

# 第5章 多态

- 本章主要教学内容

- 面向对象语言的多态语言机制，多态的功能及其与软件维护的关系
- 静态联编、动态联编及其与多态的关系
- 虚函数、虚析构函数、**override**和**final**
- 纯虚函数、抽象类和抽象类作接口的程序设计
- RTTI机制

- 本章教学重点

- 动态联编及其与多态的关系
- 虚函数的特点及其在多态程序设计中的应用
- 抽象类及其应用
- 虚析构函数设计

- 教学难点

- 多态的技术原理和实现方法
- 用抽象类作接口的技术原理、实现方法和程序设计
- 用**dynamic\_cast**实现基类和派生类类型转换：向上转换和向下转换

# 第5章 多态

## 1. 多态的概念

- 多态是面向对象程序设计语言的又一重要特征，指的是不同对象接收到同一消息时会产生不同的行为。
- 简单地说，多态就是在同一个类或继承体系结构的基类与派生类中，用同名函数来实现各种不同的功能。

## 2. 多态与继承的关系

- 继承所处理的是类与类之间的层次关系问题
- 而多态则是处理类的层次结构之间，以及同一个类内部同名函数的关系问题。但通常是指继承结构中基类和派生类之间通过同名虚函数实现不同函数功能的问题。

# 5.1.1 多态概述

## 1、多态的实现形式

- 对象根据所接收的消息而做出动作，同样的消息为不同的对象接收时可导致完全不同的行动，该现象称为多态性。

– 解决的问题：单接口，多实现

- `Void F(家用电器 *p)`

```
{ p->on();  
  p->off();  
}
```

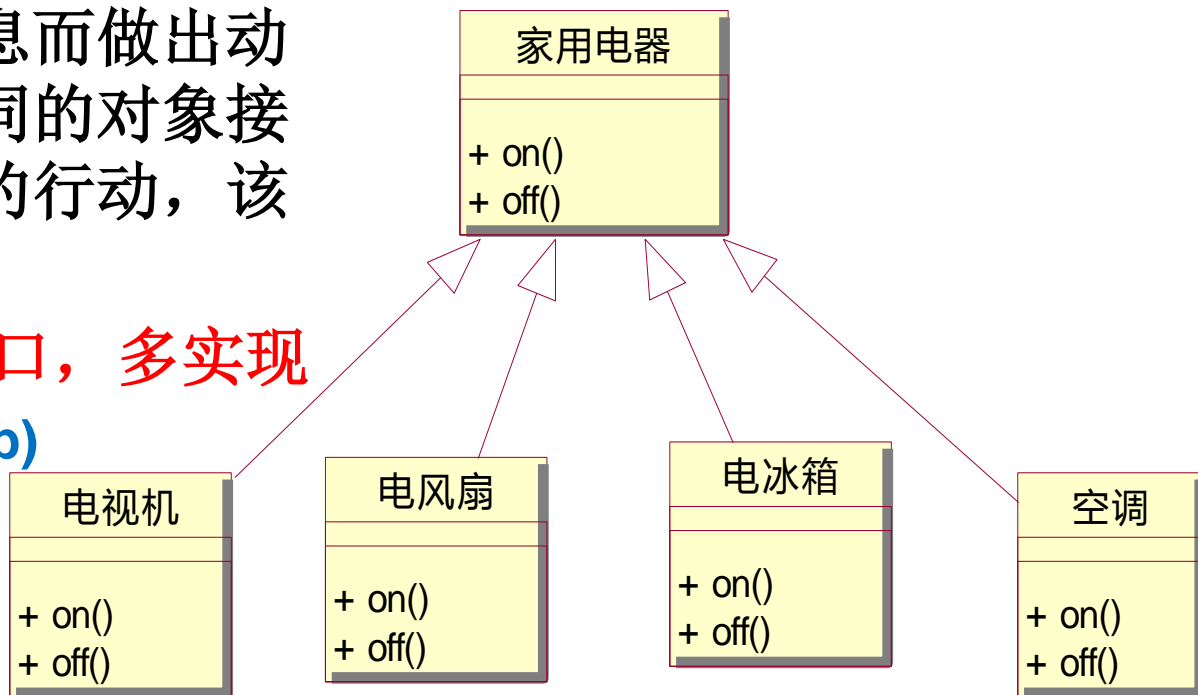
`Void main()`

```
{ 电视机 a; 电风扇 b; 电冰箱 c;
```

```
  F(&a)
```

```
  F(&b);
```

```
  .....
```



函数F以基类家用电器为接口，通过基类指针实现了对派生类电器的on和off函数的调用！

事实上，它可以访问任何派生对象的on和off函数

# 5.1.1 多态的概念

---

## 2. 多态的类型

– O O P 中广义的多态通常有3种表现形式

- ① **重载多态**：包括函数重载和运算符重载；
- ② **模板多态**：通过一个模板生成不同的函数或类（第7章介绍）；
- ③ **继承多态**：通过基类对象的**指针**（**引用**），调用不同派生类对象的重定义同名成员函数，表现出不同的行为。一般情况下，多态即指这种类型。

**实际意义上的多态是指继承多态！**

# 5.1.1 多态的概念

## 3. 实现多态的条件

— 要实现继承多态性，须具备三个必要条件：

- ① 要有继承；
  - ② 派生类要覆盖（重定义）基类的虚函数，即派生类具有和基类函数原形完全相同的虚成员函数；
  - ③ 把基类的指针或引用绑定到派生类对象上。
- 也就是说，没有继承，或者派生类没有重定义基类的虚函数，或者具备前两者，但直接把派生类对象赋值给基类对象（没有通过指针或引用），都不能实现多态。

## 5.1.1 多态的概念

【例5-1】设计一个管理动物声音的软件。

### (1) 问题分析

- 所有的动物都会发声，但是当没有说明是猫，狗或鸟等具体动物时，则不知道它发出什么声音。
- 虽然无法实施，但又确实知道动物有声音，面向对象程序设计语言提出了用（纯）虚函数来表达这类确实存在但又无法实施的抽象概念。
- 当到了可知的具体动物时，它会发出什么声音就是明确的了，此时再对相应的虚函数进行编码实现。

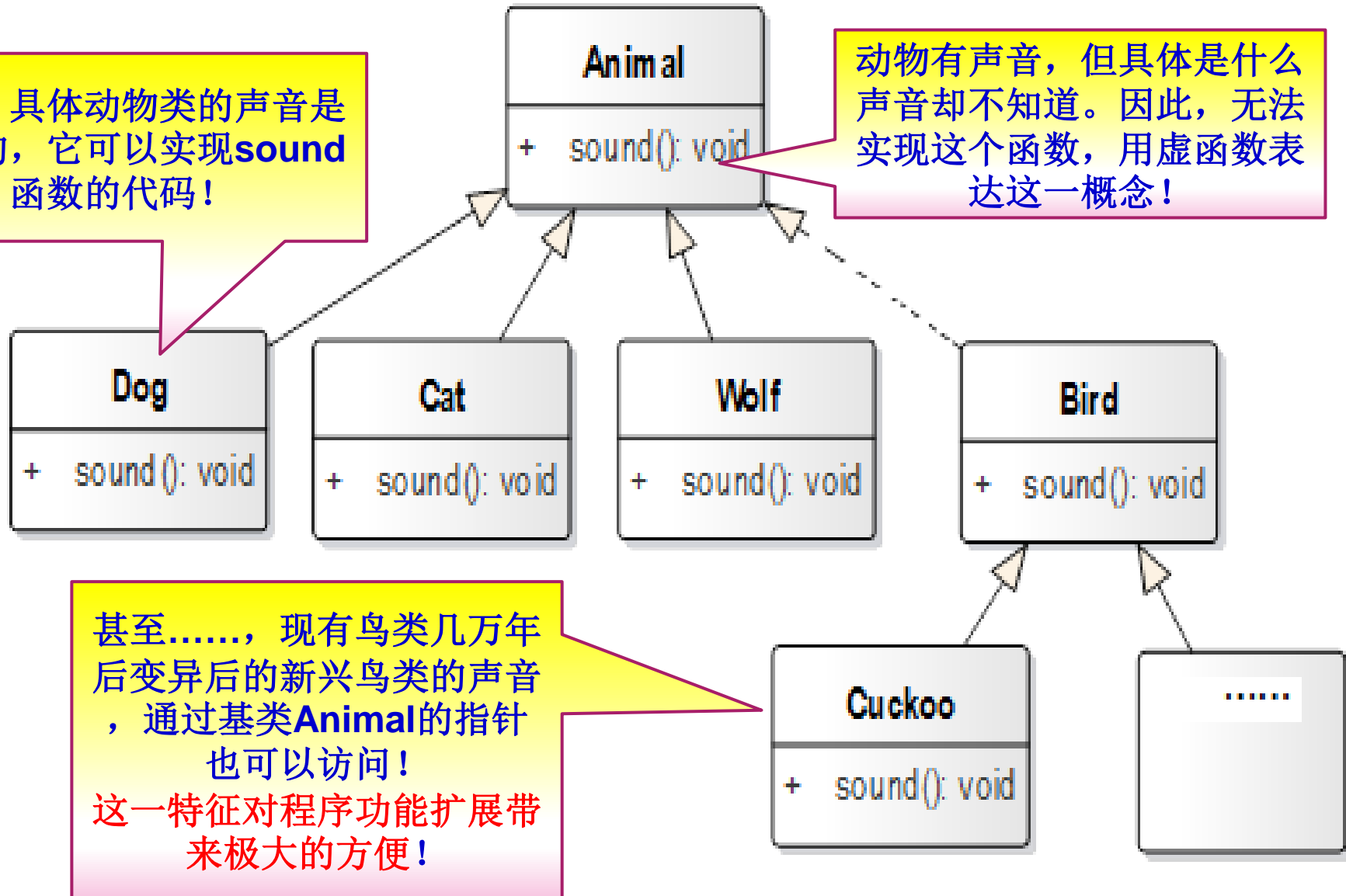
# 5.1.1 多态的概念

---

## (2) 数据抽象

- 用**Animal**表示动物类，用虚成员函数**sound**表示动物会发声这一行为。
- **Dog, Cat, Wolf, Bird**则是具体的动物，它们可以继承**Animal**的所有特征和行为。
- 每类动物能够发出什么声音是明确的，而且各不相同，需要**覆盖（重定义）从Animal继承来的sound成员函数**。
- **Animal**和**Dog**等动物的继承关系形成了图5-1所示的继承层次结构

