

02.多态

1. 多态的概念

多态(polymorphism)的概念：通俗来说，就是多种形态。多态分为编译时多态(静态多态)和运行时多态(动态多态)，这里我们重点讲运行时多态，编译时多态(静态多态)和运行时多态(动态多态)。编译时多态(静态多态)主要就是前面讲的函数重载和函数模板，他们传不同类型的参数就可以调用不同的函数，通过参数不同达到多种形态，之所以叫编译时多态，是因为他们实参传给形参的参数匹配是在编译时完成的，我们把编译时一般归为静态，运行时归为动态。

运行时多态，具体点就是去完成某个行为(函数)，可以传不同的对象就会完成不同的行为，就达到多种形态。比如买票这个行为，当普通人买票时，是全价买票；学生买票时，是优惠买票(5折或75折)；军人买票时是优先买票。再比如，同样是动物叫的一个行为(函数)，传猫对象过去，就是"喵"，传狗对象过去，就是"汪汪"。



准大一新生

如何购买学生票

学生票优惠说明：

火车硬座：5折
火车硬卧：减去该车次硬座票的一半价格
(优惠价=原价-5折硬座价)
高铁(二等座)：75折
购票方法一：线下购票
凭录取通知书和本人身份证到人工售票口购票。
购票方法二：线上购票



通过铁路12306网站或者铁路12306 APP。
线上购票。(详细步骤下一页)

