



华中科技大学

第8章 多态与虚函数

许向阳

xuxy@hust.edu.cn





大纲

8.0 多态性

8.1 虚函数

8.2 虚析构函数

8.3 类的引用

8.4 抽象类

8.5 虚函数友元与晚期绑定

8.6 有虚函数的内存布局





要点

什么是虚函数？

如何使用虚函数？

虚函数的工作原理？

核心：动态多态

动态多态的应用

动态多态的内在运行机理



8.0 多态性

菜农:



商贩:



食客:





8.0 多态性

水果：买水果，水果叫什么名字，水果什么价格。
在不同的水果摊位前，可以使用**同一样**的词汇。
一词多义

```
苹果老板: SoldFruit() {  
    cout<< “好吃的苹果，便宜呀”;  
}
```

```
香蕉老板: SoldFruit() {  
    我这有生一点的香蕉，也有熟香蕉;  
    您要哪一种?  
    .....  
}
```



8.0 多态性

主人 → 客人

客人吃水果





8.0 多态性

静态多态

相同的函数名称，但参数不同

动态多态

- 相同的函数名称、相同的参数
- 分属不同的类；基类与多个派生类
- 第一个隐含的参数 `this` 所指的类不相同
即函数调用时，前面的对象不同。
- 函数的返回类型一般也相同
除非分别返回指向基类 和派生类的指针或者引用





8.0 多态性

多态性:

- 具有相似功能的不同函数，使用同一名称
- 用相同的调用方式，调用不同功能的同名函数。
- 动态多态，用同一个接口、同一个调用形式，实现不同实例函数的访问

我问 -> 白菜什么价？

我对面站的是谁（即指针指向的对象）？

菜农？ 商贩？ 餐馆服务员？





8.0 多态性

多态性的实现——联编

联编：绑定、装配

将一个标识符名和一个地址联系在一起。

静态联编：前期联编、早期联编

在生成可执行程序时已经完成。

编译时多态性：函数重载、运算符重载

动态联编：晚期联编、后期联编

程序运行时才动态完成。

运行时多态：使用虚函数





8.1 虚函数

virtual 函数类型 函数名(形式参数表);

基类中，在函数声明时，加“**virtual**”。
在体外实现时，不要加 **virtual**。

```
class fruit {  
    public:  
        virtual void EatFruit();  
};  
  
void fruit::EatFruit() {  
    cout << "Can not eat abstract fruit !" << endl;  
}
```





8.1 虛函數

```
class apple : public fruit {  
    public:  
        void EatFruit()  
        {  
            cout << " I like to eat apple." <<endl;  
        }  
};
```

```
class banana : public fruit {  
    public:  
        void EatFruit()  
        { cout << " Banana, Ha Ha." <<endl; }  
};
```





8.1 虚函数

```
fruit    *ptrfruit;  
apple    ptrapple;  
banana   ptrbanana;
```

```
int i;  
cout << "select fruit : 1 apple , other banana "<<endl;  
cin >>i;  
if (i==1)    ptrfruit = &ptrapple;  
else    ptrfruit = &ptrbanana;
```

```
ptrfruit->EatFruit();
```

Q: 输入1, 程序的运行结果是什么?
输入2, 程序的运行结果是什么?
去掉 virtual, 程序的运行结果又是什么?





8.1 虚函数

Microsoft Visual Studio 调试控制台

```
select fruit : 1 apple , other banana  
1  
I like to eat apple.
```

Microsoft Visual Studio 调试控制台

```
select fruit : 1 apple , other banana  
2  
Banana, Ha Ha.
```

将 **virtual** 去掉，
程序的运行结果

Microsoft Visual Studio 调试控制台

```
select fruit : 1 apple , other banana  
1  
Can not eat abstract fruit !
```

Microsoft Visual Studio 调试控制台

```
select fruit : 1 apple , other banana  
2  
Can not eat abstract fruit !
```

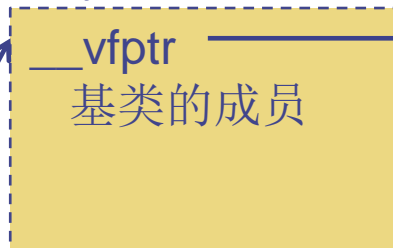
8.1 虚函数

```
fruit    *ptrfruit;  
fruit    myfruit;  
apple    myapple;  
banana   mybanana;
```

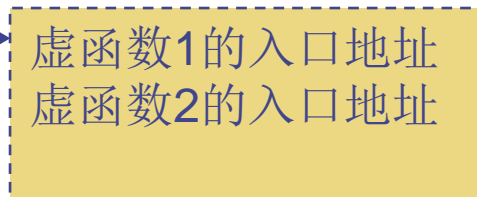
ptrfruit



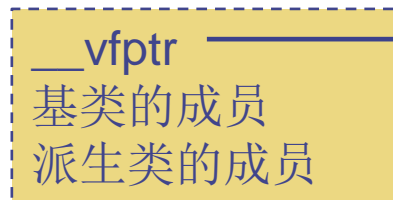
myfruit



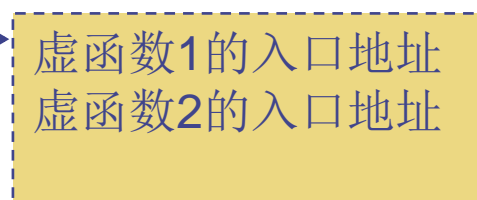
vftable



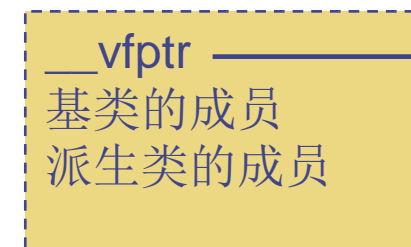
myapple



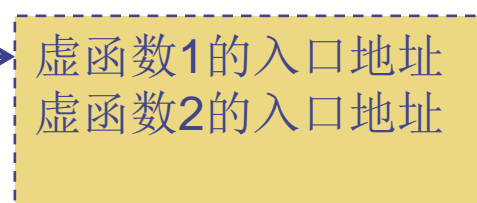
vftable



mybanana



vftable





8.1 虚函数

名称	值
ptrfruit	0x00acfce8
__vfptr	0x002a78d4 const fruit::`vftable'
[0]	0x002a1235 fruit::DispFruitName(void)
[1]	0x002a1023 fruit::EatFruit(void)

