



# C++程序设计精要教程

华中科技大学

# 第8章 虚函数与多态

## ◆8.1 虚函数

- 虚函数：即用virtual定义的成员函数。Java几乎所有函数都默认为虚函数。当基类对象指针或引用指向或引用不同类型派生类对象时，通过虚函数到基类或派生类中同名函数的映射实现(动态)多态。
- 动态多态：重载函数表现的是静态(编译时)多态性，虚函数表现的是动态(运行时)多态性：
- 重载函数是静态多态函数，通过静态绑定调用重载函数；虚函数是动态多态函数，通过动态绑定调用(虚映射到实)函数。动态绑定是程序运行时自己完成的，静态绑定是编译或操作系统完成的。
- 虚函数的动态绑定通过存储在对象中的一个指针完成，因此虚函数一定有this(指向这个对象)。(该指针指向虚函数入口地址表VFT)

# 第8章 虚函数与多态

【例8.1】 定义父类POINT2D和子类CIRCLE的绘图函数成员show()

```
#include <iostream>
using namespace std;
class POINT2D{
    int x, y;
public:
    int getx( ) { return x; }
    int gety( ) { return y; }
    virtual POINT2D* show( ){ cout<<"Show a point\n"; return this;} //定义虚函数
    POINT2D(int x, int y) { POINT2D::x=x; POINT2D::y=y; }
};
class CIRCLE: public POINT2D{ //POINT2D和CIRCE满足父子关系
    int r;
```

# 第8章 虚函数与多态

public:

```
int getr( ) { return r; }
```

```
CIRCLE* show( ) { cout<<"Show a circle\n"; return this;}//原型 “相容” ， 自动成为虚函数
```

```
CIRCLE(int x, int y, int r):POINT2D(x, y) { CIRCLE::r=r; }
```

```
};
```

```
void main(void)
```

```
{
```

```
    CIRCLE c(3, 7, 8);
```

```
    POINT2D *p=&c;
```

//父类指针p可以直接指向子类对象c

```
    cout<<"The circle with radius "<<c.getr( );
```

```
    cout<<" is at ("<<p->getx( )<<", "<<p->gety( )<<")\n";
```

```
    p->show( );
```

```
}
```



# 第8章 虚函数与多态

## ◆8.1 虚函数

- 虚函数必须是类的成员函数，非成员函数不能说明为虚函数，普通函数如main不能说明为虚函数（与编译器有关）。
- 虚函数一般在基类的public或protected部分。在派生类中重新定义成员函数时，函数原型必须完全相同；
- 虚函数只有在具有继承关系的类层次结构中定义才有意义，否则引起额外开销（需要通过VFT访问）；
- 一般用父类指针(或引用)访问虚函数。根据父类指针所指对象类型的不同，动态绑定相应对象的虚函数；(虚函数的动态多态性)

# 第8章 虚函数与多态

## ◆8.1 虚函数

- 虚函数有隐含的this参数，参数表后可出现const和volatile，静态函数成员没有this参数，不能定义为虚函数：即不能有virtual static之类的说明；
- 构造函数构造对象的类型是确定的，不需根据类型表现出多态性，故不能定义为虚函数；析构函数可通过父类指针(引用)或delete调用，父类指针指向的对象类型可能是不确定的，因此析构函数可定义为虚函数。
- 一旦父类(基类)定义了虚函数，即使没有“virtual”声明，所有派生类中原型相同的非静态成员函数自动成为虚函数；(虚函数特性的无限传递性)

# 第8章 虚函数与多态

## ◆8.1 虚函数

- 虚函数同普通函数成员一样，可声明为或自动成为inline函数，也可重载、缺省和省略参数。
- 虚函数能根据对象类型适当地绑定函数成员，且绑定函数成员的效率非常高，因此，最好将普通函数成员全部定义为虚函数。
- 注意：虚函数主要通过基类和派生类表现出多态特性，由于union既不能定义基类又不能定义派生类，故不能在union中定义虚函数。

