方法

```
class   CDragon  {
 private:
        int nPower ; //代表攻击力
        int nLifeValue ; //代表生命值
 public:
         void Attack( CWolf * pWolf) {
                . . . 表现攻击动作的代码
                pWolf->Hurted( nPower);
                pWolf->FightBack( this);
        }
        void Attack( CGhost * pGhost) {
                . . . 表现攻击动作的代码
                pGhost->Hurted( 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 ->Hurt( nPower / 2);
        }
        void FightBack( CGhost * pGhost) {
                . . . . 表现反击动作的代码
                pGhost->Hurt( 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。



- 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:

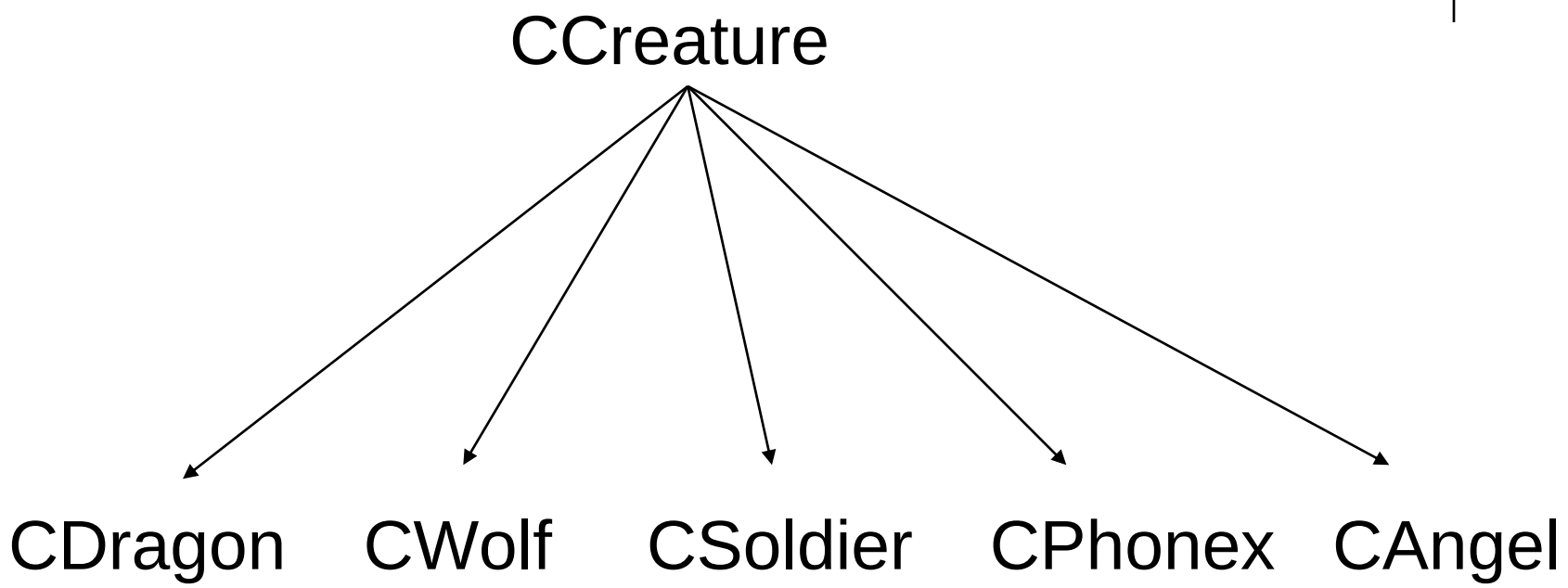
```
void Attack( CThunderBird * pThunderBird) ;
void FightBack( CThunderBird * pThunderBird) ;
```

- 在怪物种类多的时候，工作量较大。

- 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，形成一个CCreature类，然后再从CCreature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CCreature，对程序的可扩充性也无帮助。

使用多态的实现方法

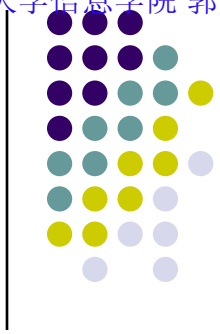
- 设置基类 CCreature，并且使CDragon, CWolf等其他类都从CCreature派生而来。



基类 CCreature：

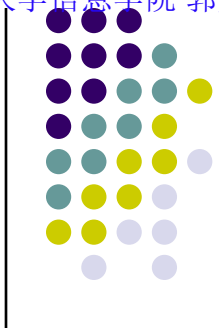
```
class CCreature  {
        protected :
                int m_nLifeValue, m_nPower;
        public:
          virtual void Attack( CCreature * pCreature) {}
          virtual void  Hurted( int nPower) { }
          virtual void FightBack( CCreature * pCreature) { };
};
```

- 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。

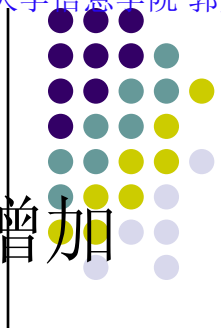


派生类　CDragon:

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



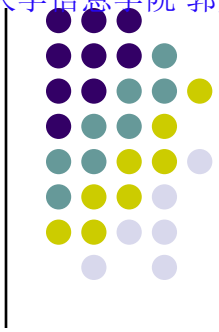
```
void CDragon::Attack(CCreature * p)
{
        p->Hurted(m_nPower);
    p->FightBack(this);
}