ptrfruit	0x00acfc0 {cAppleName=0x00acfcc0 "Red Apple" }
[apple]	{cAppleName=0x00acfcc0 "Red Apple" }
fruit	{...}
__vfptr	0x002a7924 const apple::`vftable'
[0]	0x002a1032 apple::DispFruitName(void)
[1]	0x002a1208 apple::EatFruit(void)
cAppleName	0x00acfcc0 "Red Apple"

```
fruit    *ptrfruit;  
fruit    myfruit;  
ptrfruit = &myfruit  
ptrfruit->EatFruit();
```

```
apple myapple  
ptrfruit = &myapple;  
ptrfruit->EatFruit();
```

ptrfruit=(fruit *) &myapple;
不是用public 派生控制时，强制类型转换





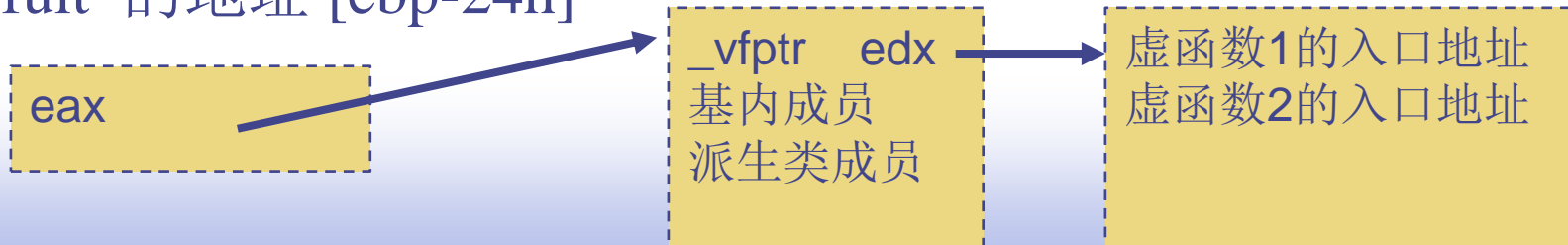
8.1 虚函数

```
fruit    *ptrfruit;  
ptrfruit->EatFruit();
```

```
ptrfruit->EatFruit();  
mov      eax,dword ptr [ebp-24h]  // ptrfruit的地址是 [ebp-24h]  
mov      edx,dword ptr [eax]      // 指向的对象中的4个字节  
mov      esi,esp  
mov      ecx,dword ptr [ebp-24h]  
mov      eax,dword ptr [edx+4]    // 函数的入口地址  
call     eax
```

Q: 有 virtual 时，是如何找到函数的入口地址的？

ptrfruit 的地址 [ebp-24h]





8.1 虚函数

Q: 将 `virtual` 去掉，如何找到函数的入口地址？

```
fruit    *ptrfruit;  
apple    myapple;  
ptrfruit = &myapple;
```

```
ptrfruit->EatFruit();
```

```
mov      ecx,dword ptr [ebp-24h]  
call     fruit::EatFruit (0EC1028h)
```

编译器看到 `ptrfruit` 是指向基类的，
`EatFruit` 又不是虚函数，不会到虚函数表找其入口地址，
因而调用的是基类的函数。





8.1 虚函数

正确理解 虚函数

```
fruit    *ptrfruit;  
apple    myapple;    ptrfruit = &myapple;  
myapple.EatFruit();    // 运行子类的函数  
myapple.fruit::EatFruit( ); // 运行基类的函数  
ptrfruit->EatFruit();
```

- 从子类的角度看，它有自己的 **EatFruit**函数，同时也有继承的父类的 **EatFruit**。
- 单纯的父类对象或者父类对象指针，调用的是父类的成员函数。
- 父类指针指向子类对象时，虚函数的功效才显现出来。挂羊头卖狗肉、表里不一
- **编译按定义的类型（表象）来执行检查；**
- 函数调用是根据虚函数表找入口地址，找到的是实际指向对象的函数地址（本质）





8.1 虚函数

实验

- 1、有虚函数时，对象的存储空间是如何布局的？
- 2、虚函数表中存放什么信息？
- 3、通过对象访问虚函数、非虚函数，编译生成的代码有无差别？
- 4、通过对象、对象指针或对象引用 访问虚函数，编译生成的代码有无差别？
- 5、通过对象、对象指针或对象引用 访问非虚函数，编译生成的代码有无差别？
- 6、在有继承关系时，派生类的虚函数表如何变迁？
- 7、基类指针（或引用）指向子类对象，能访问哪些函数？





8.1 虚函数

- 1、有虚函数时，对象存储空间的布局
- 2、虚函数表中存放的信息

```
class A {  
public :  
    int x;  
    A(int x) { A::x = x; }  
    virtual void vf1(int v) { cout << x+v<<endl; }  
    virtual void vf2(int v) { cout << 2 * x+v<<endl; }  
    void fa(int v) { cout << 3 * x+v<<endl; }  
};  
A a(5);           // sizeof (A) == 8
```