# 第8章 虚函数与多态

```
#include <iostream>                                     // 【例8.2】 虚函数的使用方法
using namespace std;
struct A{
    virtual void f1( ){ cout<<"A::f1\n"; }; //定义虚函数f1()
    virtual void f2( ){ cout<<"A::f2\n"; }; //this指向基类对象，定义虚函数f2()
    virtual void f3( ){ cout<<"A::f3\n"; }; //定义虚函数f3()
    virtual void f4( ){ cout<<"A::f4\n"; }; //定义虚函数f4()
};
class B: public A{ //A和B满足父子关系
    virtual void f1( ){//virtual可省略，f1()自动成为虚函数
        cout<<"B::f1\n";
    };
    void f2( ) {          //除this指向派生类对象外，f2()和基类函数原型相同，自动成为虚函数
        cout<<"B::f2\n";
    };
};
```



# 第8章 虚函数与多态

```
class C: B{           //B和C不满足父子关系，故A和C也不满足父子关系
    void f4( ) {       //f4()自动成为虚函数
        cout<<"C::f4\n";
    };
};
void main(void)
{
    C c;
    A *p=(A *)&c;      //A和C不满足父子关系，需要进行强制类型转换
    p->f1( );           //调用B::f1( )
    p->f2( );           //调用B::f2( )
    p->f3( );           //调用A::f3( )
    p->f4( );           //调用C::f4( )
    p->A::f2( );        //明确调用实函数A::f2( )
}
```

# 第8章 虚函数与多态

## ◆8.1 虚函数

- 重载函数使用静态联编（早期绑定）机制；虚函数采用动态联编（晚期绑定）机制；
- 早期绑定：在程序运行之前的绑定；晚期绑定：在程序运行中，由程序自己完成的绑定。
- 对于父类A中声明的虚函数f()，若在子类B中重定义f()，必须确保子类B::f()与父类A::f()具有完全相同的函数原型，才能覆盖原虚函数f()而产生虚特性，执行动态联编机制。否则，只要有一个参数不同，编译系统就认为它是一个全新的（函数名相同时重载）函数，而不实现动态联编。

# 第8章 虚函数与多态

## ◆8.2 虚析构函数

- 如果基类的析构函数定义为虚析构函数，则派生类的析构函数就会自动成为虚析构函数（原型不同）。
- 说明虚析构函数的目的在于在使用delete运算符释放一个对象时，能够保证所执行的析构函数就是该对象的析构函数；最好将所有的析构函数都定义为虚析构函数。注意，对象数组指针p应用delete []p释放。
- 注意：如果为基类和派生类的对象分配了动态内存，或者为派生类的对象成员分配了动态内存，则一定要将基类和派生类的析构函数定义为虚析构函数，否则便可能造成内存泄漏，导致系统出现内存保护错误。

# 第8章 虚函数与多态

【例8.5】输入职员的花名册,如果职员的姓名、编号和年龄等信息齐全,则登记该职员的个人信息,否则只登记职员的姓名。

```
#include <stdio.h>
#include <string.h>
class STRING{
    char *str;
public:
    STRING(char *s);
    virtual ~STRING(){
        if (str){delete str; str=0; }
    }
};
STRING::STRING(char *s){
    str=new char[strlen(s)+1];
```

```
    strcpy(str, s);
}
class CLERK: public STRING{
    STRING clkid;   int age;
public:
    CLERK(char *n, char *i, int a);
    ~CLERK(){} //自动成为虚函数
    //自动调用clkid.~STRING()
    //和STRING::~~STRING()
};
CLERK::CLERK(char *n, char *i,
    int a):STRING(n), clkid(i){
    age=a;
}
```



# 第8章 虚函数与多态

```
const int max=10;
void main(void){
    STRING *s[max]; //花名册
    int a, k, m;  char n[12], i[12], t[256];
    printf("Please input name, number and age:\n");
    for(k=0; k<max; k++) {
        gets(t);
        m=sscanf(t,"%8s %8s %d",n,i,&a)!=3; //m记录信息是否齐全
        s[k]=m?new STRING(n):new CLERK(n,i,a);
    }
    for(k=0; k<max; k++)
        delete s[k]; //多态调用析构函数
    //若s[k]指向STRING的对象, 则用s[k]->~STRING()析构
    //若s[k]指向CLERK的对象, 则用s[k]->~CLERK()析构
}
```