2. 多态的定义及实现

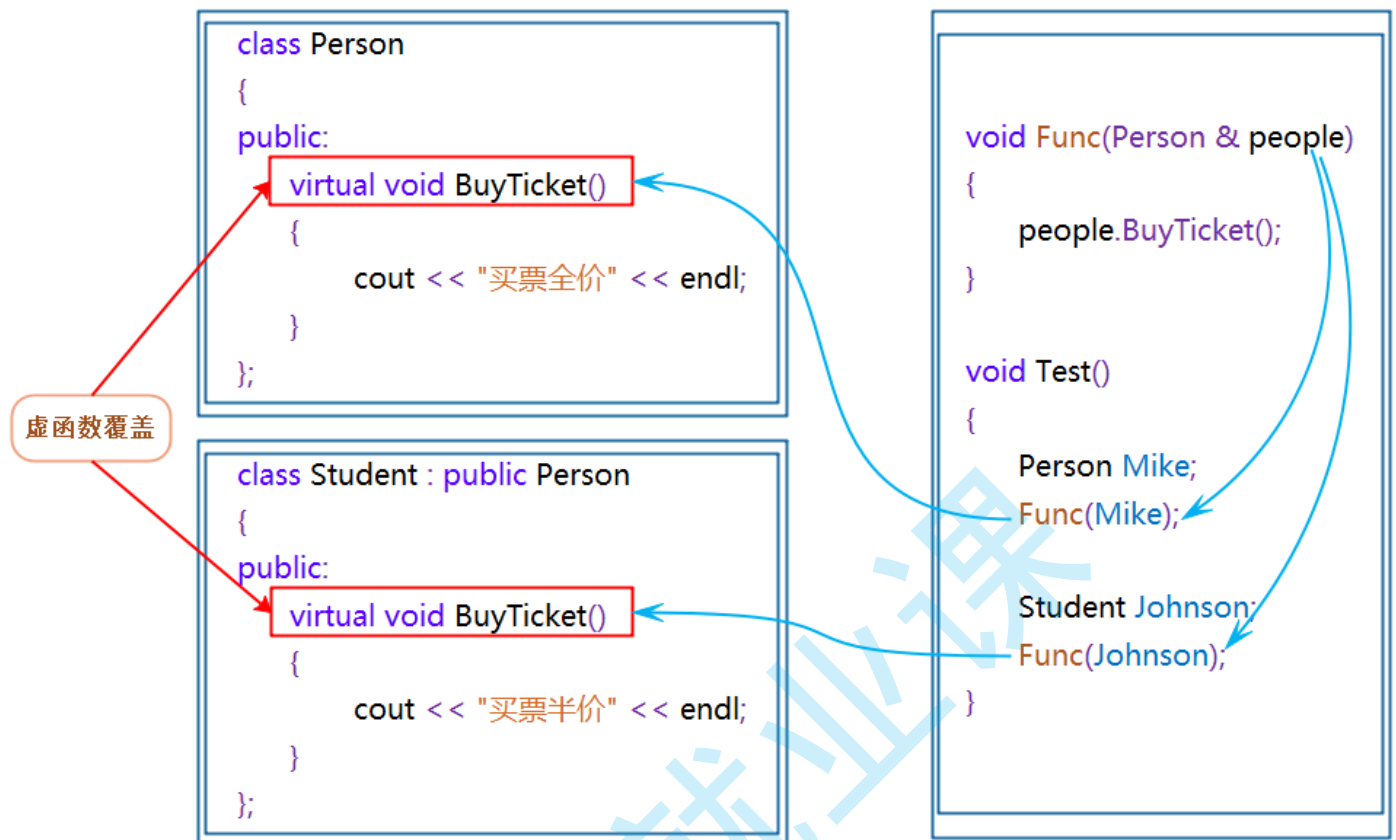
2.1 多态的构成条件

多态是一个继承关系的下的类对象，去调用同一函数，产生了不同的行为。比如Student继承了Person。Person对象买票全价，Student对象优惠买票。

2.1.1 实现多态还有两个必须重要条件：

- 必须是基类的指针或者引用调用虚函数
- 被调用的函数必须是虚函数，并且完成了虚函数重写/覆盖。

说明：要实现多态效果，第一必须是基类的指针或引用，因为只有基类的指针或引用才能既指向基类对象又指向派生类对象；第二派生类必须对基类的虚函数完成重写/覆盖，重写或者覆盖了，基类和派生类之间才能有不同函数，多态的不同形态效果才能达到。



2.1.2 虚函数

类成员函数前面加virtual修饰，那么这个成员函数被称为虚函数。注意非成员函数不能加virtual修饰。

```
1 class Person
2 {
3 public:
4     virtual void BuyTicket() { cout << "买票-全价" << endl; }
5 };
```

2.1.3 虚函数的重写/覆盖

虚函数的重写/覆盖：派生类中有一个跟基类完全相同的虚函数(即派生类虚函数与基类虚函数的返回值类型、函数名字、参数列表完全相同)，称派生类的虚函数重写了基类的虚函数。

注意：在重写基类虚函数时，派生类的虚函数在不加virtual关键字时，虽然也可以构成重写(因为继承后基类的虚函数被继承下来了在派生类依旧保持虚函数属性)，但是该种写法不是很规范，不建议这样使用，不过在考试选择题中，经常会故意买这个坑，让你判断是否构成多态。

```

1 class Person {
2 public:
3     virtual void BuyTicket() { cout << "买票-全价" << endl; }
4 };
5
6 class Student : public Person {
7 public:
8     virtual void BuyTicket() { cout << "买票-打折" << endl; }
9 };
10
11 void Func(Person* ptr)
12 {
13     // 这里可以看到虽然都是Person指针Ptr在调用BuyTicket
14     // 但是跟ptr没关系，而是由ptr指向的对象决定的。
15     ptr->BuyTicket();
16 }
17
18 int main()
19 {
20     Person ps;
21     Student st;
22
23     Func(&ps);
24     Func(&st);
25
26     return 0;
27 }

```

```

1 class Animal
2 {
3 public:
4     virtual void talk() const
5     {}
6 };
7
8 class Dog : public Animal
9 {
10 public:
11     virtual void talk() const
12     {
13         std::cout << "汪汪" << std::endl;
14     }
15 };
16
17 class Cat : public Animal

```

```

18 {
19 public:
20     virtual void talk() const
21     {
22         std::cout << "(>^ω^<)喵" << std::endl;
23     }
24 };
25
26 void letsHear(const Animal& animal)
27 {
28     animal.talk();
29 }
30
31 int main()
32 {
33     Cat cat;
34     Dog dog;
35
36     letsHear(cat);
37     letsHear(dog);
38
39     return 0;
40 }