监视 1		
搜索(Ctrl+E) 🔍 ⏪ ⏩ 搜索深度: 3 ▼ ▼ A		
名称	值	类型
▲ a	{x=5 }	A
▲ _vfptr	0x00e29b34 {虚函数表分析.exe!void(* A::`vftable'[3])() {0x00e210b...	void * *
[0]	0x00e210b4 {虚函数表分析.exe!A::vf1(int)}	void *
[1]	0x00e2137f {虚函数表分析.exe!A::vf2(int)}	void *
x	5	int





8.1 虚函数

3、通过对象访问虚函数、非虚函数，编译生成的代码有无差别？

```
A  a(5);  
a. vf1(10);  
    push        0Ah  
    lea         ecx, [a]  
    call        A::vf1 (0E210B4h)  
a. vf2(10);  
    push        0Ah  
    lea         ecx, [a]  
    call        A::vf2 (0E2137Fh)  
a. fa(10);  
    push        0Ah  
    lea         ecx, [a]  
    call        A::fa (0E210BEh)
```





8.1 虚函数

4、通过对象、对象指针或对象引用 访问虚函数， 编译生成的代码有无差别？

```
A  a(5);  
a. vf1(10);  
    push    0Ah  
    lea     ecx, [a]  
    call    A::vf1 (0E210B4h)  
a. vf2(10);  
    push    0Ah  
    lea     ecx, [a]  
    call    A::vf2 (0E2137Fh)
```

```
A *pa =&a;  
pa->vf1(10);  
    push    0Ah  
    mov     eax, dword ptr [pa]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [pa]  
    mov     eax, dword ptr [edx]  
    call    eax  
pa->vf2(10);  
    push    0Ah  
    mov     eax, dword ptr [pa]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [pa]  
    mov     eax, dword ptr [edx+4]  
    call    eax
```





8.1 虚函数

4、通过对象、对象指针或对象引用 访问虚函数， 编译生成的代码有无差别？

```
A  a(5);  
a.vf1(10);  
    push    0Ah  
    lea     ecx, [a]  
    call    A::vf1 (0E210B4h)  
a.vf2(10);  
    push    0Ah  
    lea     ecx, [a]  
    call    A::vf2 (0E2137Fh)
```

```
A &ra =a;  
ra.vf1(10);  
    push    0Ah  
    mov     eax, dword ptr [ra]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [ra]  
    mov     eax, dword ptr [edx]  
    call    eax  
ra.vf2(10);  
    push    0Ah  
    mov     eax, dword ptr [ra]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [ra]  
    mov     eax, dword ptr [edx+4]  
    call    eax
```





8.1 虚函数

5、通过对象、对象指针或对象引用 访问非虚函数， 编译生成的代码有无差别？

```
A  a(5);  
a. fa(10);  
push      0Ah  
lea       ecx, [a]  
call      A::fa (0E210BEh)
```

```
A *pa =&a;  
pa->fa(10);  
push      0Ah  
mov       ecx, dword ptr [pa]  
call      A::fa (0E210BEh)
```

```
A &ra =a;  
ra. fa(10);  
push      0Ah  
mov       ecx, dword ptr [ra]  
call      A::fa (0E210BEh)
```



8.1 虚函数

6、在有继承关系时，派生类的虚函数表如何变迁？

```
class B : public A {
public:
    int y;
    B(int y) :A(2 * y) { B::y = y; }
    void vf1(int v) { cout << 4 * y + v << endl; }
    virtual void vf3() { cout << "new virtual func" << endl; }
};
B b(10);    // sizeof(B) ==12
```

▲ b	{y=10 }	B
▲ A	{x=20 }	A
▲ _vfptr	0x00e29b44 {虚函数表分析.exe!void(* B::`vftable'[4])() {0x00e2120...	void * *
[0]	0x00e2120d {虚函数表分析.exe!B::vf1(int)}	void *
[1]	0x00e2137f {虚函数表分析.exe!A::vf2(int)}	void *
x	20	int
y	10	int

8.1 虚函数

监视 1

搜索(Ctrl+E)



搜索深度: 3



A

名称	值	类型
▲ a	{x=5 }	A
▲ _vfptr	0x00e29b34 {虚函数表分析.exe!void(* A::`vftable'[3])() {0x00e210b...	void **
[0]	0x00e210b4 {虚函数表分析.exe!A::vf1(int)}	void *
[1]	0x00e2137f {虚函数表分析.exe!A::vf2(int)}	void *
x	5	int

▲ b	{y=10 }	B
▲ A	{x=20 }	A
▲ _vfptr	0x00e29b44 {虚函数表分析.exe!void(* B::`vftable'[4])() {0x00e2120...	void **
[0]	0x00e2120d {虚函数表分析.exe!B::vf1(int)}	void *
[1]	0x00e2137f {虚函数表分析.exe!A::vf2(int)}	void *
x	20	int
y	10	int

8.1 虚函数

比较：在对象 a 中与对象 b 中虚函数表的差异

❏ b._vfptr[0]	0x00ca1212 {虚函数表分析.exe!B::vf1(int)}
❏ b._vfptr[1]	0x00ca1389 {虚函数表分析.exe!A::vf2(int)}
❏ b._vfptr[2]	0x00ca12e4 {虚函数表分析.exe!B::vf3(void)}
❏ b._vfptr[3]	0x00000000
❏ a._vfptr[0]	0x00ca10b4 {虚函数表分析.exe!A::vf1(int)}
❏ a._vfptr[1]	0x00ca1389 {虚函数表分析.exe!A::vf2(int)}
❏ a._vfptr[2]	0x00000000