# 动物继承体系





# 5.1.1 多态的概念

据Animal继承体系，可以设计出下面的简易类

```
class Animal {    //不知道动物会怎么叫！
    public: virtual void sound() { cout << "unknow!" << endl; }
};

class Dog :public Animal {    //狗儿叫声“汪汪汪！”
    public: void sound() { cout << "wang,wang,wang!" << endl; }
};

class Cat :public Animal {    //猫儿叫声“喵喵喵！”
    public: void sound() { cout << "miao,miao,miao!" << endl; }
};

class Wolf :public Animal {    //狼嚎叫声“”！
    public: void sound() { cout << "wu,wu,wu!" << endl; }
};
```

# 5.1.1 多态的概念

## (3) Animal的多态实现

多态是指当基类的指针（或引用）绑定到派生类对象上，通过此指针（引用）调用基类的成员函数时，实际上调用到的是该函数在派生类中的覆盖函数版本。

– 例如，对于上面的继承结构，下面的pA指针实现的就是多态。

```
void main() {
```

```
    Animal *pA;
```

```
    Dog dog;
```

```
    Cat cat;
```

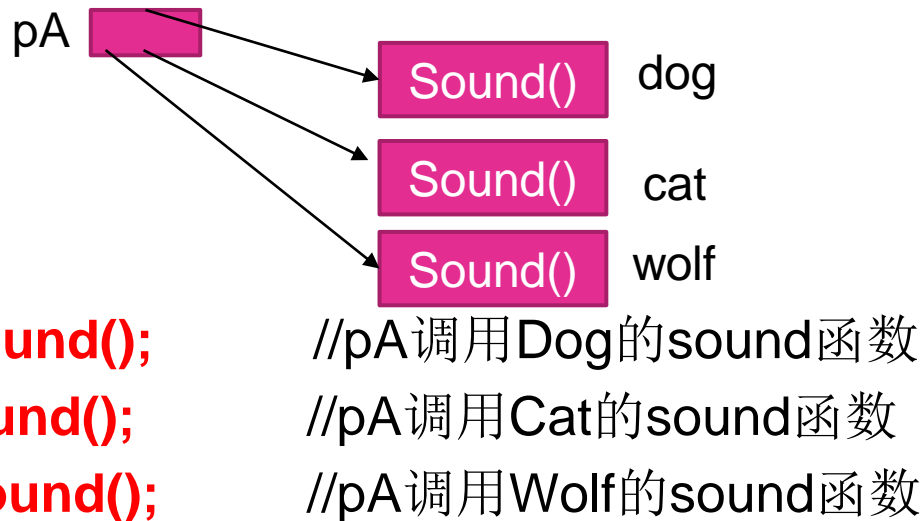
```
    Wolf wolf;
```

```
    pA = &dog; pA->sound();
```

```
    pA = &cat; pA->sound();
```

```
    pA = &wolf; pA->sound();
```

```
}
```



## 5.1.1 多态的概念

- 更一般地，多态更多地体现在用基类对象的指针或引用作为函数的参数，通过它调用派生类对象中的覆盖函数版本。
- 例如，针对Animal继承体系，设计animalSound函数管理每种动物的声音，多态能够很好地实现此需求。
- `void animalSound(Animal &animal) { animal.sound(); }`
  - animalSound函数体现了“一个接口，多种实现”。即以基类Animal的引用为接口，可以访问到图5-1所示继承体系中Animal类的任何派生类对象的sound函数。

```
Animal *pA;
```

```
Dog dog; Cat cat; Wolf wolf;
```

```
animalSound(&dog);           //调用Dog::sound()
```

```
animalSound(&cat);           //调用Cat::sound()
```

```
animalSound(&wolf);          //调用Wolf::sound()
```

## 5.1.2 多态的意义

- 多态使开发者在没有确定某些具体功能如何实施的情况下，可以站在高层（基类）设计并完成系统开发，等新功能明确并实现后，通过多态可以很容易地融入系统。多态对于软件开发和维护而言，意义重大。

### 1. 可替换性

- 多态对已存在代码具有可替换性。软件升级变得简单易行。
- 例如，在Animal继承体系中，如果现有的Dog类需要更新，重新编写了sound成员函数，只要该函数的原形保持不变，然后用新编写的Dog类更换以前的Dog类，原系统不受影响就能够调用新类的功能。

### 2. 可扩充性

- 增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。在不影响原系统功能的情况下，很容易派生新类，扩展系统新功能。

### 3. 灵活性

- 在多态程序结构中，基类提供接口，派生类提供实现，两者可以分离开来，使软件功能的整体设计和功能的逐步实现、扩展更加灵活。

## 5.1.3 多态与联编

---

### 1、联编的概念

- 一个程序常常会调用到来自于不同文件或C++库中的资源（如函数、对话框）等，需要经过编译、连接才能形成为可执行文件
- 在这个过程中要把调用函数名与对应函数（这些函数可能来源于不同的文件或库）关联在一起，这个过程就是**绑定（binding）**，又称**联编**。

### 2、联编的类型

- 根据把调用函数名和调用函数绑定在一起的时间，分为静态联编和动态联编。

## 5.1.3 多态与联编

---

### 3、静态联编

- 静态联编又称**静态绑定**，是指在编译程序时就根据调用函数提供的信息，把它所对应的具体函数确定下来，即在编译时就把调用函数名与具体函数绑定在一起。

### 4、动态联编

- 动态联编又称**动态绑定**，是指在编译程序时还不能确定函数调用所对应的具体函数，只有在程序运行过程中才能够确定函数调用所对应的具体函数，即在程序运行时才把调用函数名与具体函数绑定在一起。

## 5.1.3 多态与联编

### 4、多态性的实现方式

- **编译时多态性**： ---静态联编(连接)---系统在编译时就决定如何实现某一动作, 即对某一消息如何处理。静态联编具有执行速度快的优点。
- 在C++中的编译时多态性是通过**函数重载**和**运算符重载**实现的。
- **运行时多态性**： ---动态联编(连接)---系统在运行时动态实现某一动作, 即对某一消息在运行过程实现其如何响应。动态联编为系统提供了灵活和高度问题抽象的优点。
- 在C++中的运行时多态性是通过**继承**和**虚函数**实现的。

# 5.2 虚函数

## 5.2.1 虚函数的意义

### 1、回顾：4.6基类与派生类的赋值相容 (Page 208)

- 派生类对象可以赋值给基类对象。
- 派生类对象的地址可以赋值给指向基类对象的指针。
- 派生类对象可以作为基类对象的引用。

#### • 赋值相容的问题：

- 不论哪种赋值方式，都只能通过基类对象（或基类对象的指针或引用）访问到派生类对象从基类中继承到的成员，不能借此访问派生类定义的成员。

### 2、虚函数解决的问题

- 虚函数使得通过基类对象的指针或引用访问派生类重定义的虚成员函数可以施行。

```
class B{
public:
    int f(){}
}
class D:public B{
    int g(){}
    int f(){}
}
D d;
B b=d;
B *pb=&d;
B &rb=d;

d.f();           //B::f ()
pb->f();         //B::f()
rb.f();          //B::f()
```



## 5.2.1 虚函数的意义

【例5-2】某公司有经理、销售员、小时工等多类人员。经理按周计算薪金；销售员每月底薪800元，然后加销售提成，每销售一件产品提取销售利润的5%；小时工按小时计算薪金。每类人员都有姓名和身份证号等数据，设计管理员工薪金的程序。