void CDragon::Hurted( int nPower)
{
        m_nLifeValue -= nPower;
}
void CDragon::FightBack(CCreature * p)
{
        p->Hurted(m_nPower/2);
}
```



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::Hurted

CGhost::Hurted

CBird::Hurted

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。

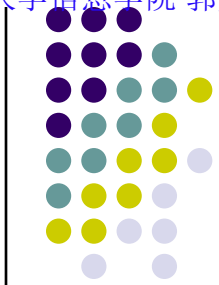
Input:

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



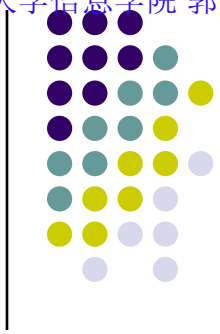
使用多态的另一例子：

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



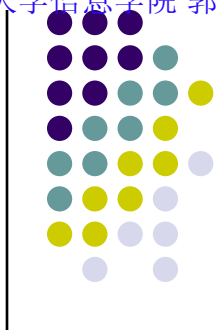
使用多态的另一例子：

Sample Input:

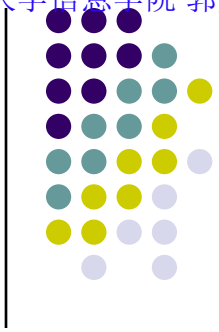
3
R 3 5
C 9
T 3 4 5

Sample Output

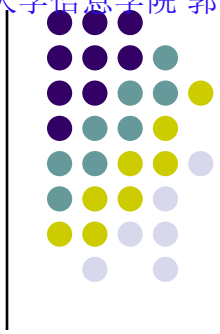
Triangle:6
Rectangle:15
Circle:254.34



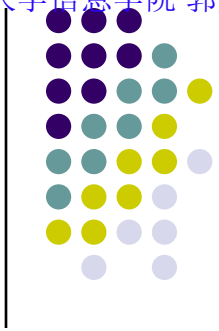
```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;
class CShape
{
    public:
        virtual double Area() = 0;
        virtual void PrintInfo() = 0;
};
class CRectangle:public CShape
{
    public:
        int w,h;
        virtual double Area();
        virtual void PrintInfo();
};
```



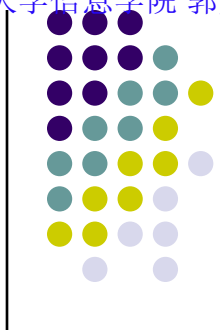
```
class CCircle:public CShape
{
    public:
        int r;
        virtual double Area();
        virtual void PrintInfo();
};
class CTriangle:public CShape
{
    public:
        int a,b,c;
        virtual double Area();
        virtual void PrintInfo();
};
```



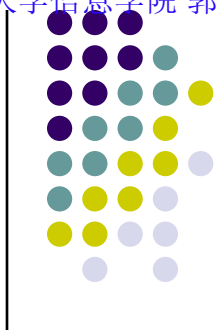
```
double CRectangle::Area()  {
    return w * h;
}
void CRectangle::PrintInfo()  {
    cout << "Rectangle:" << Area() << endl;
}
double CCircle::Area()  {
    return 3.14 * r * r ;
}
void CCircle::PrintInfo()  {
    cout << "Circle:" << Area() << endl;
}
double CTriangle::Area()  {
    double p = (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