A类中有 2 个虚函数

B类中，存在同名的函数 vf1，地址覆盖
直接继承的函数 vf2，地址保持
新增的虚函数 vf3，新增地址



8.1 虚函数

通过对象 与 通过指针（引用）访问虚函数的差别

```
B  b(10);  
b.vf1(1);  
    push    1  
    lea     ecx, [b]  
    call    B::vf1 (0E2120Dh)  
        b.vf2(1);  
    push    1  
    lea     ecx, [b]  
    call    A::vf2 (0E2137Fh)  
        b.vf3( );  
    lea     ecx, [b]  
    call    B::vf3 (0E212DAh)
```

```
B  *pb = &b;  
pb->vf1(1);  
    push    1  
    mov     eax, dword ptr [pb]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [pb]  
    mov     eax, dword ptr [edx]  
    call    eax  
pb->vf2(1);  
    push    1  
    mov     eax, dword ptr [pb]  
    mov     edx, dword ptr [eax]  
    mov     ecx, dword ptr [pb]  
    mov     eax, dword ptr [edx+4]  
    call    eax
```





8.1 虚函数

7、基类指针（或引用）指向子类对象，能访问哪些函数？

```
A* p = &b;  
p->vf1(1);    // 虚函数表中 vf1的地址已变为 B类 vf1 的地址  
p->A::vf1(1); // 调用 A类的 vf1  
              // 直接调用A类函数，不通过虚函数表
```

```
p->vf2(2);  
p->vf3(); // error vf3 不是 A类的成员
```

通过基类指针/引用，只能访问基类的函数





8.1 虚函数

实验总结

- 1、有虚函数时，对象的存储空间是如何布局的？
- 2、虚函数表中存放什么信息？
- 3、通过对象访问虚函数、非虚函数，编译生成的代码有无差别？
- 4、通过对象、对象指针或对象引用 访问虚函数，编译生成的代码有无差别？
- 5、通过对象、对象指针或对象引用 访问非虚函数，编译生成的代码有无差别？
- 6、在有继承关系时，派生类的虚函数表如何变迁？
- 7、基类指针（引用）指向子类对象，能调用哪些函数？

- 1、增加了4个字节
虚函数表指针
- 2、虚函数的入口地址
- 3、无差别
- 4、有差别
实现细节.....
- 5、无差别
- 6、继承、覆盖
新增
- 7、编译按定义的类型来检查





8.1 虚函数

总结

基类指针（引用）指向子类对象，
在调用形式为访问基类的虚函数时，
实际访问的是子类中定义的同名函数。





8.1 虚函数

- 虚函数一般在基类的**public**部分（注意可访问性检查）。
- 在派生类定义取代型虚函数时，函数原型应和基类的虚函数相同。
- 在派生类，无论是否使用 **virtual** 保留字都将成为虚函数。
- 虚函数只有在具有继承关系的类层次中才需要表现多态。

```
fruit :          *fruitptr;  
    virtual void EatFruit();
```

```
Apple :          fruitptr=&o_apple;  
    void EatFruit( );    // 虚函数          fruitptr->EatFruit();  
    void EatFruit(int x); // 新增的成员函数
```





8.1 虚函数

- 虚函数一般在**基类**的**public**或**protected**部分。
- 在派生类**定义取代型**虚函数时，函数原型应和基类的虚函数相同。
- 在派生类，无论是否使用 **virtual** 保留字都将成为虚函数。
- 虚函数只有在具有继承关系的类层次中才需要表现多态。

```
fruit :          *fruitptr;  
              virtual void EatFruit();
```

```
Apple :          fruitptr=&ptrapple;  
                void EatFruit( );    // 虚函数          fruitptr->EatFruit();  
                void EatFruit(int x); // 新增的成员函数
```





8.1 虚函数

- 构造函数不能定义为虚函数

构造对象时类型是确定的，不需根据类型不同表现多态行为。

- 析构造函数可定义为虚函数

派生类的析构造函数可通过父类指针、引用或**delete**调用。





8.1 虚函数

函数重载

- 函数名称相同，参数不同；
- 可以是成员函数和非成员函数；
- 以传递参数的差别，确定调用哪一个函数。

虚函数

- 函数名称相同、参数、返回值相同；
（隐含的第一参数是不同的）
- 只能是成员函数；
- 根据对象的不同，去调用不同类的虚函数。





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。
- 在使用delete运算符删除一个对象时，
为了保证执行的析构函数就是该对象自己的析构函数，
应将析构函数定义为虚析构函数。