### (1) 问题分析

经理、销售员、小时工等各类工作人员都是公司的雇员，每类人员都有姓名和身份证号等信息，可以将它们抽象为雇员类Employee，其余人员则从Employee类派生。

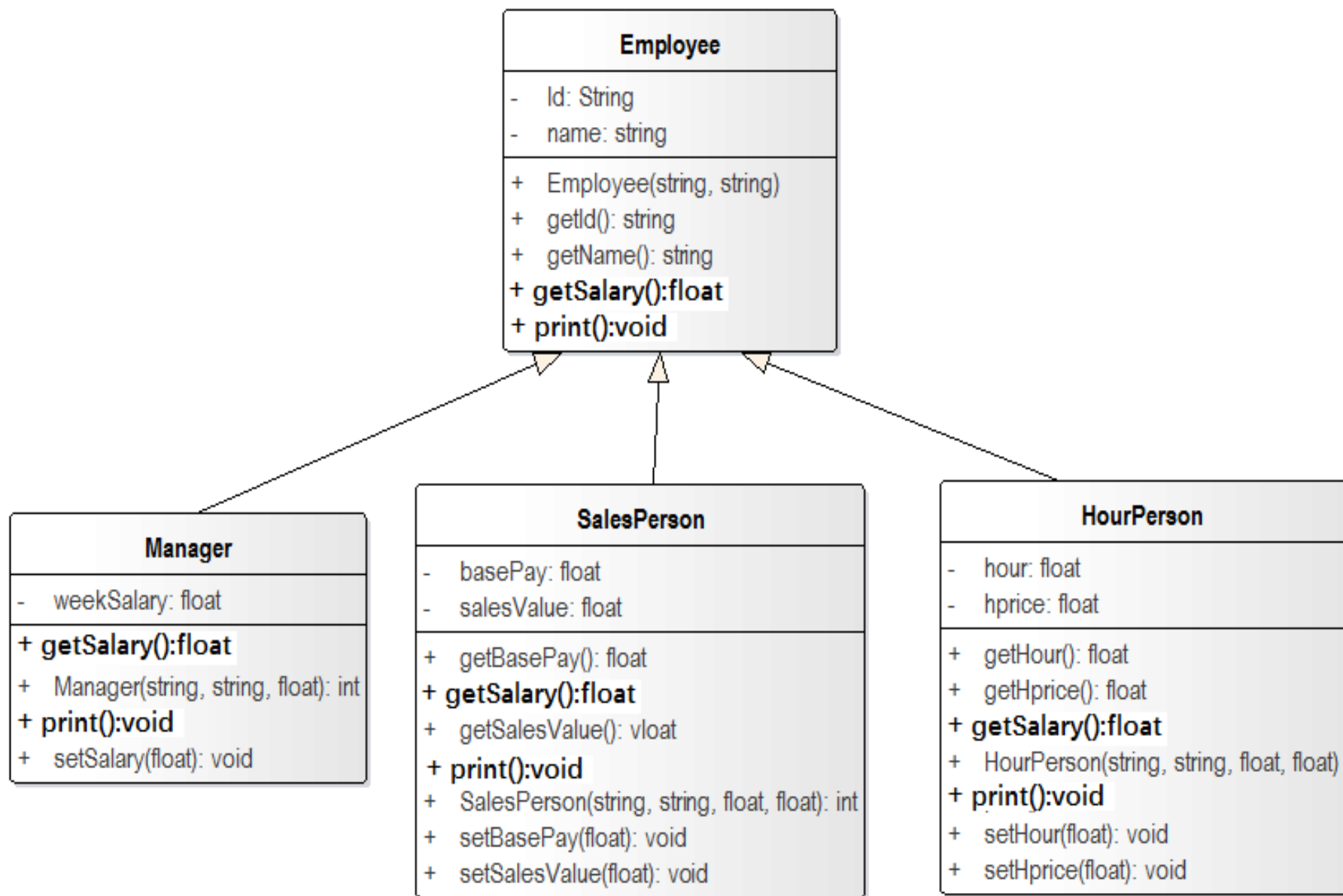
## 5.2.1 虚函数的意义

---

### (2) 数据抽象

- 雇员类**Employee**，用**name**和**Id**分别表示姓名和身份证编号；
- 将经理抽象成**Manager**类，用**WeeklySalary**表示周工资，并设计**setSalary/getSalary**修改和访问周工资。
- 将销售员抽象成**SalesPerson**类，用**basePay**表示底薪，**salesValue**表示销售额，以及**setBasePay/getBasePay**，**setSalesValue/getSalesValue**成员函数设置和读取底薪与销售额数据。
- 将小时工抽象成**HourPerson**类，用**hprice**表示小时工资，用**hour**表示工作时间，并用**setHprice/getHprice**，**setHour/getHour**设置/读取小时工价及工作时间。

# 数据抽象结果——类继承体系



# 人员管理的非虚函数简化实现版本

//Eg5-2.cpp

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class Employee{
```

```
public:
```

```
    Employee(string Name ,string id){ name=Name; Id=id; }
```

```
    string getName(){ return name; }           //返回姓名
```

```
    string getID(){ return Id; }               //返回身份证号
```

```
    float getSalary(){ return 0.0; }          //返回薪水
```

```
    void print() {                          //输出姓名和身份证号
```

```
        cout<<"姓名: "<<name<<"\t\t 编号: "<<Id<<endl;
```

```
}
```

```
private:
```

```
    string name;
```

```
    string Id;
```

```
};
```

# 人员管理的非虚函数简化实现版本

```
class Manager:public Employee{
public:
    Manager(string Name,string id,float s=0.0):Employee(Name,id){
        WeeklySalary=s;
    }
    void setSalary(float s) { WeeklySalary=s; }    //设置经理的周薪
    float getSalary(){ return WeeklySalary; }      //获取经理的周薪
    void print(){                                  //打印经理姓名、身份证、周薪
        cout<<"经理: "<<getName()<<"\t\t 编号: "<<getID()
            <<"\t\t 周工资: "<<getSalary()<<endl;
    }
private:
    float WeeklySalary;                           //周薪
};

void main(){
    Employee e("黄春秀","NO0009"),*pM;
    Manager m("刘大海","NO0001",128);
    m.print();
    pM=&m;
    pM->print();
    Employee &rM=m;
    rM.print();
}
```

## 5.2.1 虚函数的意义

- 程序的运行结果如下：

经理: 刘大海          编号: NO0001          周工资: 128

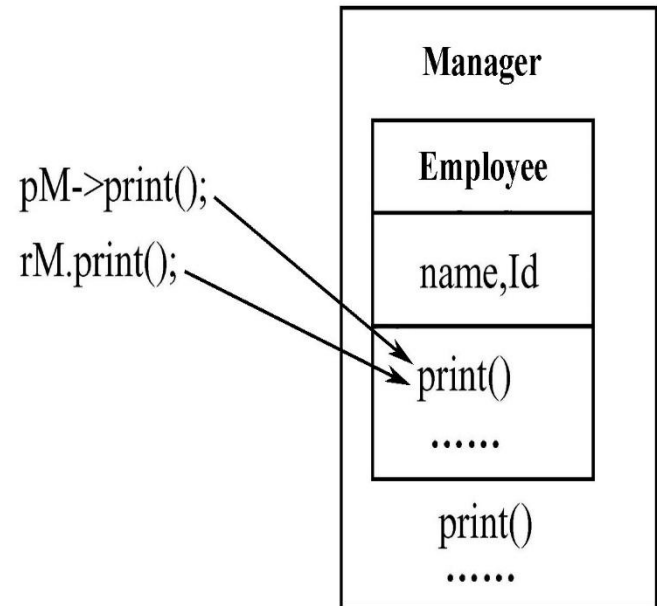
姓名: 刘大海          编号: NO0001

姓名: 刘大海          编号: NO0001

- 输出的第2、3行表明，通过基类对象的指针和引用只访问到了在基类中定义的`print`函数。

- 原因：

- `pM->print()\rM.print()` 采用静态联编
- 而`pM`、`rM`的类型都是`Employee`
- 所以只能访问`Employee`类的成员



## 5.2.1 虚函数的意义

- 将基类Employee的print指定为虚函数，如下形式：