# 第8章 虚函数与多态

## ◆8.3 类的引用

- 用父类引用实现动态多态性时需要注意，若被(new产生)引用对象自身不能析构，则必须用delete &析构：

```
STRING &z=*new CLERK("zang","982021",23);  
delete &z; //析构对象z并释放对象z占用的内存
```

- 上述delete &z完成了两个任务：①调用该对象析构函数~CLERK()，释放其基类和对象成员各自为字符指针str分配的空间；②释放CLERK对象自身占用的存储空间。
- 如将delete &z改为z.~STRING()，则只完成任务①而没完成②；如果改为free(&z)，则只完成任务②而没完成①。造成内存泄露。为什么z.~STRING()执行~CLERK()？(z实现为指针)

# 第8章 虚函数与多态

## ◆8.3 类的引用

- 引用变量引用类的变量、函数参数或者常量，一般不需要引用变量负责构造和析构。由被引用的类的变量、参数或常量自动完成析构。
- 当用常量对象、类型为&&的返回对象作为实参调用函数时，优先调用的函数是带有&&参数的函数。
- 常量对象既可以被有址变量引用（分配对象内存），也可以被无址变量引用（分配对象缓存），但优先被无址形参引用。

# 第8章 虚函数与多态

【例8.7】应用delete析构有址引用变量引用的通过new生成的对象

```
#include <iostream>
using namespace std;
class A{
    int i;
public:
    A(int i) { A::i=i; cout<<"A: i="<<i<<"\n"; };
    ~A( ) { if(i) cout<<"~A: i="<<i<<"\n"; i=0; };
};
void g(A &a) {cout<<"g is running\n"; }           //调用时初始化形参a
void h(A &&a=A(5)) {cout<<"h is running\n"; }//调用时初始化形参a, A(5)为默认值
```



# 第8章 虚函数与多态

```
void main(void)
```

```
{
```

```
    A a(1), b(2);
```

```
    A &p=a;
```

```
    A &q=*new A(3);
```

```
    A &r=p;
```

```
    cout<<"CALL g(b)\n";
```

```
    g(b);
```

```
    h( );
```

```
    h(A(4));
```

```
    cout<<"main return\n";
```

```
    delete &q;
```

```
}
```

```
//自动调用构造函数构造a、b
```

```
//p本身不用负责构造和析构a
```

```
//q有址引用new生成的无名对象
```

```
//r有址引用p所引用的对象a
```

```
//使用同类型的传统左值作为实参调用函数g()
```

```
//使用无址右值A(5)作为实参调用h(), 初始化h()的形参a
```

```
//使用无址右值A(4)作为实参调用h(), 初始化h()的形参a
```

```
//q析构并释放通过new产生的对象A(3)
```

```
//退出main()时依次自动析构b、a
```

# 第8章 虚函数与多态

## ◆8.3 类的引用

- 当类的内部包含指针成员时，为了防止内存泄漏，不应使用编译自动生成的构造函数、赋值运算符函数和析构函数。
- 对于类型为A且内部有指针的类，应自定义A()、A(A&&) noexcept、A(const A&)、A& operator=(const A&)、A& operator=(A&&) noexcept以及~A()函数。
- A(A&&)、A& operator=(A&&)通常应按移动语义实现，构造和赋值分别是浅拷贝移动构造和浅拷贝移动赋值。“移动”即将一个对象（通常是常量）内部的(分配内存的) 指针成员浅拷贝赋给新对象的内部指针成员，而前者的内部指针成员设置为空指针（即内存被移走了）。
- 对于A的派生类B，在构造和赋值以基类A相关的对象时，若B类参数为&&，则应对用A类参数为&&的拷贝和赋值运算函数。

# 第8章 虚函数与多态

```
class A {  
    int* p;  
    int m;  
public:  
    A(): p(nullptr), m(0) {}  
    A(int m): p(new int[m]), m(p?m:0){ }  
    A(const A&a): p(new int[a.m]), m(p ? a.m : 0) { //深拷贝构造必须为p重新分配内存  
        for (int x = 0; x < m; x++) p[x] = a.p[x];  
    }  
    A(A&& a) noexcept: p(a.p), m(a.m) { //移动拷贝构造不要为p重新分配内存  
        a.p = nullptr; a.m = 0;  
    }  
    ~A() {  
        if (p){ delete p; p = nullptr; m = 0; }  
    };  
};
```