```

2.1.4 多态场景的一个选择题

以下程序输出结果是什么（）

A: A->0 B: B->1 C: A->1 D: B->0 E: 编译出错 F: 以上都不正确

```

1     class A
2     {
3     public:
4         virtual void func(int val = 1){ std::cout<<"A->"<< val <<std::endl;}
5         virtual void test(){ func();}
6     };
7
8     class B : public A
9     {
10    public:
11        void func(int val = 0){ std::cout<<"B->"<< val <<std::endl; }
12    };
13
14    int main(int argc ,char* argv[])
15    {
16        B*p = new B;

```

```
17     p->test();
18     return 0;
19 }
```

2.1.5 虚函数重写的一些其他问题

- 协变(了解)

派生类重写基类虚函数时，与基类虚函数返回值类型不同。即基类虚函数返回基类对象的指针或者引用，派生类虚函数返回派生类对象的指针或者引用时，称为协变。协变的实际意义并不大，所以我们了解一下即可。

```
1  class A {};
2  class B : public A {};
3
4  class Person {
5  public:
6      virtual A* BuyTicket()
7      {
8          cout << "买票-全价" << endl;
9          return nullptr;
10     }
11 };
12
13 class Student : public Person {
14 public:
15     virtual B* BuyTicket()
16     {
17         cout << "买票-打折" << endl;
18         return nullptr;
19     }
20 };
21
22 void Func(Person* ptr)
23 {
24     ptr->BuyTicket();
25 }
26
27 int main()
28 {
29     Person ps;
30     Student st;
31
32     Func(&ps);
33     Func(&st);
34 }
```

```
35     return 0;
36 }
```

• 析构函数的重写

基类的析构函数为虚函数，此时派生类析构函数只要定义，无论是否加virtual关键字，都与基类的析构函数构成重写，虽然基类与派生类析构函数名字不同看起来不符合重写的规则，实际上编译器对析构函数的名称做了特殊处理，编译后析构函数的名称统一处理成destructor，所以基类的析构函数加了virtual修饰，派生类的析构函数就构成重写。

下面的代码我们可以看到，如果~A()，不加virtual，那么delete p2时只调用的A的析构函数，没有调用B的析构函数，就会导致内存泄漏问题，因为~B()中在释放资源。

注意：这个问题面试中经常考察，大家一定要结合类似下面的样例才能讲清楚，为什么基类中的析构函数建议设计为虚函数。

```
1  class A
2  {
3  public:
4      virtual ~A()
5      {
6          cout << "~A()" << endl;
7      }
8  };
9
10 class B : public A {
11 public:
12     ~B()
13     {
14         cout << "~B()->delete:"<<_p<< endl;
15         delete _p;
16     }
17 protected:
18     int* _p = new int[10];
19 };
20
21 // 只有派生类Student的析构函数重写了Person的析构函数，下面的delete对象调用析构函数，才能
    构成多态，才能保证p1和p2指向的对象正确的调用析构函数。
22 int main()
23 {
24     A* p1 = new A;
25     A* p2 = new B;
26
27     delete p1;
28     delete p2;
29 }
```

```
30     return 0;
31 }
```