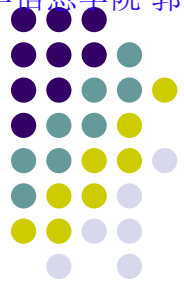
```
CShape * pShapes[100];
int MyCompare(const void * s1, const void * s2)
{
    double a1,a2;
    CShape * * p1 ;
    CShape * * p2;
    p1 = ( CShape * * ) s1;
    p2 = ( CShape * * ) s2;
    a1 = (*p1)->Area();
    a2 = (*p2)->Area();
    if( a1 < a2 )
        return -1;
    else if ( a2 < a1 )
        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();
                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;
}
```



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。

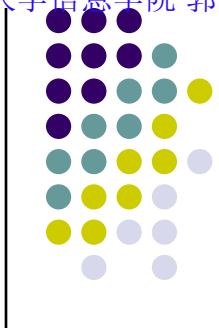
为什么需要动态联编？

```
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。

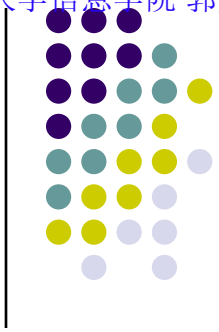
➢所以只能等程序运行到 pa->Get()了，才能决定到底调用哪个Get()。



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

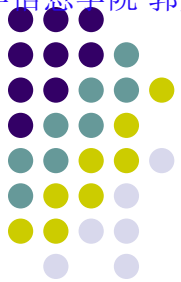
```
void fun1()  {
        fun2();
}
```

相当于

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

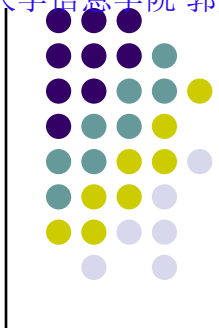
➢编译这个函数的代码的时候，由于fun2()是虚函数，this是基类指针，所以是动态联编。

➢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()。



思考

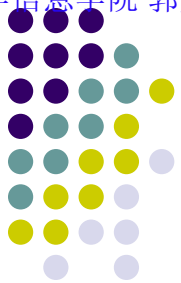
“多态”的关键在于通过基类指针或引用调用一个虚函数时，编译时不确定到底调用的是基类还是派生类的函数，运行时才确定。这到底是怎么实现的呢？



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;
}
```