# 第8章 虚函数与多态

A& operator=(const A&a) { //浅拷贝移动构造不为e重新分配内存

if (&a == this) return \*this;

if (p) delete p;

p = new int[a.m];

m = p ? a.m: 0;

for (int x = 0; x < m; x++) p[x] = a.p[x];

return \*this;

}

A& operator=(A&& a) noexcept { //浅拷贝移动构造不为e重新分配内存

if (&a == this) return \*this;

if (p) delete p;

p = a.p; m = p ? a.m : 0; //移动语义：资源a.p转移

a.p = nullptr; a.m = 0; //移动语义：资源a.p已经转移，故资源数量设为 0

return \*this;

}

};

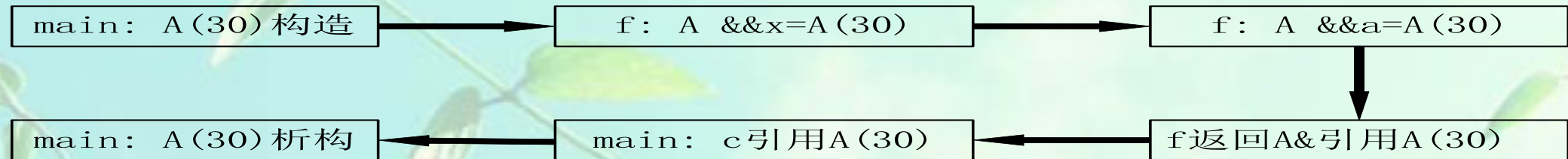


# 第8章 虚函数与多态

```
A &f(A&& x) {  
    //A &&a= static_cast<A&&>(x); //a引用x所引用的对象;  
    //return a; //返回A& : 参数有名有址, 类型&&自动转换为&。x和a都不负责析构  
    return x; //结果同上述两条语句  
}
```

```
void main() { A &c=f(A(30)); /* c引用A(30) */ }
```

在函数f中，移动构造或赋值新变量，不用反复释放和申请内存，提高了程序执行效率



# 第8章 虚函数与多态

## ◆8.4 抽象类

- 纯虚函数：不必定义函数体的虚函数，也可以重载、缺省参数、省略参数、内联等，相当于Java的interface。
- 定义格式：virtual 函数原型=0。（0即函数体为空）
- 纯虚函数有this，不能同时用static定义(表示无this)。
- 构造函数不能定义为虚函数，同样也不能定义为纯虚函数。
- 析构函数可以定义为虚函数，也可定义为纯虚函数。
- 函数体定义应在派生类中实现，成为非纯虚函数。

# 第8章 虚函数与多态

## ◆8.4 抽象类

- 抽象类：含有纯虚函数的类。
- 抽象类常用作派生类的基类，不应该有对象或类实例(相当于Java的interface)。
- 如果派生类继承了抽象类的纯虚函数，却没有在派生类中重新定义该原型虚函数，或者派生类定义了基类所没有的纯虚函数，则派生类就会自动成为抽象类。
- 在多级派生的过程中，如果到某个派生类为止，所有纯虚函数都已在派生类中全部重新定义，则该派生类就会成为非抽象类（具体类）。

# 第8章 虚函数与多态

## 【例8.10】 多级派生中的抽象类与非抽象类用法

```
#include <iostream>
using namespace std;
struct A{ //A被定义为抽象类
    virtual void f1( )=0;
    virtual void f2( )=0;
};
void A::f1( ){ cout<<"A1"; }//不是在派
void A::f2( ){ cout<<"A2"; }//生类中定义
class B: public A{
    //重新定义f2，未定义f1，B为抽象类
    void f2( ) { this->A::f2( ); cout<<"B2"; }
};
class C: public B{// f1和f2均重定义，C为具体类
    void f1( ) { cout<<"C1"; }//自动成虚函数，但内联失败
};
```

```
void main(void)
{
    C c;
    A *p=(A *)&c;//p=&c
    p->f1( );//调用C::f1( )
    p->f2( );//调用B::f2( )
}
```