2.1.6 override 和 final关键字

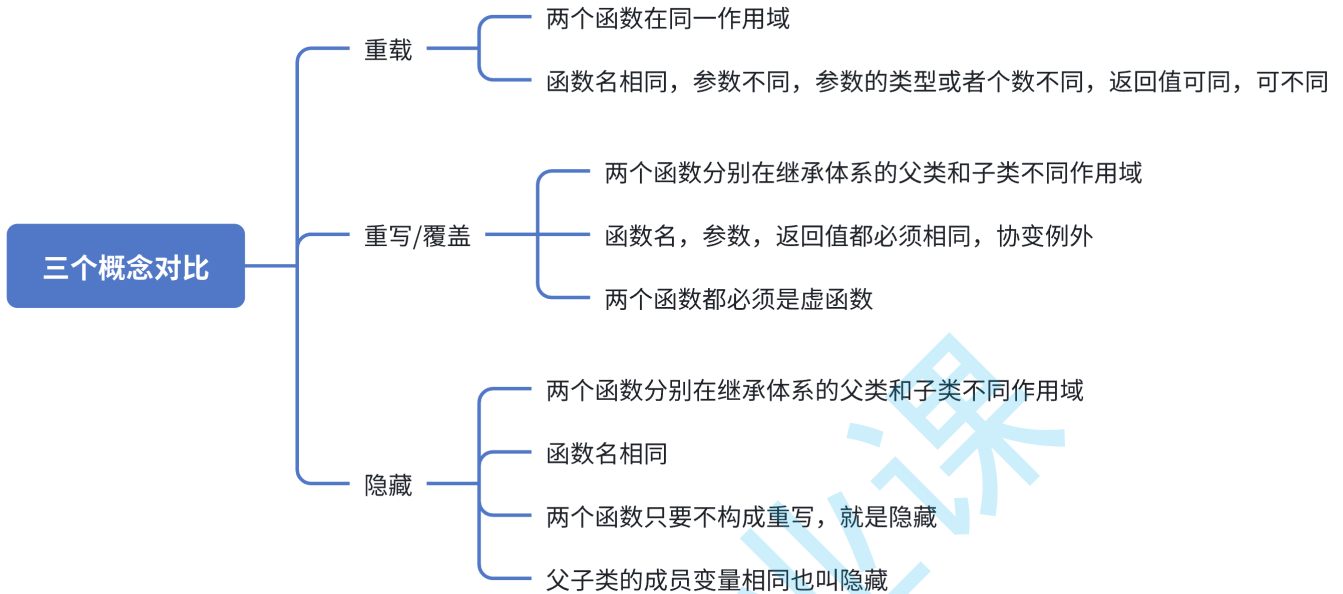
从上面可以看出，C++对虚函数重写的要求比较严格，但是有些情况下由于疏忽，比如函数名写错参数写错等导致无法构成重写，而这种错误在编译期间是不会报出的，只有在程序运行时没有得到预期结果才来debug会得不偿失，因此C++11提供了override，可以帮助用户检测是否重写。如果我们不想让派生类重写这个虚函数，那么可以用final去修饰。

```
1 // error C3668: "Benz::Drive": 包含重写说明符"override"的方法没有重写任何基类方法
2 class Car {
3 public:
4     virtual void Dirve()
5     {}
6 };
7
8 class Benz :public Car {
9 public:
10     virtual void Drive() override { cout << "Benz-舒适" << endl; }
11 };
12
13 int main()
14 {
15     return 0;
16 }
```

```
1 // error C3248: "Car::Drive": 声明为"final"的函数无法被"Benz::Drive"重写
2 class Car
3 {
4 public:
5     virtual void Drive() final {}
6 };
7
8 class Benz :public Car
9 {
10 public:
11     virtual void Drive() { cout << "Benz-舒适" << endl; }
12 };
13
14 int main()
15 {
16     return 0;
17 }
```

2.1.7 重载/重写/隐藏的对比

注意：这个概念对比经常考，大家得理解记忆一下



3. 纯虚函数和抽象类

在虚函数的后面写上 `=0`，则这个函数为纯虚函数，纯虚函数不需要定义实现(实现没啥意义因为要被派生类重写，但是语法上可以实现)，只要声明即可。包含纯虚函数的类叫做抽象类，抽象类不能实例化出对象，如果派生类继承后不重写纯虚函数，那么派生类也是抽象类。纯虚函数某种程度上强制了派生类重写虚函数，因为不重写实例化不出对象。

```

1 class Car
2 {
3 public:
4     virtual void Drive() = 0;
5 };
6
7 class Benz :public Car
8 {
9 public:
10     virtual void Drive()
11     {
12         cout << "Benz-舒适" << endl;
13     }
14 };
15

```



```

16 class BMW :public Car
17 {
18 public:
19     virtual void Drive()
20     {
21         cout << "BMW-操控" << endl;
22     }
23 };
24
25 int main()
26 {
27     // 编译报错: error C2259: "Car": 无法实例化抽象类
28     Car car;
29
30     Car* pBenz = new Benz;
31     pBenz->Drive();
32
33     Car* pBMW = new BMW;
34     pBMW->Drive();
35
36     return 0;
37 }