```
class Employee{
```

```
.....
```

```
    virtual void print(){
```

```
        cout<<"姓名: "<<name<<"\t\t 编号: "<<Id<<endl;
```

```
    }
```

```
};
```

- 将得到下面的程序运行结果：

经理：刘大海

编号：NO0001

周工资：128

经理：刘大海

编号：NO0001

周工资：128

经理：刘大海

编号：NO0001

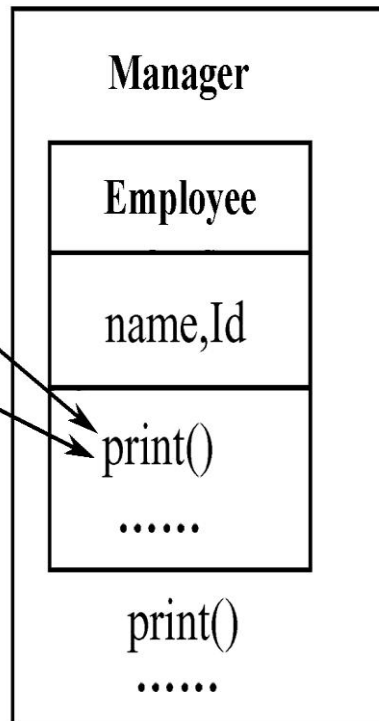
周工资：128

## 5.2.1 虚函数的意义

- 基类指针或引用指向派生类对象时，虚函数与非虚函数的对象，**图左为非虚函数**，**图右为虚函数**

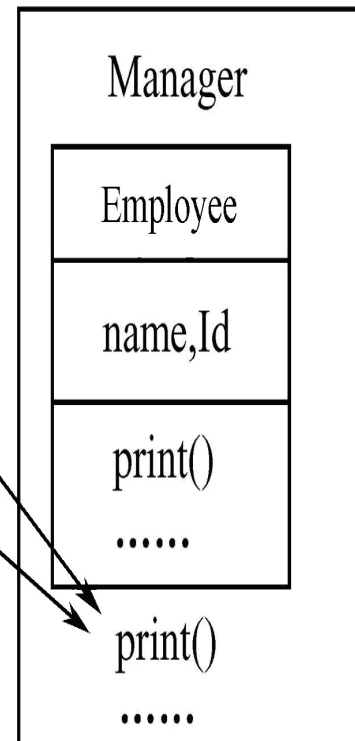
**print()非虚函数**  
，采用静态联编，  
只能绑定到  
pM,rM的定义类  
型Employee的  
print()上

pM->print();  
rM.print();



**print()为虚函数**，采用动态  
联编，  
运行到调用语句时，才绑定  
到pM,rM实际内存对象类型  
Manager的print()上

pM->print();  
rM.print();





# 5.2.1 虚函数的意义

## 1、什么是虚函数

- 用**virtual**关键字修饰的成员函数，**virtual**关键字其实质是告知编译系统，被指定为**virtual**的函数采用动态联编的形式编译。
- 只有类的**成员函数才能声明为虚函数**，普通函数（不属于任何类）不能定义为虚函数
- **多态类**：拥有虚函数的类也称为多态类。

## 2、虚函数的定义形式

```
class x{  
    .....  
    virtual f(参数表);  
}
```

## 3、虚函数的执行机制

- 如果基类中的非静态成员函数被定义为虚函数，且派生类覆盖了基类的虚函数，当通过基类的指针或引用调用派生类对象中的虚函数时，编译器将执行动态绑定，**调用到该指针（或引用）实际所指对象所在类中的虚函数版本。**

## 5.2.1 虚函数的意义

### 4、虚函数的虚特征

- 基类指针指向派生类的对象时，通过**该指针（或引用）**访问其虚函数时将调用派生类的版本。例题：**没有虚函数的情况**

```
class B {  
public:  
    void f ( ) {cout << "B::f";}  
};  
class D : public B {  
public:  
    void f ( ) { cout << "D::f"; }  
};  
void main()  
{  
    D d;  
    B * pb = & d;  
    pb->f( );  
}
```

运行结果

B::f

## 5.2.1 虚函数的意义

- 例题：虚函数版

```
class B
{public: virtual void f ( ) {cout << "B::f";}; };
class D : public B
{public: void f ( ) { cout << "D::f"; };
```

```
void main()
{
    D d;      B * pb = & d;
    pb->f( );
}
```

运行结果

D::f

- 总结：通过指向派生类对象的基类指针访问函数成员时，
  - 非虚函数由定义指针的类型决定调用的函数版本
  - 虚函数由指针实际指向的对象的类型决定调用的函数版本

## 5.2.3 虚函数的特性

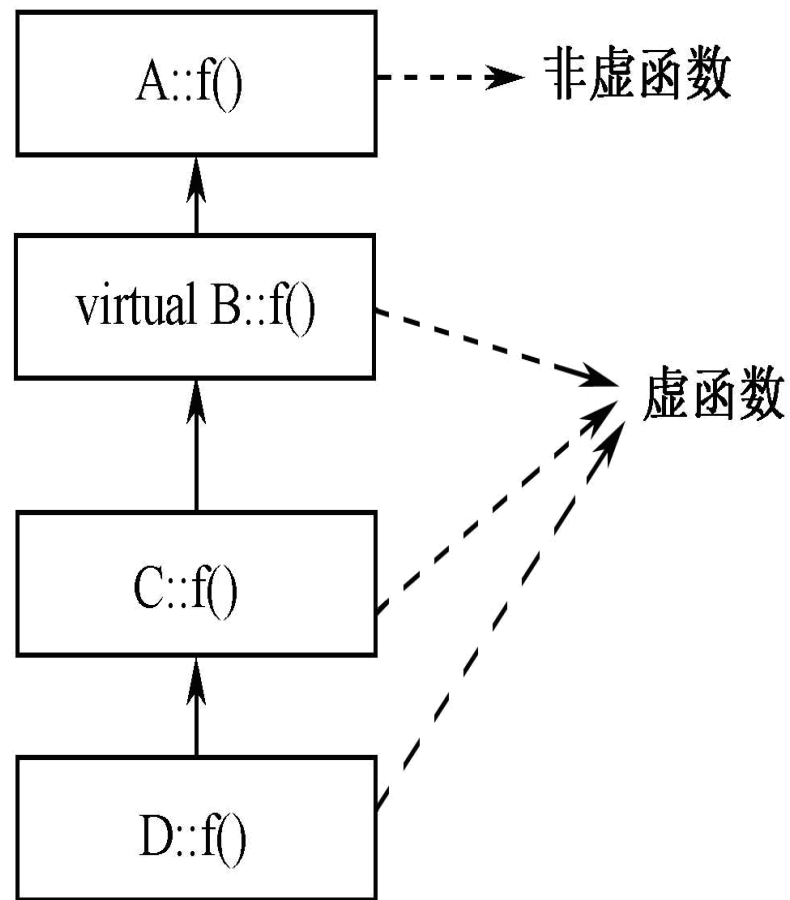
1、一旦将某个成员函数声明为虚函数后，它在继承体系中就永远为虚函数了。

【例5-3】虚函数与派生类的关系。

```
#include <iostream>
#include<string>
using namespace std;
class A {
public:
    void f(int i){cout<<"...A"<<endl;};           //非虚函数
};
class B: public A {
public:
    virtual void f(int i){cout<<"...B"<<endl;}      //虚函数
};
class C: public B {
public:
    void f(int i){cout<<"...C"<<endl;};           //虚函数
};
```

## 5.2.3 虚函数的特性

```
class D: public C{
public:
    void f (int){cout<<"...D"<<endl;
};
void main(){
    A *pA,a;
    B *pB, b;   C c;   D d;
    pA=&a;   pA->f(1);   //A::f()
    pA=&b;   pA->f(1);   //A::f()
    pA=&c;   pA->f(1);   //A::f()
    pA=&d;   pA->f(1);   //A::f()
}
```

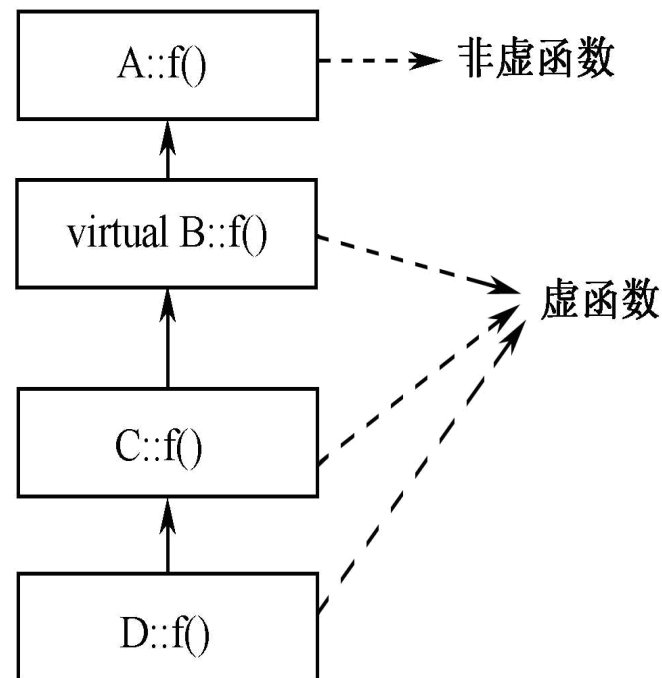


## 5.2.3 虚函数的特性

2、如果基类定义了虚函数，当通过基类指针或引用调用派生类对象时，将访问到它们实际所指对象中的虚函数版本。

- 例如，若把例5-3中的main的pA指针修改为pB，将会体现虚函数的特征。

```
void main(){  
    A *pA,a;  
    B *pB, b; C c; D d;  
    // pB=&a; pB->f(1); //错误，派生类不能访问基类对象  
    pB=&b; pB->f(1);    //调用B::f  
    pB=&c; pB->f(1);    //调用C::f  
    pB=&d; pB->f(1);    //调用D::f  
}
```



## 5.2.3 虚函数的特性

**3、只有通过基类对象的指针和引用访问派生类对象的虚函数时，才能体现虚函数的特性。**

**【例5-4】** 只能通过基类对象的指针和引用才能实现虚函数的特性。

**//Eg5-4.cpp**

**#include <iostream>**

**using namespace std;**

**class B{**

**public:**

**virtual void f(){ cout << "B::f"<<endl; };**

**};**

**class D : public B{**

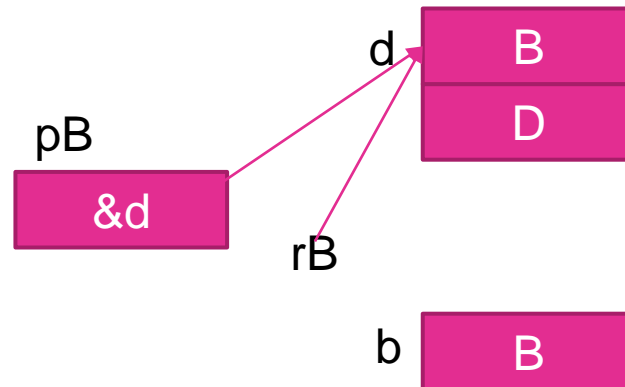
**public:**

**void f(){ cout << "D::f"<<endl; };**

**};**

## 5.2.3 虚函数的特性