# 第8章 虚函数与多态

## ◆8.4 抽象类

- 抽象类不能定义或产生任何对象，包括用new创建的对象，故不能用作函数参数的类型和函数的返回类型（调用前后要产生该类型的对象）。
- 抽象类可作派生类的基类(父类)，若定义相应的基类引用和指针，就可引用或指向非抽象派生类对象。
- 通过抽象类指针或引用可调用抽象类的纯虚函数，根据多态性，实际调用的应是该类的非抽象派生类的虚函数。如果该派生类没有重新定义被调虚函数，则会导致程序出现不可意料的运行错误。调用抽象类的普通函数成员不会出现该问题。

# 第8章 虚函数与多态

## ◆8.4 抽象类

- 抽象类：含有纯虚函数的类。
- 抽象类常用作派生类的基类，不应该有对象或类实例(相当于Java的interface)。
- 如果派生类继承了抽象类的纯虚函数，却没有在派生类中重新定义该原型虚函数，或者派生类定义了基类所没有的纯虚函数，则派生类就会自动成为抽象类。
- 在多级派生的过程中，如果到某个派生类为止，所有纯虚函数都已在派生类中全部重新定义，则该派生类就会成为非抽象类（具体类）。

# 第8章 虚函数与多态

【例8.11】 本例说明抽象类不能产生对象

```
#include <iostream>
using namespace std;
struct A{
//定义类A为抽象类
    virtual void f1( )=0;
    void f2( ) { };
};
struct B: A{
//定义A的非抽象子类B
    void f1( ){ };
};
A f( ); //×, 返回类A意味着抽
        //象类要产生A类对象
int g(A x); // ×, 调用时要传递
        //一个A类的对象
```

```
A &h(A &y); //√, 可以引用非
        //抽象子类B的对象
void main(void)
{
    A a;    //×, 抽象类不能产生
            //对象a
    A *p;   //√, 可以指向非抽象
            //子类B的对象
    p->f1( ); //×, 运行时无A::f1( )
            //如p指向B类对象则正确
    p->f2( ); //√, 调用A::f2( )
}
```

# 第8章 虚函数与多态

## ◆8.4 抽象类

- 内存管理函数malloc可以为抽象类分配空间，但不调用构造函数，因此，内存管理函数malloc实质上不产生抽象类对象(VFT没有填好)。只有成功地构造了某个类的对象，才能通过抽象类指针或引用访问(VFT),进而通过VFT调用这个类的虚函数。
- 抽象类作为抽象级别最高的类，主要用于定义派生类共有的数据和函数成员。抽象类的纯虚函数没有函数体，意味目前尚无法描述该函数的功能。例如，如果图形是点、线和圆等类的抽象类，那么抽象类的绘图函数就无法绘出具体的图形。



# 第8章 虚函数与多态

## ◆8.4 抽象类

- 纯虚函数和虚函数都能定义成另一个类的成员友元。由于纯虚函数一般不会定义函数体，故纯虚函数一般不要定义为其其他类的成员友元。
- 如果类A的函数成员f定义为类B的友元，那么f就可以访问类B的所有成员，但是，f并不能访问从类B派生的类C的所有成员，除非f也定义为类C的友元或者类A就是类C。（即友元对派生不具备传递性）

# 第8章 虚函数与多态

【例8.13】 说明纯虚函数和虚函数定义为友元的用法

```
#include <iostream>
using namespace std;
class C;
struct A {
    virtual void f1(C &c)=0;
    virtual void f2(C &c);
};
class B: A{
public:
    void f1(C &c); //f1自动成虚函数
};
class C {
    char c;
    //允许但无意义, A::f1无函数体
    friend void A::f1(C &c);
    friend void A::f2(C &c);
```

```
public:
    C(char c) { C::c=c; }
};
void A::f1(C &c)
{ cout<<"B outputs "<<c.c<<"\n"; }
void A::f2(C &c)
{ cout<<"A outputs "<<c.c<<"\n"; }
void B::f1(C &c)
{ cout<<c.c; } //×, B::f1不是C的
               //友元, 不能访问c.c
void main( void){
    B b;   C c('C');
    A *p=(A *) new B;
    p->f1(c);      //调用B::f1()
    p->f2(c);      //调用A::f2()
}
```