8.2 虚析构函数

```
class STACK{
    int *e, p, c;
public:
    virtual int getp( ){ return p; }
    virtual int push(int f){ return p<c?(e[p++]=f,1): 0; }
    virtual int pop (int&f){ return p>0?(f=e[--p],1): 0; }
    STACK(int m): e(new int[m]), c(e?m:0), p(0){ }
    virtual ~STACK( ){ if(e){ delete e; e=0; c=0; p=0;}}
};

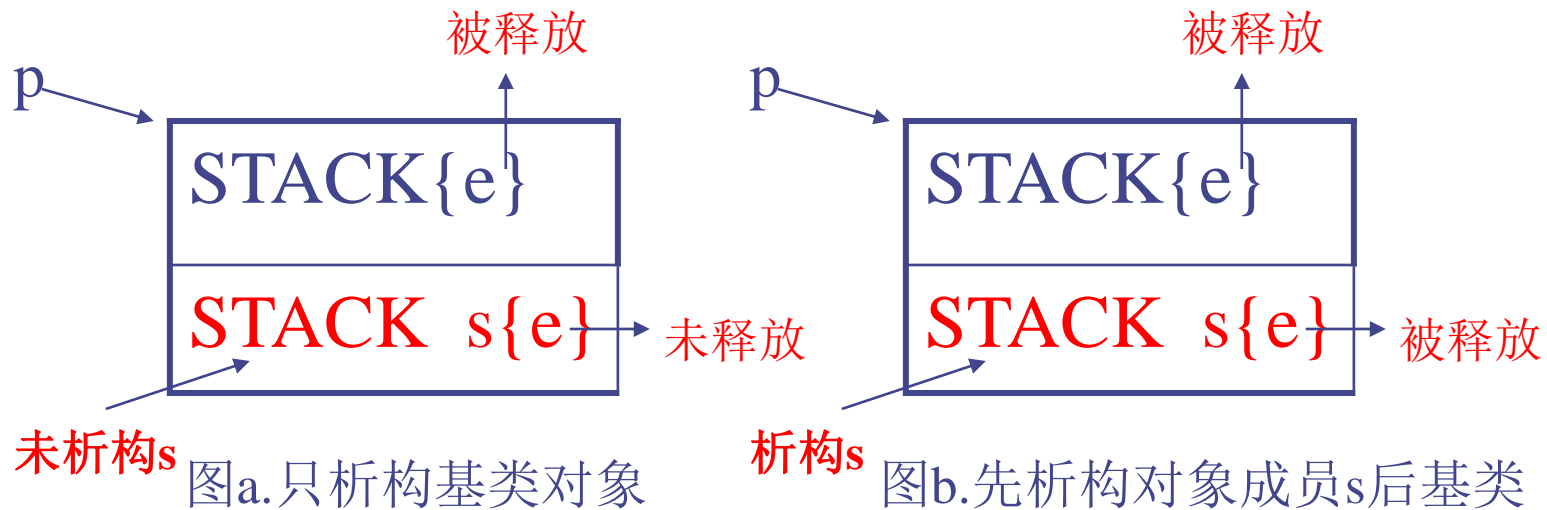
class QUEUE: public STACK{//公有派生，基类和派生类构成父子关系
    STACK s;
public:
    virtual int enter(int f) { return s.getp( ) ? push(f): s.push(f); }
    virtual int leave(int&f) { if (!s.getp( )) while(pop(f)) s.push(f); return
        s.pop(f); }
    QUEUE(int m): STACK(m), s(m){ }
    ~QUEUE( ){ }
};

void main(void){ STACK *p=new QUEUE(9); delete p; }
```



8.2 虚析构函数

- ◆ 如果`~STACK`没定义为虚函数，则`delete p`调用析构函数`~STACK`，只释放基类对象成员`e`占用的内存，未析构对象成员`s`
- ◆ 如果`~STACK`定义为虚函数，则`delete p`调用析构函数`~QUEUE`，把`QUEUE(9)`当作`QUEUE`对象析构(如图b)。





8.3 类的引用

类名 $\&p$ = 类的一个对象;





8.3 类的引用

```
#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
```





8.3 类的引用

```
void main(void)
{
    A a(1), b(2);           //自动调用构造函数构造a、b
    A &p=a;                 //p本身不用负责构造和析构a
    A &q=*new A(3);         //q有址引用new生成的无名对象
    A &r=p;                 //r有址引用p所引用的对象a
    cout<<"CALL g(b)\n";
    g(b);                  //使用同类型的传统左值作为实参调用函数g()
    h();                   //使用无址右值A(5)作为实参初始化形参a
    h(A(4));               //使用无址右值A(4)作为实参初始化形参a
    cout<<"main return\n";
    delete &q;             // 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
```





8.3 类的引用

- 当类包含指针成员时，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，一般应自定义：

A()

A(A&&) noexcept

A(const A&)

A& operator=(const A&)

A& operator=(A&&) noexcept

~A()





8.3 类的引用

```
class QUEUE :
```

```
    QUEUE(int m); //初始化队列：最多申请m个元素
```

```
    QUEUE(const QUEUE& q); //用q深拷贝初始化队列
```

```
    QUEUE(QUEUE&& q) noexcept; //用q移动初始化队列
```

```
        //深拷贝赋值并返回被赋值队列
```

```
    virtual QUEUE& operator=(const QUEUE& q);
```

```
        //移动赋值并返回被赋值队列
```

```
    virtual QUEUE& operator=(QUEUE&& q) noexcept;
```

```
    virtual ~QUEUE();
```





8.3 类的引用

- $A(A\&\&)$ 、 $A\& \text{operator}=(A\&\&)$ 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。





8.3 类的引用

- $A(A\&\&)$ 、 $A\& \text{operator}=(A\&\&)$ 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。





8.3 类的引用

```
STACK(int m);           //初始化栈：最多存放2m-2个元素
STACK(const STACK& s);   //用栈s深拷贝初始化栈
STACK(STACK&& s)noexcept; //用栈s移动拷贝初始化栈

STACK& operator=(const STACK& s);   //深拷贝赋值并返回被赋值栈
STACK& operator=(STACK&& s)noexcept; //移动赋值并返回被赋值栈

~STACK()noexcept;           //销毁栈

STACK::STACK(int m):QUEUE(m), q(m) { }

STACK::STACK(const STACK& s):QUEUE(s), q(s.q) { }

STACK::STACK(STACK&& s)noexcept
    : QUEUE((QUEUE &&)s), q((QUEUE &&)s.q) { }
```





8.4 抽象类

virtual 函数类型 函数名(形式参数表) = 0;