```

4. 多态的原理

4.1 虚函数表指针

下面编译为32位程序的运行结果是什么 ()

A. 编译报错 B. 运行报错 C. 8 D. 12

```

1 class Base
2 {
3 public:
4     virtual void Func1()
5     {
6         cout << "Func1()" << endl;
7     }
8 protected:
9     int _b = 1;
10    char _ch = 'x';
11 };
12
13 int main()
14 {








```

```

15     Base b;
16     cout << sizeof(b) << endl;
17
18     return 0;
19 }

```

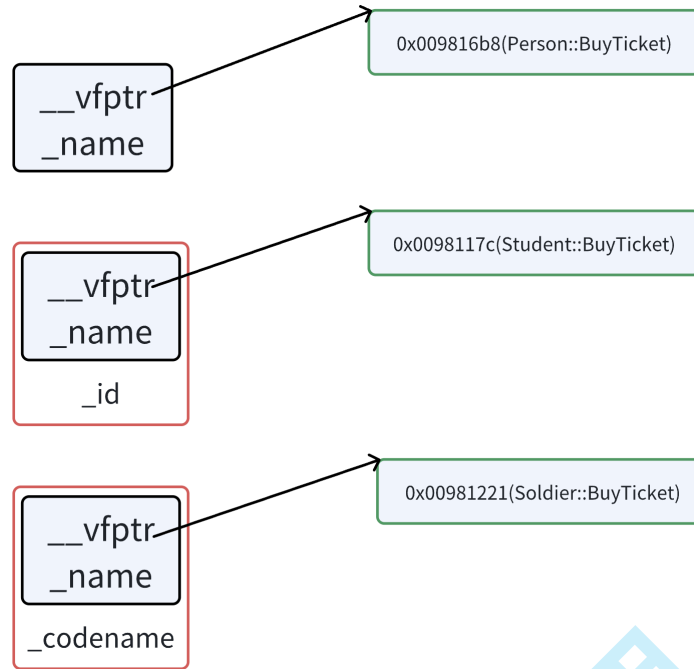
上面题目运行结果12bytes，除了_b和_ch成员，还多一个__vfptr放在对象的前面(注意有些平台可能会放到对象的最后面，这个跟平台有关)，对象中的这个指针我们叫做虚函数表指针(v代表virtual，f代表function)。一个含有虚函数的类中都至少都有一个虚函数表指针，因为一个类所有虚函数的地址要被放到这个类对象的虚函数表中，虚函数表也简称虚表。

监视 1	
搜索(Ctrl+E)  < > 搜索深度: 3  	
名称	值
▾  b	{_b=1 _ch=120 'x' }
▸  __vfptr	0x00007ff73708acc0 {6-4.exe!void(* Base::`vftable'[2])()
 _b	1
 _ch	120 'x'

4.2 多态的原理

4.2.1 多态是如何实现的

从底层的角度Func函数中ptr->BuyTicket()，是如何作为ptr指向Person对象调用Person::BuyTicket，ptr指向Student对象调用Student::BuyTicket的呢？通过下图我们可以看到，满足多态条件后，底层不再是编译时通过调用对象确定函数的地址，而是运行时到指向的对象的虚表中确定对应的虚函数的地址，这样就实现了指针或引用指向基类就调用基类的虚函数，指向派生类就调用派生类对应的虚函数。第一张图，ptr指向的Person对象，调用的是Person的虚函数；第二张图，ptr指向的Student对象，调用的是Student的虚函数。



```
void Func(Person* ptr)
{
    ptr->BuyTicket();
}
```

// 这里可以看到虽然都是Person指针Ptr在调用BuyTicket
// 但是跟ptr没关系，而是由ptr指向的对象决定的。

```
int main()
{
    Person ps;
    Student st;

    Func(&ps);
    Func(&st);

    return 0;
}
```