# 第8章 虚函数与多态

## ◆8.5 虚函数友元与晚期绑定

### ●虚函数动态绑定：

- C++使用虚函数地址表(VFT)来实现虚函数的动态绑定。VFT是一个函数指针列表，存放对象的所有虚函数的入口地址。
- 编译程序为有虚函数的类创建一个VFT，其首地址通常存放在对象的起始单元中。调用虚函数的对象通过起始单元□VFT动态绑定相应的函数成员，从而使虚函数随调用对象的不同而表现多态特性。
- 动态绑定比静态绑定多一次地址访问，在一定程度上降低了程序的执行效率，但同虚函数的多态特性带来的优点相比，效率降低所产生的影响是微不足道的。

# 第8章 虚函数与多态

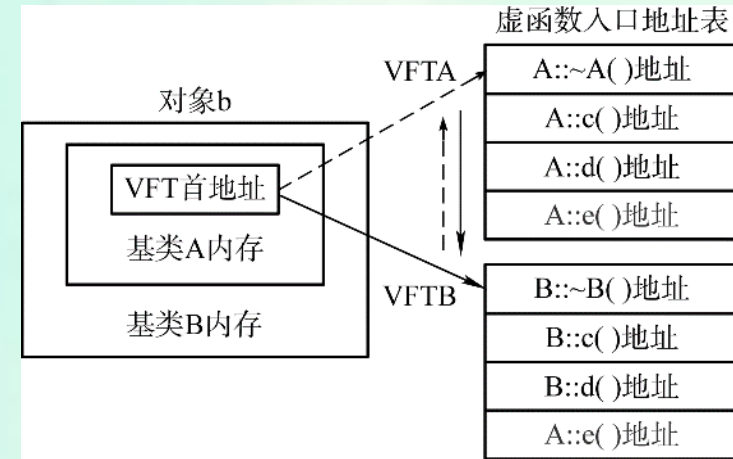
## ◆8.5 虚函数友元与晚期绑定

- **虚函数动态绑定过程**：设基类A和派生类B对应的虚函数表分别为VFTA和VFTB。则派生类对象b的虚函数动态绑定过程如下：
  - **对象构造**：先将VFTA的首地址存放到b的起始单元，在A类构造函数的函数体执行前甚至初试化前，使A类对象调用的虚函数与VFTA绑定，可使A类构造函数执行A的虚函数；在B类构造函数的函数体执行前（甚至初试化前），将VFTB的首地址存放到b的起始单元，使B类对象调用的虚函数与VFTB绑定，可使B类构造函数执行B的虚函数。
  - **对象使用(生成期间)**：b的起始单元指向VFTB，执行B的虚函数。
  - **对象析构**：由于b的起始单元已指向VFTB，故析构函数调用的是B的虚函数；然后将VFTA的首地址存放到b的起始单元，使基类析构函数调用的虚函数与VFTA绑定，使基类析构函数调用基类A的虚函数。



# 第8章 虚函数与多态

```
#include <iostream> //例8.14
using namespace std;
class A {
    virtual void c() { cout<<"Construct A\n"; }
    virtual void d() { cout<<"Deconstruct A\n"; }
    virtual void e() { };
public:
    A() { c(); };
    virtual ~A() { d(); };
};
class B: A {
    virtual void c() { cout<<"Construct B\n"; }
    virtual void d() { cout<<"Deconstruct B\n"; }
public:
    B() { c(); }; //等价于B(): A() { c(); };
    virtual ~B() { d(); };
};
```



```
void main(void){ B b; }
```

输出结果：

**Construct A**

**Construct B**

**Deconstruct B**

**Deconstruct A**

# 第8章 虚函数与多态

## ◆8.6 有虚函数时的内存布局

- 派生类的存储空间由基类和派生类的非静态数据成员构成。当基类或派生类包含虚函数或纯虚函数时，派生类的存储空间还包括虚函数入口地址表首址所占存储单元。
- 如果基类定义了虚函数或者纯虚函数，则派生类对象将其起始单元作为共享单元，用于存放基类和派生类的虚函数地址表首址。【例7.10】
- 如果基类没有定义虚函数，而派生类定义了虚函数，则派生类的存储空间由三部分组成：第一部分为基类存储空间，第二部分为派生类虚函数入口地址表首址，第三部分为该派生类新定义的数据成员