```
void main(){
    D d;
    B *pB = &d, &rB=d;
    B b;
    b=d;
    b.f();
    pB->f();
    rB.f();
}
```



本程序的运行结果如下：第1行输出没有体现虚函数特征

B::f

D::f

D::f



## 5.2.3 虚函数的特性

4、派生类中的虚函数要保持其虚特征，必须与基类虚函数的函数原型完全相同，否则就是普通的重载函数，与基类的虚函数无关。

【例5-5】 基类B和派生类D都具有成员函数f，但它们的参数类型不同，因此不能体现函数f在派生类D中的虚函数特性。

**//Eg5-5.cpp**

**#include <iostream>**

**using namespace std;**

**class B{**

**public:**

**virtual void f(int i){ cout << "B::f"<<endl; };**

**};**

## 5.2.3 虚函数的特性

```
class D : public B{
public:
    int f(char c){ cout << "D::f..."<<c<<endl; }
};
void main(){
    D d;
    B *pB = &d, &rB=d, b;
    pB->f('1');
    rB.f('1');
}
```

本程序的运行结果如下：

B::f

B::f

此运行结果表明  
， 没有实现虚特  
征！

## 5.2.3 虚函数的特性

**5、派生类通过从基类继承的成员函数调用虚函数时，将访问到派生类中的版本。**

**【例5-6】 派生类D的对象通过基类B的普通函数f调用派生类D中的虚函数g**

//Eg5-6.cpp

```
#include <iostream>
using namespace std;
class B{
public:
    void f(){ g(); }
    virtual void g(){ cout << "B::g"; }
};
class D : public B{
public:
    void g(){ cout << "D::g"; }
};
void main(){
    D d;
    d.f();
}
```

调用成员函数时，  
若为虚函数，  
将访问：

实际调用对象对  
应类中的函数版  
本！

## 5.2.3 虚函数的特性

【例5-7】 分析下面程序的输出结果，理解虚函数的调用过程。

```
class B{
public:
    void f ( ) { cout << "bf "; }
    virtual void vf ( ) { cout << "bvf "; }
    void ff ( ) { vf(); f(); };
    virtual void vff ( ) { vf(); f(); }
};

class D: public B{
public:
    void f ( ) { cout << "df "; }
    void ff ( ) { f(); vf(); }
    void vf ( ) { cout << "dvf "; }
};

void main()
{
    D d;    B * pB = &d;
    pB->f();  pB->ff();    pB->vf();    pB->vff();
}
```

运行结果

bf dvf bf dvf dvf bf

## 5.2.3 虚函数的特性

---

6、只有类的非静态成员函数才能被定义为虚函数，类的构造函数和静态成员函数不能定义为虚函数。

- 原因是虚函数在继承层次结构中才能够发生作用，而静态成员是不能够被继承的，构造函数虽然可被继承，但它只用于创建本类对象。

7、内联函数也不能是虚函数。

- 因为内联函数采用的是静态联编的方式，而虚函数是在程序运行时才与具体函数动态绑定的，采用的是动态联编的方式，即使虚函数在类体内被定义，C++编译器也将它视为非内联函数。

## 5.3 虚析构函数

- 为什么要用虚析构函数？

- 原因是：假定使用**delete**来销毁一个指向派生类的基类指针，如果基类析构函数不是虚函数，就如一个普通成员函数那样，**delete**函数调用的就是基类析构函数，而不会调用派生类的析构函数。
- 这样，在通过基类对象的引用或指针调用派生类对象时，**将致使对象析构不彻底！**

**【例5-8】** 在非虚析构函数的情况下，通过基类指针对派生对象的析构是不彻底的。

## 5.3 虚析构造函数

```
//Eg5-8.cpp
#include <iostream>
using namespace std;
class A{
public:
    ~A(){ cout<<"call A::~~A()"<<endl; }
};
class B:public A{
    char *buf;
public:
    B(int i){buf=new char[i];}
    ~B(){
        delete [] buf;
        cout<<"call B::~~B()"<<endl;
    }
};
void main(){
    A* a=new B(10);
    delete a;
}
```

程序运行结果：

**call A::~~A()**

此结果表明没有析构buf

## 例5-8的虚析构函数版本

```
class A{
public:
    virtual ~A(){
        cout<<"call A::~~A()"<<endl;    }
};

class B:public A{
    char *buf;
public:
    B(int i){buf=new char[i];}
    virtual ~B(){
        delete [] buf;
        cout<<"call B::~~B()"<<endl;    }
};

void main(){
    A* a=new B(10);
    delete a;
}
```

程序运行结果:

call B::~~B()

call A::~~A()

此结果表明回收了  
buf空间!



## 5.4 纯虚函数和抽象类

### 1、纯虚函数与抽象类的概念

- 在有些情况下，定义类的时候却并不知道如何实现它的某些成员函数，但这些函数又确实存在。
- 定义该类的目的也并不是为了建立它的对象，而是为了表达某种概念，并作为继承结构顶层的基类，然后以它为接口访问派生类对象。
- 那些在基类中无法实现的成员函数，在派生类中却有具体的实现方法。
- 在面向对象程序设计语言中，用纯虚函数来表示这类函数。具有纯虚函数的类就称为抽象类。

# 5.4.1 纯虚函数和抽象类

---

## 1、纯虚函数的概念

纯虚函数是指并无实现代码，在声明时被初始化为0的类成员函数。

## 2、纯虚函数的声明形式

```
class X{  
.....  
    virtual returnType funcName (param) = 0;  
}
```

## 3、抽象类与纯虚函数的关系

只要含有纯虚函数（无论是一个，还是多个）的类就是抽象类。

# 5.4.1 纯虚函数和抽象类

---

## 4、C++对抽象类的限定

- ① 抽象类中含有纯虚函数，由于纯虚函数没有实现代码，所以不能建立抽象类的对象。
- ② 抽象类只能作为其他类的基类，又称为抽象基类。但是，可以创建抽象类的指针或引用，并通过它们访问到派生类对象，实现运行时的多态性。
- ③ 如果派生类只是简单地继承了抽象类的纯虚函数，而没有覆盖基类的纯虚函数，则派生类也是一个抽象类。

## 5.4.1 纯虚函数和抽象类

【例5-9】 在一个图形系统中，实现计算各种图形面积的程序设计。

问题分析：

- ① 所有图形都有面积，但只有落实到三角形、矩形等具体图形时才能够计算出它的面积。
- ② 设计抽象类**Figure**来表示图形这一概念，并为它设置纯虚函数**area**计算图形的面积。
- ③ 圆、三角形、矩形等具体图形则从**Figure**派生，由它们提供纯虚函数**area**的实现版本。借助于虚函数，就可以通过**Figure**的指针或引用访问到三角形、矩形等派生类实现的面积函数。

## 5.4.1 纯虚函数和抽象类

- //Eg5-9.cpp

```
#include <iostream>
using namespace std;
class Figure{
protected:
    double x,y;
public:
    void set(double i,double j){ x=i; y=j; }
    virtual void area()=0;                //纯虚函数
};
class Triangle:public Figure{
public:
    void area(){cout<<"三角形面积: "
                << x*y*0.5<<endl;}      //重写基类纯虚函数
};
```

## 5.4.1 纯虚函数和抽象类