监视 1	
名称	值
ps	{ _name="张三" _age=18 }
__vfptr	0x00007ff6820af2c0 {6-4.exe!void(* Person::vftable[2])() {0x00007ff6820a1055
[0]	0x00007ff6820a1055 {6-4.exe!Person::BuyTicket(void)}
_name	"张三"
_age	18
st	{ _id=1 }
Person	{ _name="张三" _age=18 }
__vfptr	0x00007ff6820af2e8 {6-4.exe!void(* Student::vftable[2])() {0x00007ff6820a160
[0]	0x00007ff6820a1609 {6-4.exe!Student::BuyTicket(void)}
_name	"张三"
_age	18
_id	1

买票-全价
买票-打折

```

void Func(Person* ptr)
{
    // 这里可以看到虽然都是Person指针Ptr在调用BuyTicket
    // 但是跟ptr没关系, 而是由ptr指向的对象决定的。
    ptr->BuyTicket();
}

```

```

int main()
{
    Person ps;
    Student st;

    Func(&ps);
    Func(&st);

    return 0; 已用时间 <= 1ms
}

```

监视 1

搜索(Ctrl+E) 搜索深度: 3

名称	值
ps	{_name="张三" _age=18 }
__vfptr	0x00007ff6820af2c0 {6-4.exe!void(* Person::vftable[2])()} {0x00007ff6820a1055
[0]	0x00007ff6820a1055 {6-4.exe!Person::BuyTicket(void)}
_name	"张三"
_age	18
st	{_id=1 }
Person	{_name="张三" _age=18 }
__vfptr	0x00007ff6820af2e8 {6-4.exe!void(* Student::vftable[2])()} {0x00007ff6820a160
[0]	0x00007ff6820a1609 {6-4.exe!Student::BuyTicket(void)}
_name	"张三"
_age	18
_id	1

```

1 class Person {
2 public:
3     virtual void BuyTicket() { cout << "买票-全价" << endl; }
4 private:
5     string _name;
6 };
7
8 class Student : public Person {
9 public:
10     virtual void BuyTicket() { cout << "买票-打折" << endl; }
11 private:
12     string _id;
13 };
14
15 class Soldier: public Person {
16 public:
17     virtual void BuyTicket() { cout << "买票-优先" << endl; }
18 private:
19     string _codename;
20 };
21
22 void Func(Person* ptr)
23 {
24     // 这里可以看到虽然都是Person指针Ptr在调用BuyTicket
25     // 但是跟ptr没关系, 而是由ptr指向的对象决定的。
26     ptr->BuyTicket();
27 }
28
29 int main()

```

```

30 {
31     // 其次多态不仅仅发生在派生类对象之间，多个派生类继承基类，重写虚函数后
32     // 多态也会发生在多个派生类之间。
33     Person ps;
34     Student st;
35     Soldier sr;
36
37     Func(&ps);
38     Func(&st);
39     Func(&sr);
40
41     return 0;
42 }

```

4.2.2 动态绑定与静态绑定

- 对不满足多态条件(指针或者引用+调用虚函数)的函数调用是在编译时绑定，也就是编译时确定调用函数的地址，叫做静态绑定。
- 满足多态条件的函数调用是在运行时绑定，也就是在运行时到指向对象的虚函数表中找到调用函数的地址，也就做动态绑定。

```

1     // ptr是指针+BuyTicket是虚函数满足多态条件。
2     // 这里就是动态绑定，编译在运行时到ptr指向对象的虚函数表中确定调用函数地址
3     ptr->BuyTicket();
4 00EF2001 mov     eax,dword ptr [ptr]
5 00EF2004 mov     edx,dword ptr [eax]
6 00EF2006 mov     esi,esp
7 00EF2008 mov     ecx,dword ptr [ptr]
8 00EF200B mov     eax,dword ptr [edx]
9 00EF200D call    eax
10
11     // BuyTicket不是虚函数，不满足多态条件。
12     // 这里就是静态绑定，编译器直接确定调用函数地址
13     ptr->BuyTicket();
14 00EA2C91 mov     ecx,dword ptr [ptr]
15 00EA2C94 call    Student::Student (0EA153Ch)