纯虚函数没有函数体。

函数体由派生类给出。

含有纯虚函数的类：**抽象类**。

设计抽象类的目的：

建立一个公有的接口，动态的使用它的成员函数。





8.4 抽象类

- ◆ 抽象类常用作派生类的基类，不能有具体的对象（不能有抽象类定义的变量、常量或new产生）。
- ◆ 如果派生类**没有重新定义**该基类的虚函数，或者定义了基类所没有**新的纯虚函数**，则派生类也会成为抽象类。
- ◆ 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**全部重新定义了函数体，则该派生类就会成为具体类。





8.4 抽象类

```
class A {  
public:  
    virtual void f1()=0;  
    virtual void f2()=0;  
    void f3() {cout<<"A3"<<endl;}  
};  
void A::f1() {cout<<"A1 "<<endl;}  
void A::f2() {cout<<"A2 "<<endl;}
```

A 为抽象类

- 可以有非虚函数 f3
- 尽管在A的体外定义了f1, f2, 但A仍是抽象类

// A a1; error 抽象类不能定义任何对象





8.4 抽象类

```
class B : public A {  
private:  
    void f2() {  
        this->A::f2();  
        cout<<"B2 "<<endl;  
    }  
};
```

B 为抽象类

- B 中未重新定义 f1
- B 中重新定义了 f2

A:
public: f1、f2、f3

B:
public:
 A::f1、A::f2、A::f3
Private:
 f2

B:
public:
 f1、A::f2、f3
Private:
 f2





8.4 抽象类

```
class C:public B {  
private:  
    void f1() {cout<<"C1 "<<endl;}  
public:  
    void f4() {cout<<"f4 "<<endl;}  
};  
void main(void)  
{    C c;  
    A *p=(A *)&c;  
    p->f1();    // 显示 C1  
    p->f2();    // 显示 A2、 B2  
    p->f3();    // 显示 A3  
    p->f4();    // error f4 不是A类的成员  
}
```

A:
public: f1、 f2、 f3

B: public:
f1、 A::f2、 f3
private: f2

C: public:
f1、 A::f2、 f3
f4
private: f1





8.4 抽象类

```
class C: public B {  
private: void f1() {cout<<"C1 "<<endl;}  
public: void f4() {cout<<"f4 "<<endl;}  
};
```

```
void main(void)  
{ C c;
```

A a1; // A 无法实例化抽象类

B b1; // B 无法实例化抽象类

A *a2;

a2= new A; // A 无法实例化抽象类

c.f1(); // 无法访问私有成员（在C类中申明）

c.f2(); // 无法访问私有成员（在B类中申明）

```
} // c类中未定义f2, B中定义了 f2, 同时继承了 A 类的f2,  
// 按作用域小优先的原则, c.f2() 等价于 c.B::f2()
```

```
C: public:  
f1、A::f2、 f3  
f4  
private: f1
```





8.4 抽象类

```
class C:public B {  
private:  void f1() {cout<<"C1 "<<endl;}  
public:   void f4() {cout<<"f4 "<<endl;}  
};  
void main(void)  
{   C c;  
    c.f1();           // 无法访问私有成员（在C类中申明）  
    c.A::f1();        // f1 在 A是public, A ->public 派生 B  
                      //                B ->public 派生 C  
    c.f4();  
    c.f2(); // 等同于 c.B::f2(); 无法访问私有成员  
    c.A::f2(); //显示 A2  
}
```





8.5 虚函数友元与晚期绑定

虚函数友元

- 一个函数是虚函数
- 该虚函数是另外一个类的友元
- 友元关系不能传递
- 友元关系不能继承
- 友元特性与是否是虚函数（或者实例函数）无关



8.5 虚函数友元与晚期绑定

晚期绑定

- 编译程序为有虚函数的类创建一个虚函数入口地址表VFT,
- 表首地址存放在对象的起始单元中。
- 当对象调用虚函数时, 通过其起始单元得到VFT首址, 动态绑定到相应的函数成员。





8.5 虚函数友元与晚期绑定

设基类A和派生类B都有虚函数，对应的虚函数入口地址表分别为VFT_A和VFT_B。

VFT_A

在 A类中声明的虚函数

VFT_B

在 B类中声明的虚函数

A类中未被取代的虚函数



8.5 虚函数友元与晚期绑定

派生类对象**b**构造阶段

- 先将VFT_A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时，**A**类对象调用的虚函数与VFT_A绑定，执行的虚函数将是**A**类的函数；
- 在**B**类构造函数的函数体执行前，将VFT_B的首地址存放到**b**的起始单元，绑定和执行的将是**B**类的函数。
- 如果**B**类没有定义这样的函数，根据面向对象的作用域，将调用基类**A**的相同原型的函数。





8.5 虚函数友元与晚期绑定

生存阶段:

b的起始单元指向VFTB，绑定和执行的将是B类的函数。
如果B类没有定义这样的函数，根据面向对象的作用域，
将调用基类A的相同原型的函数。

析构阶段:

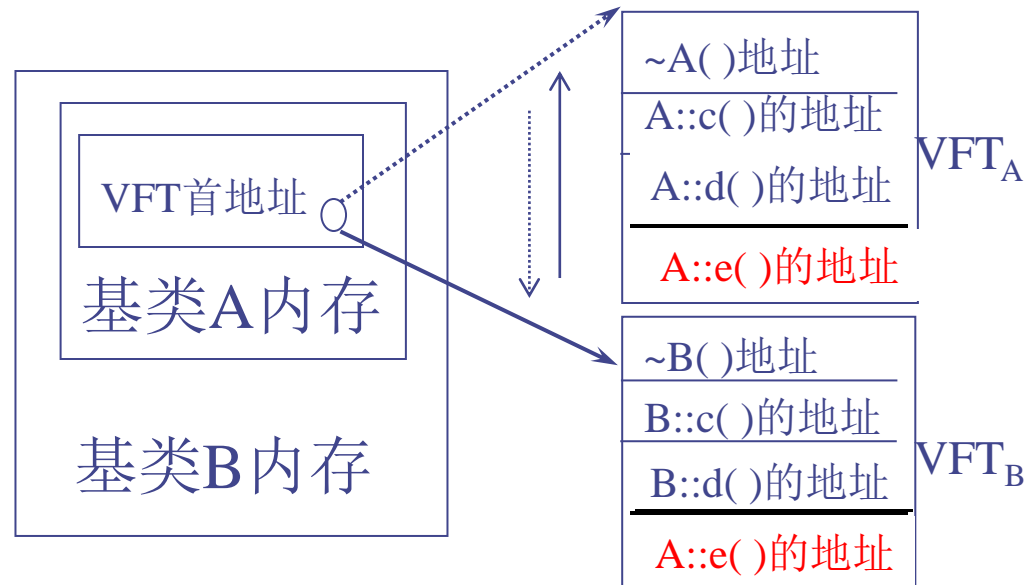
b的起始单元指向VFTB，绑定和执行的方式同上；在b的
析构函数执行完后、基类的析构函数执行前，将VFTA首
地址存放到b的起始单元，此后绑定和执行的将是A的函
数。如果A类没有定义这样的函数，根据面向对象的作用
域，将调用基类A的相同原型的函数。





8.5 虚函数友元与晚期绑定

```
#include <iostream.h>
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();//virtual可省
};
```



```
void main(void){ B b; }
输出结果（先构造的后析构：像栈）
Construct A
Construct B
Deconstruct B
Deconstruct A
```



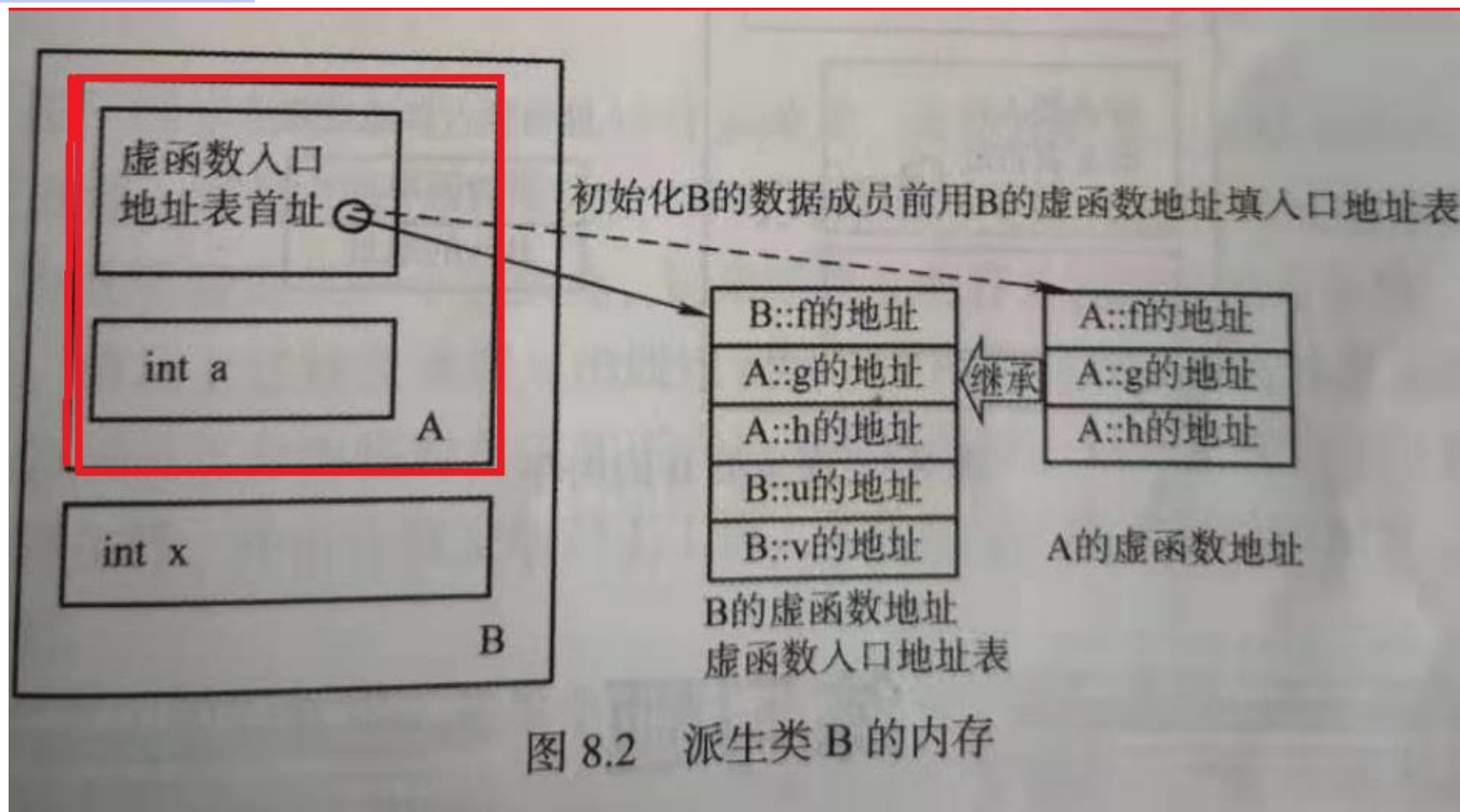


8.6 有虚函数时的内存布局

- ▶ 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。
- ▶ 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