```
class Rectangle:public Figure{
public:
    void area(int i){cout<<"这是矩形，它的面积是： "<<x*y<<endl;}
};
void main(){
    Figure *pF;
    // Figure f1;    //L1， 错误， 不能定义抽象类的对象
    // Rectangle r; //L2， 错误， 矩形的area函数不是基类虚函数
    //           的覆盖版本， 它们的参数表不一致。Rectangle仍然是抽象类
    Triangle t;           //L3
    t.set(10,20);
    pF=&t;
    pF->area();           //L4
    Figure &rF=t;
    rF.set(20,20);
    rF.area();           //L5
}
```

## 5.4.2 抽象类的应用

- 抽象类作为访问派生类的接口

- 在设计类的继承结构时，可以把各派生类都需要的功能设计成抽象基类的**虚函数**，每个**派生类**根据自己的情况**重新定义虚函数的功能**，以便描述每个类特有的行为。
- 由于抽象基类具有各派生类成员函数的虚函数版本，可以把它作为访问整个继承结构的接口，通过抽象基类的指针或引用访问在各个派生类中实现的虚函数，这种方式也称为**接口重用**，即不同的**派生类都可以把抽象基类作为接口，让其他程序通过此接口访问各派生类的功能**。

- 多态

- 从外部看：同一方法（函数）作用不同对象时，导致不同行为发生
- 从内部看：单接口、多实现

- 好处

- 代码重用
- 软件功能局部的修改和替代
- 抽象手段（抽象类）

## 5.4.2 抽象类的应用

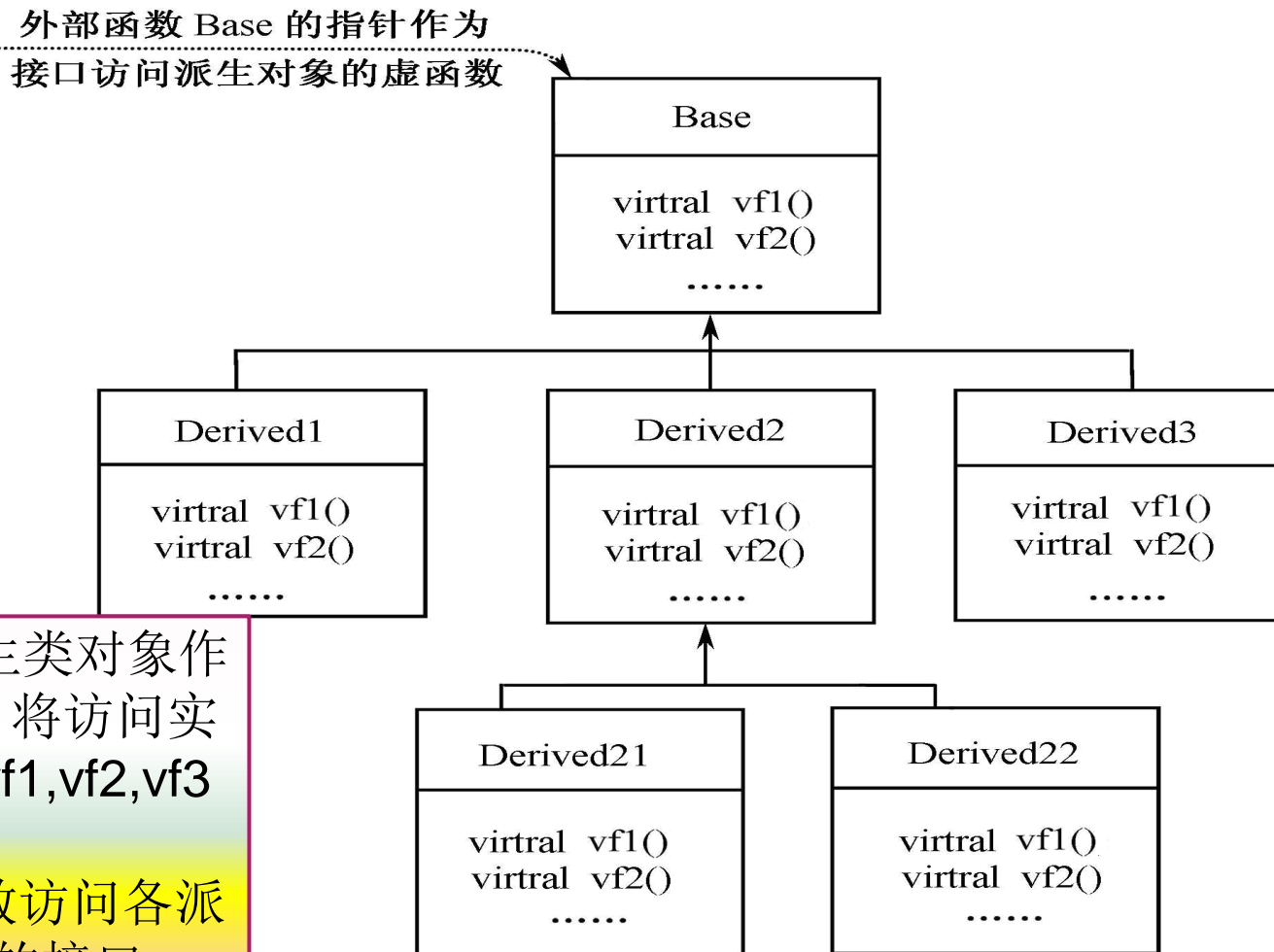
### • 抽象类的主要用途——作接口

外部函数 Base 的指针作为  
接口访问派生对象的虚函数

```
pf(Base *p){  
    p->vf1()  
    p->vf2()  
    p->vf3()  
    .....  
}
```

以Base的任何派生类对象作  
实参调用pf函数，将访问实  
参对象所在类的vf1,vf2,vf3  
等函数。

Base实际是pf函数访问各派  
生类成员函数的接口





## 5.4.2 抽象类的应用

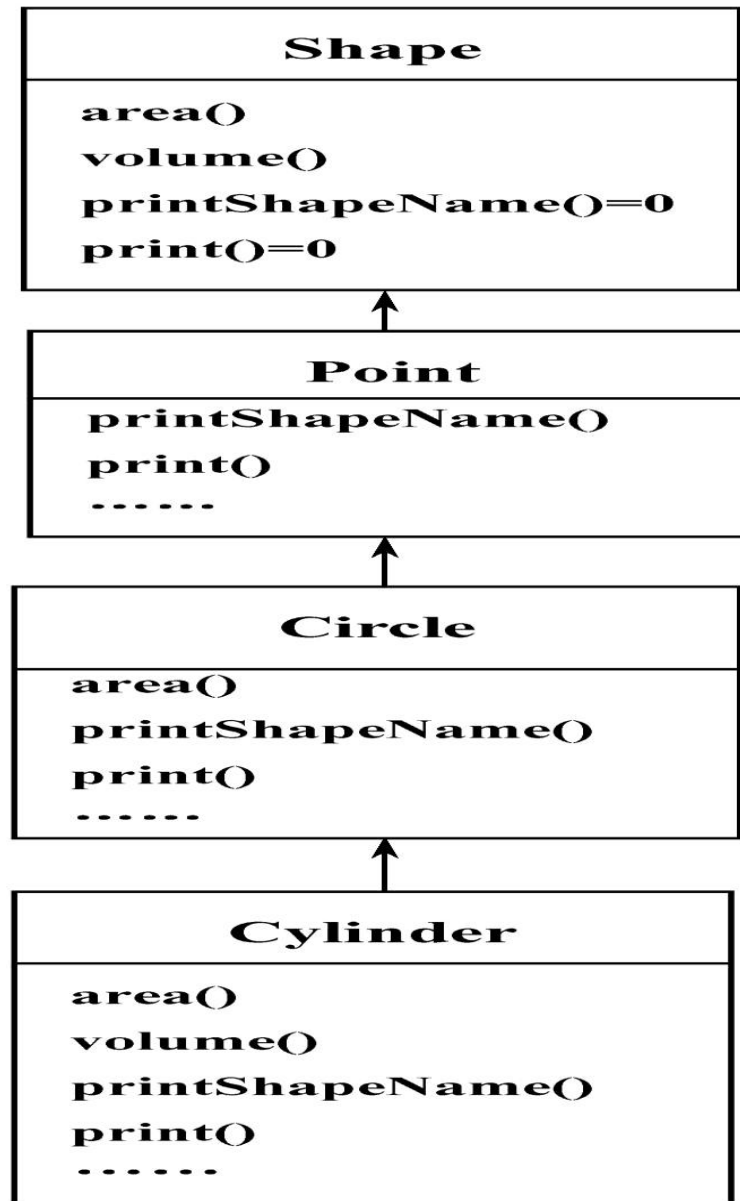
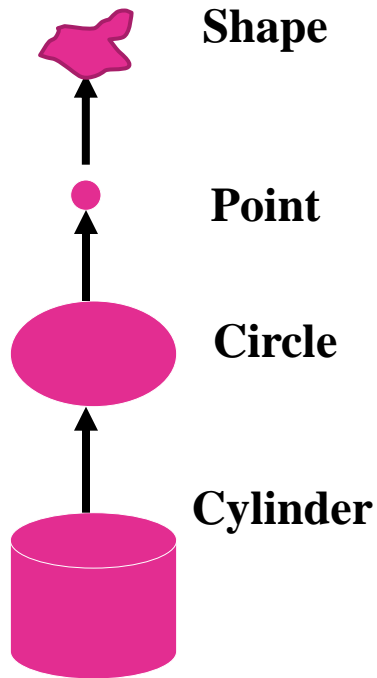
【例5-10】扩展例5-9图形面积和体积的程序功能，用接口与实现分离的方式计算点、圆、圆柱体几种图形每种图形的面积和体积，并且要求输出各种图形的类名字及各类定义对象的数据成员。

### 问题分析

- 点、圆、圆柱体、三角形、四边形等都是几何图形，它们都具有共性，比如有类型名，有面积、体积和周长等。但当没有具体到某种形状时，又无法计算，仅仅是个概念，但又确实存在，适合用纯虚函数和抽象类来描述它们。
- 用类**Shape**表示几何图形这一概念，把各类图形计算面积、体积的函数设置成它的虚成员函数**area**和**volume**，并设置纯虚函数**printShapeName**和**print**输出图形类型、面积、体积、圆半径等数据。
- 将点、圆、圆柱体等具体图形抽象成类**Point**、**Circle**、**Cylinder**，它们从**Shape**类派生，每个类都据自己的实情重定义从**Shape**继承到的**area**、**volume**等纯虚函数。
- 以**Shape**为接口，通过**Shape**的指针或引用能够访问到**Point**、**Circle**等派生类实现的**area**、**volume**等覆盖函数版本，实现各图形的面积和体积计算，以及各类图形数据输出等功能。

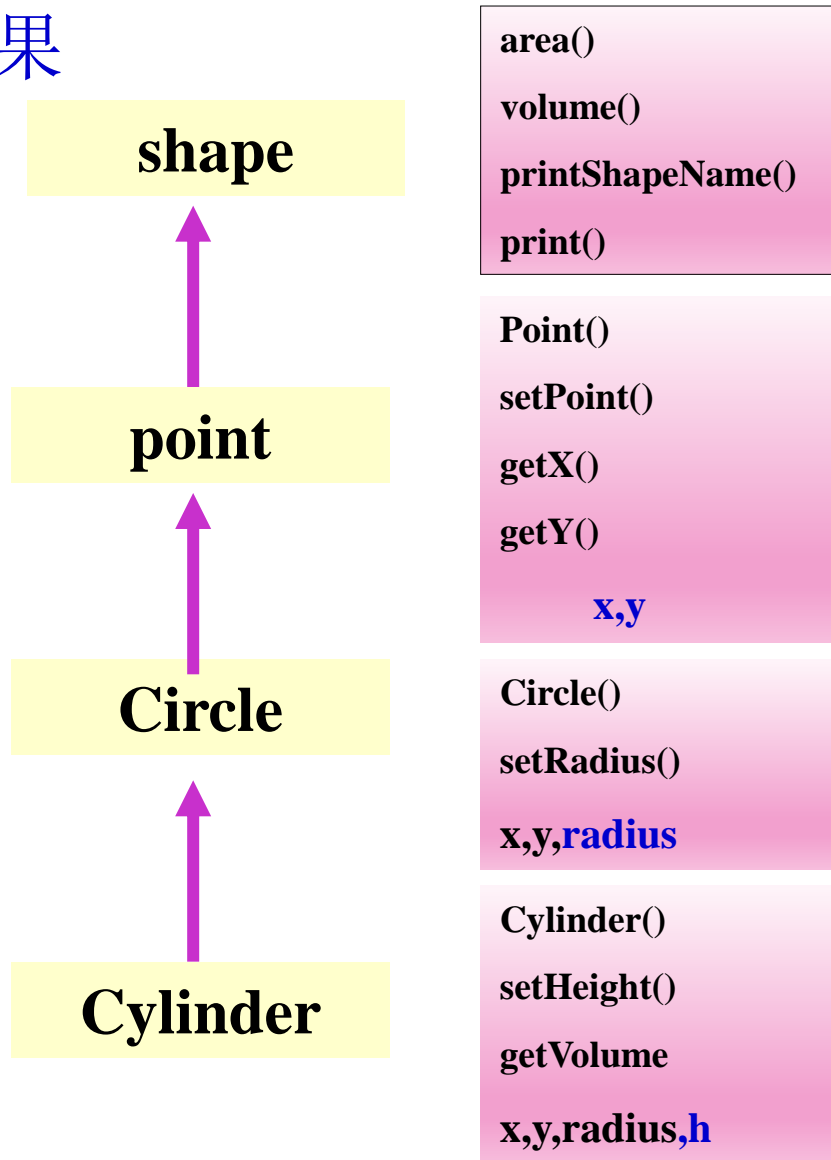
## 5.4.2 抽象类的应用

- 抽象结果的类图



## 5.4.2 抽象类的应用

### 抽象结果



**Shape**只是概念上的几何图形，永远不会有称为**shape**的对象存在，它的存在只是为了提供**point**, **circle**, **cylinder**的公有接口。所以**shape**的成员函数定义为：