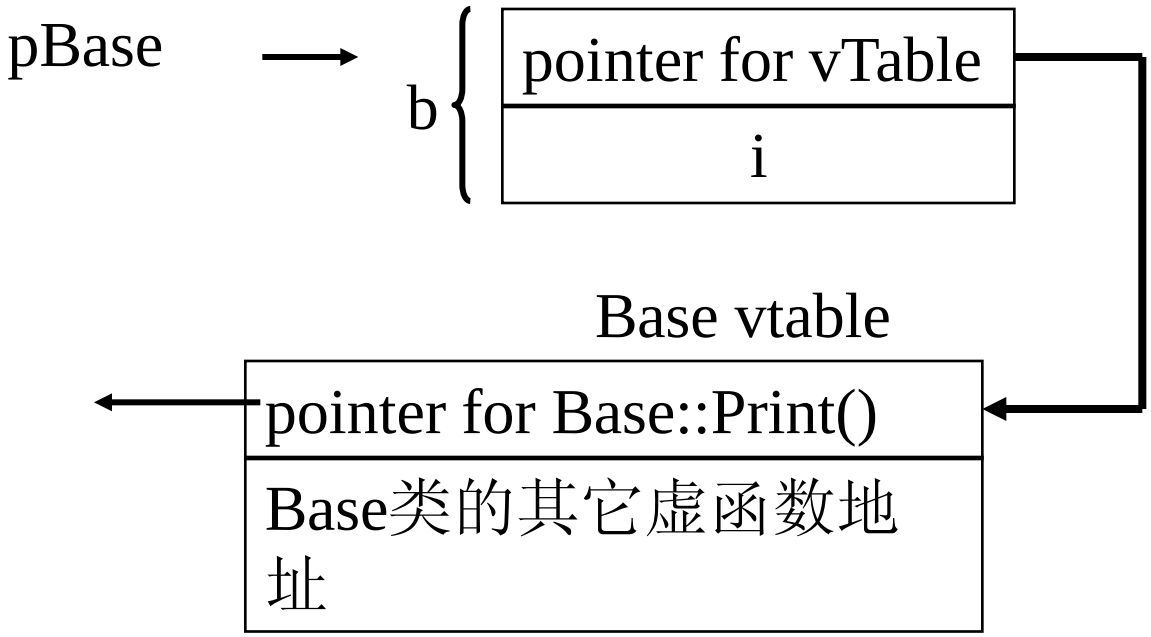
程序运行输出结果： 8, 12

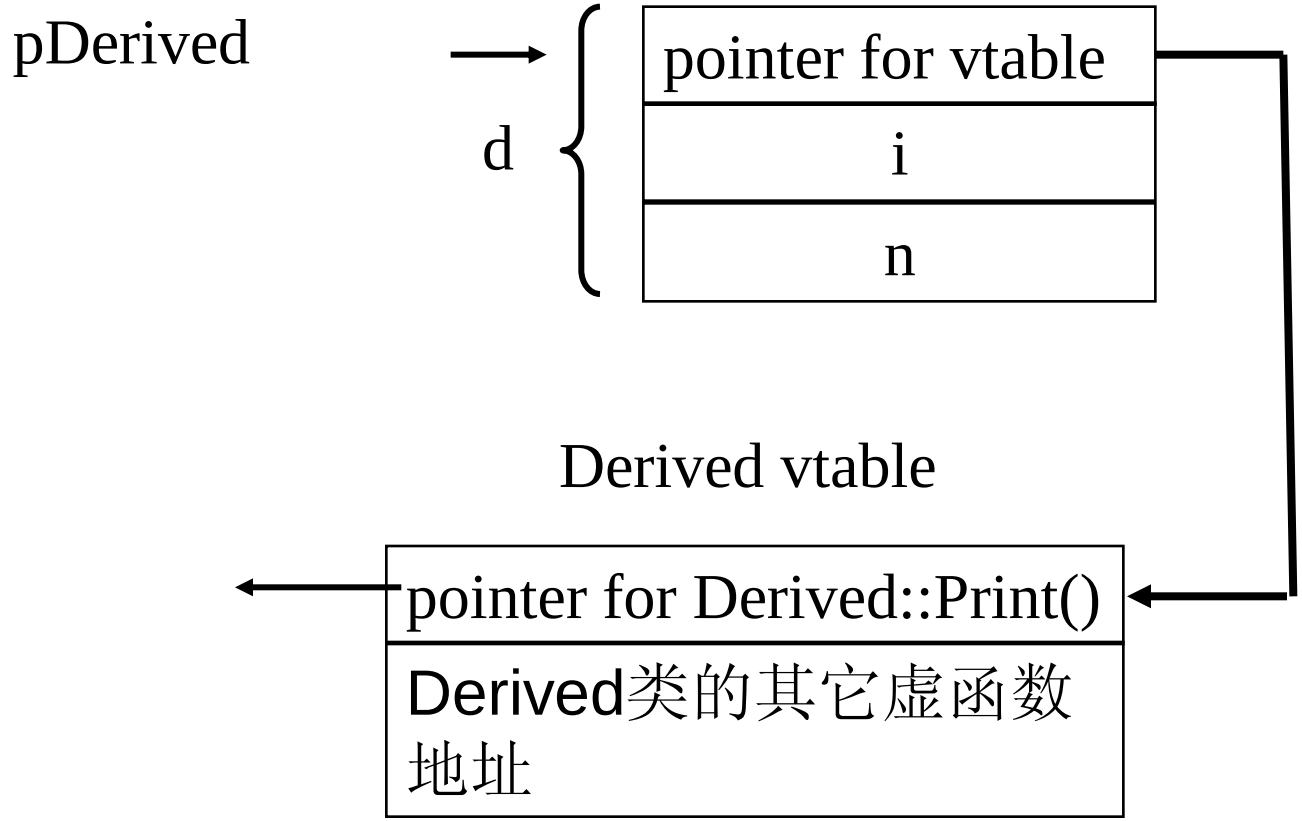


为什么 Base 对象的大小是 8 个字节而不是 4 个字节，为什么 Derived 对象的大小是12个字节而不是8个字节，多出来的 4 个字节做什么用呢？和多态的实现有什么关系？

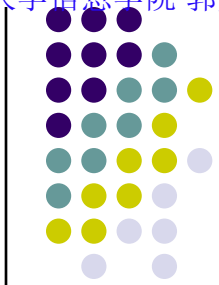
多态的实现：虚函数表

➢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。






```
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
```

- 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()。

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：他们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

- 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。
- 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;
 }
```

输出结果： bye from son

没有执行grandson::~grandson()!!!

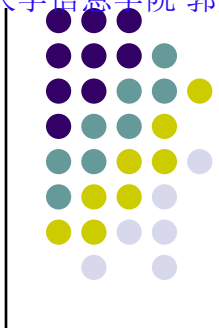
```
class son{
public:
    virtual ~son() {cout<<"bye from son"<<endl;};
};
class grandson:public son{
public:
    ~grandson(){cout<<"bye from grandson"<<endl;};
};
int main() {
    son *pson;
    pson= new grandson();
    delete pson;
    return 0;
 }
```

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



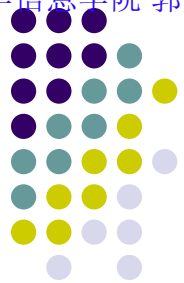
- 纯虚函数： 没有函数体的虚函数

```
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，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()