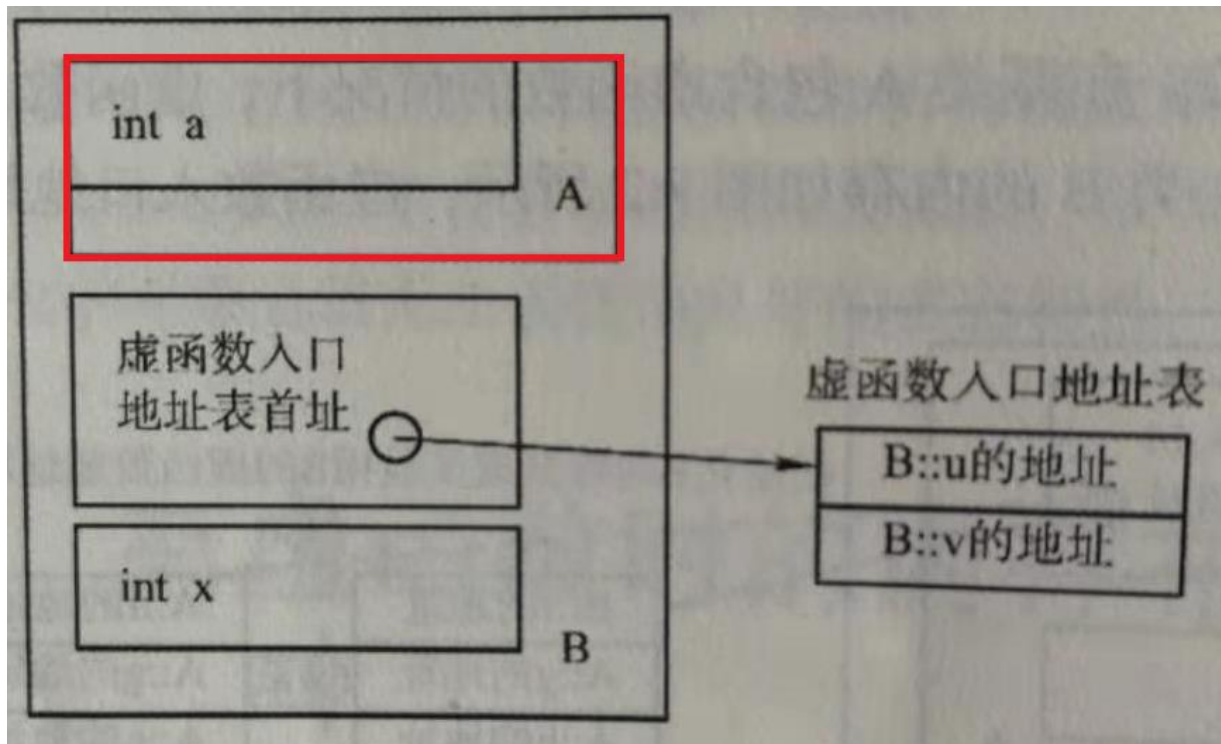


8.6 有虚函数时的内存布局



基类有虚函数时，派生类的内存布局

8.6 有虚函数时的内存布局



基类无虚函数时，派生类的内存布局



总结

- 多态性
- 虚函数
- 纯虚函数和抽象类
- 运行时多态的实现机理



作业 8.8

学校人员有学生、教师、职员等，各种人员的信息不同。

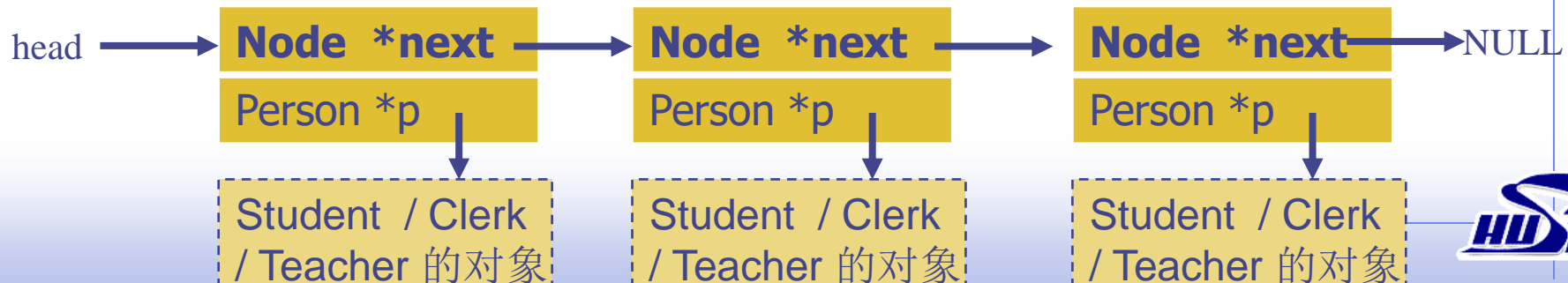
将不同人员的信息链接在一个表上，形成异质链表

实现人员的新增、退出、显示等功能

即实现链表结点的插入、删除、输出信息等工作。

作业 8.8 设计建议

- 建立一个基类：**Person**，包含学生、教师、职员的信息，如姓名、年龄、社会保险号等；
- 从基类派生出 学生 **Student**、教师 **Teacher**、职员类 **Clerk**，在各类中增加各自的新成员；
- 定义一个链表结点类，含有 **Person** 的指针成员，结点的下一个成员。





作业 8.8 设计建议

- 在 **Person** 中，提供信息输出的虚函数，并实现基本信息的输出；在学生、教师、职员等中，实现信息输出函数时，应调用基类的相应函数；
- 使用循环的方法，显示链表中各结点对应的人员信息；
- 在链表结点类中，有 增加链表结点、删除链表结点、显示链表结点对应人员信息的成员函数。
- 注意空间的申请和释放，防止内存泄露。