```
area(){return 0;}  
volume(){return 0;}  
printShapeName()=0;  
Print()=0;
```

## 三种几何图形的成员：红字是必须重定义的虚函数

### Point

**Point()**  
**setPoint()**  
**getX()**  
**getY()**  
**x,y**

**printName()**  
**print()**

### Circle

**Circle()**  
**setRadius()**  
**x,y,radius**

**area()**  
**printName()**  
**print()**

### Cylinder

**Cylinder()**  
**setHeight()**  
**getVolume**  
**x,y,radius,h**

**area()**  
**volume()**  
**printName()**  
**print()**

# Shape.h

---

```
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream.h>

class Shape {
public:
    virtual double area() const { return 0.0; }
    virtual double volume() const { return 0.0; }

    virtual void printShapeName() const = 0;
    virtual void print() const = 0;
};

#endif
```

# Shape.cpp

---

- 不需要此源文件,因为没有函数要定义

# Point.h

---

```
#ifndef POINT_H
#define POINT_H
#include "shape.h"

class Point : public Shape {
public:
    Point( int = 0, int = 0 );
    void setPoint( int, int );
    int getX() const { return x; }
    int getY() const { return y; }
    virtual void printShapeName() const
        { cout << "Point: "; }
    virtual void print() const;
private:
    int x, y;
};

#endif
```

# Point.cpp

---

```
#include "point.h"
```

```
Point::Point( int a, int b ) { setPoint( a, b ); }
```

```
void Point::setPoint( int a, int b )  
{  
    x = a;  
    y = b;  
}
```

```
void Point::print() const  
{ cout << '[' << x << ", " << y << ']; }
```



# Circle.h

---

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"

class Circle : public Point {
public:
    Circle( double r = 0.0, int x = 0, int y = 0 );
    void setRadius( double );
    double getRadius() const;
    virtual double area() const;
    virtual void printShapeName() const { cout << "Circle: "; }
    virtual void print() const;
private:
    double radius; // radius of Circle
};

#endif
```

# Circle.cpp

---

```
#include "circle.h"
```

```
Circle::Circle( double r, int a, int b ) : Point( a, b )  
{ setRadius( r ); }
```

```
void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
```

```
double Circle::getRadius() const { return radius; }
```

```
double Circle::area() const  
{ return 3.14159 * radius * radius; }
```

```
void Circle::print() const  
{ Point::print(); cout << "; Radius = " << radius;  
}
```

# Cylinder.h

---

```
#ifndef CYLINDR_H
#define CYLINDR_H
#include "circle.h"

class Cylinder : public Circle {
public:
    Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

    void setHeight( double );
    double getHeight();
    virtual double area() const;
    virtual double volume() const;
    virtual void printShapeName() const {cout << "Cylinder: ";}
    virtual void print() const;
private:
    double height;
};

#endif
```

# Cylinder.cpp

---

```
#include "cylinder.h"
```

```
Cylinder::Cylinder( double h, double r, int x, int y ) : Circle( r, x, y )  
{ setHeight( h ); }
```

```
void Cylinder::setHeight( double h )  
{ height = h > 0 ? h : 0; }  
double Cylinder::getHeight() { return height; }
```

```
double Cylinder::area() const  
{  
    return 2 * Circle::area() + 2 * 3.14159 * getRadius() * height;  
}
```

```
double Cylinder::volume() const  
{ return Circle::area() * height; }
```

```
void Cylinder::print() const  
{  
    Circle::print();  
    cout << "; Height = " << height;  
}
```

# main.cpp

---

```
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point.h"
#include "circle.h"
#include "cylindr.h"
```

```
void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );
```

# main.cpp

---

```
void virtualViaPointer( const Shape *baseClassPtr )
{
    baseClassPtr->printShapeName();
    baseClassPtr->print();
    cout << "\nArea = " << baseClassPtr->area()
    << "\nVolume = " << baseClassPtr->volume() << "\n\n";
}
```

```
void virtualViaReference( const Shape &baseClassRef )
{
    baseClassRef.printShapeName();
    baseClassRef.print();
    cout << "\nArea = " << baseClassRef.area()
    << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```

# main.cpp

---

```
int main()
{
    cout << setiosflags( ios::fixed | ios::showpoint )
    << setprecision( 2 );

    Point point( 7, 11 );
    Circle circle( 3.5, 22, 8 );
    Cylinder cylinder( 10, 3.3, 10, 10 );

    point.printShapeName();
    point.print();
    cout << '\n';

    circle.printShapeName();
    circle.print();
    cout << '\n';

    cylinder.printShapeName();
    cylinder.print();
    cout << "\n\n";
```

# main.cpp

```
Shape *arrayOfShapes[ 3 ];

arrayOfShapes[ 0 ] = &point;

arrayOfShapes[ 1 ] = &circle;

arrayOfShapes[ 2 ] = &cylinder;

cout << "Virtual function calls made off "
<< "base-class pointers\n";

for ( int i = 0; i < 3; i++ )
    virtualViaPointer( arrayOfShapes[ i ] );

cout << "Virtual function calls made off "
<< "base-class references\n";

for ( int j = 0; j < 3; j++ )
    virtualViaReference( *arrayOfShapes[ j ] );

return 0;

}
```

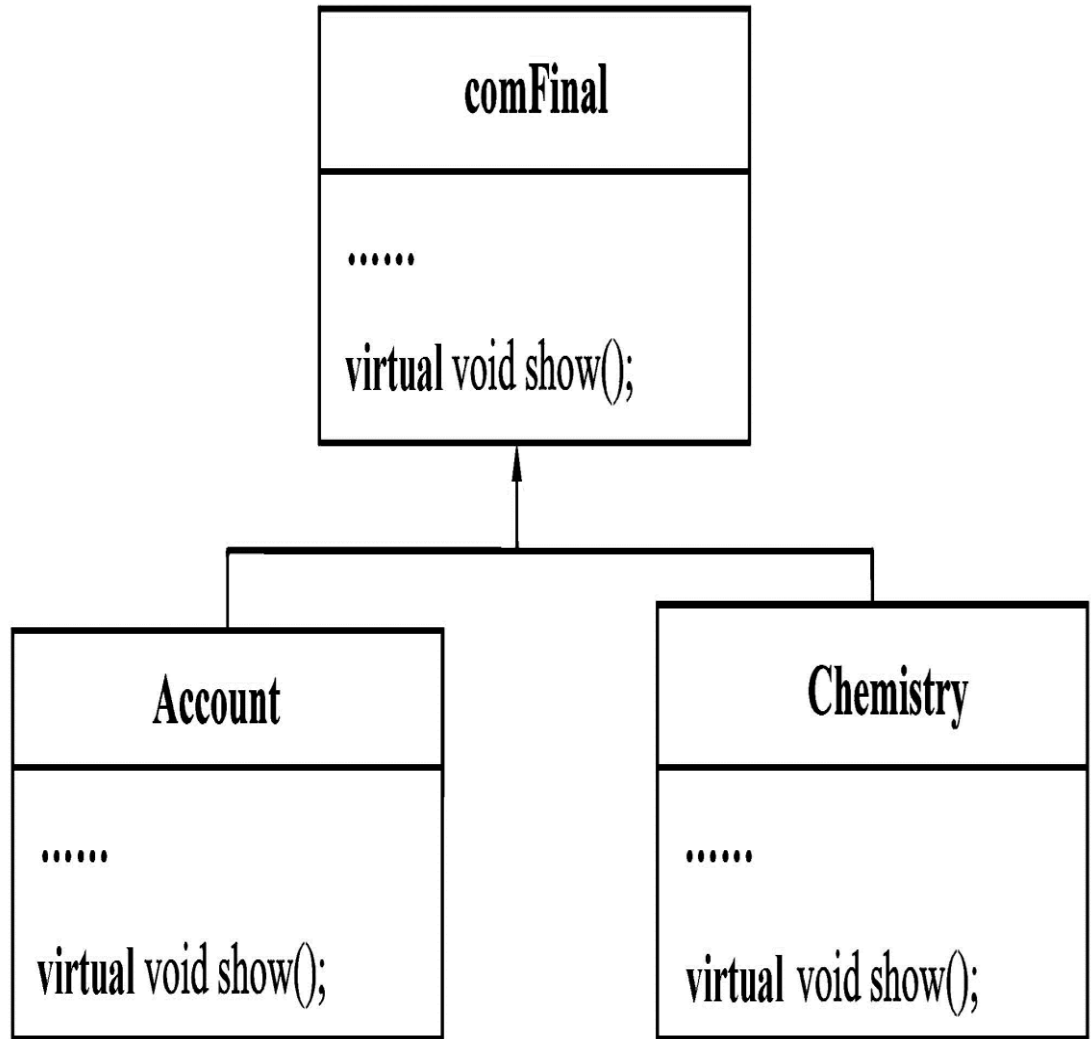
通过基类指针调用**3**个  
不同派生类对象，实  
现多态

通过基类引用调用**3**个  
不同派生类对象，实  
现多态



## 5.5 编程实作：多态编程应用

【例5-16】 现对4.10节的编程实作进行完善，将comFinal、Account、Chemistry中的成员函数show设计成虚函数，并设计一个访问该类继承结构的接口函数display，此函数通过基类comFinal对象的指针，访问派生类Account、Chemistry类对象的虚函数show。



## 5.5 编程实作：多态编程应用

实现ComFinal课程管理继承结构多态的编程过程如下：

<1> 打开4.10节建立在目录C:\course中的工程项目文件com\_main.dsw。

<2> 在comFinal类的成员函数show声明前面加上限定词**virtual**：

```
class comFinal{  
    .....  
    virtual void show();  
};
```

- 除此之外，comFinal、Account、Chemistry三个类的其他程序代码可不做任何修改。当然，也可以在Account、Chemistry类的show函数声明前面加上限定词**virtual**。由于Account、Chemistry是comFinal的派生类，即使它们的函数show前面没有**virtual**，也是虚函数。

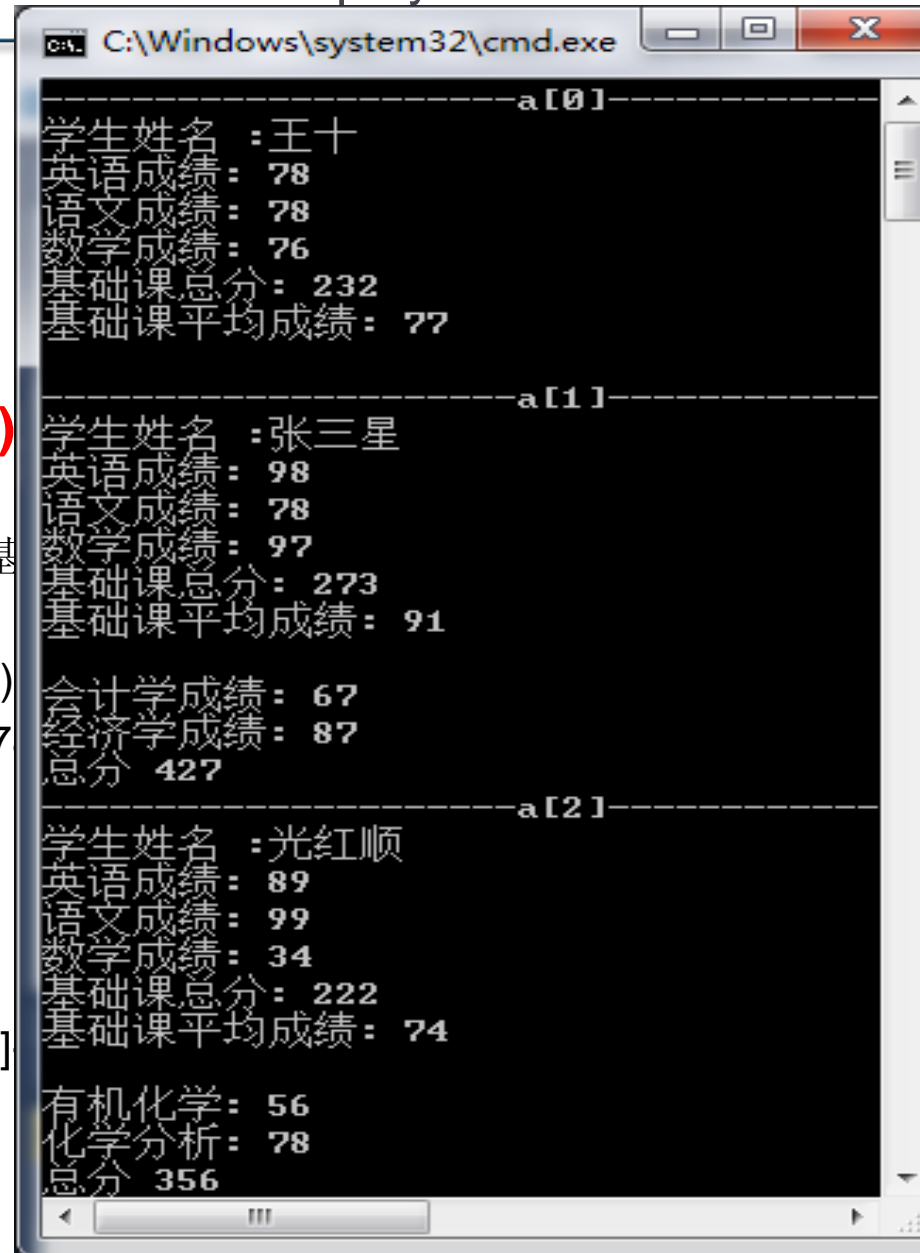
### <3> 改写主程序。

改写原来的主文件com\_main.cpp, 实现接口函数display和主函数:

```
//com_main.cpp
#include "comFinal.h"
#include "Chemistry.h"
#include "Account.h"
#include <iostream>
using namespace std;

void display(comFinal* p) { p->show(); }

void main() {
    comFinal *a[3];           //a为基
    comFinal c("王十", 78, 78, 76);
    Account a1("张三星", 98, 78, 97, 67, 87)
    Chemistry c1("光红顺", 89, 99, 34, 56, 7
    a[0] = &c;
    a[1] = &a1;
    a[2] = &c1;
    for (int i = 0; i < 3; i++) {
        cout << "-----a[" << i << " ]
        display(a[i]);
    }
}
```



```
C:\Windows\system32\cmd.exe

-----a[0]-----
学生姓名 :王十
英语成绩: 78
语文成绩: 78
数学成绩: 76
基础课总分: 232
基础课平均成绩: 77

-----a[1]-----
学生姓名 :张三星
英语成绩: 98
语文成绩: 78
数学成绩: 97
基础课总分: 273
基础课平均成绩: 91
会计学成绩: 67
经济学成绩: 87
总分 427

-----a[2]-----
学生姓名 :光红顺
英语成绩: 89
语文成绩: 99
数学成绩: 34
基础课总分: 222
基础课平均成绩: 74
有机化学: 56
化学分析: 78
总分 356
```