```

4.2.3 虚函数表

- 基类对象的虚函数表中存放基类所有虚函数的地址。同类型的对象共用同一张虚表，不同类型的对象各自有独立的虚表，所以基类和派生类有各自独立的虚表。

- 派生类由两部分构成，继承下来的基类和自己的成员，一般情况下，继承下来的基类中有虚函数表指针，自己就不会再生成虚函数表指针。但是要注意的这里继承下来的基类部分虚函数表指针和基类对象的虚函数表指针不是同一个，就像基类对象的成员和派生类对象中的基类对象成员也独立的。
- 派生类中重写的基类的虚函数，派生类的虚函数表中对应的虚函数就会被覆盖成派生类重写的虚函数地址。
- 派生类的虚函数表中包含，(1)基类的虚函数地址，(2)派生类重写的虚函数地址完成覆盖，派生类自己的虚函数地址三个部分。
- 虚函数表本质是一个存虚函数指针的指针数组，一般情况这个数组最后面放了一个0x00000000标记。(这个C++并没有进行规定，各个编译器自行定义的，vs系列编译器会再后面放个0x00000000标记，g++系列编译不会放)
- 虚函数存在哪的？虚函数和普通函数一样的，编译好后是一段指令，都是存在代码段的，只是虚函数的地址又存到了虚表中。
- 虚函数表存在哪的？这个问题严格说并没有标准答案C++标准并没有规定，我们写下面的代码可以对比验证一下。vs下是存在代码段(常量区)

```
int main()
{
    Base b;
    Derive d;

    return 0;
}
```

监视 1	
名称	值
b	{a=1 }
__vfptr	0x000a9b44 {6-4.exe!void(* Base::`vftable'[3])()}{0>
[0]	0x000a129e {6-4.exe!Base::func1(void)}
[1]	0x000a1113 {6-4.exe!Base::func2(void)}
a	1
d	{b=2 }
Base	{a=1 }
__vfptr	0x000a9b74 {6-4.exe!void(* Derive::`vftable'[4])()}{
[0]	0x000a1230 {6-4.exe!Derive::func1(void)}
[1]	0x000a1113 {6-4.exe!Base::func2(void)}
a	1
b	2

这里Derive中没有看到func3函数，这个vs监视窗口看不到，可以通过内存窗口查看

Base 对象

内存 1					内存 2				
地址: 0x0053FB3C					地址: 0x000A9B44				
0x0053FB3C	44	9b	0a	00	0x000A9B44	9e	12	0a	00
0x0053FB40	01	00	00	00	0x000A9B48	13	11	0a	00
0x0053FB44	cc	cc	cc	cc	0x000A9B4C	00	00	00	00
0x0053FB48	2d	c6	03	b8					
0x0053FB4C	6c	fb	53	00					

Derive对象

内存 1					内存 2				
地址: 0x0053FB28					地址: 0x000A9B74				
0x0053FB28	74	9b	0a	00	0x000A9B74	30	12	0a	00
0x0053FB2C	01	00	00	00	0x000A9B78	13	11	0a	00
0x0053FB30	02	00	00	00	0x000A9B7C	21	12	0a	00
0x0053FB34	cc	cc	cc	cc	0x000A9B80	00	00	00	00
0x0053FB38	cc	cc	cc	cc					

= 1ms

```
1 class Base {
2 public:
3     virtual void func1() { cout << "Base::func1" << endl; }
4     virtual void func2() { cout << "Base::func2" << endl; }
5
6     void func5() { cout << "Base::func5" << endl; }
7 protected:
8     int a = 1;
9 };
10
11 class Derive : public Base
12 {
13 public:
14     // 重写基类的func1
15     virtual void func1() { cout << "Derive::func1" << endl; }
16     virtual void func3() { cout << "Derive::func1" << endl; }
17
18     void func4() { cout << "Derive::func4" << endl; }
19 protected:
20     int b = 2;
21 };
22
23 int main()
24 {
```

```
25     Base b;  
26     Derive d;  
27  
28     return 0;  
29 }
```

```
1  int main()  
2  {  
3      int i = 0;  
4      static int j = 1;  
5      int* p1 = new int;  
6      const char* p2 = "xxxxxxxx";  
7      printf("栈:%p\n", &i);  
8      printf("静态区:%p\n", &j);  
9      printf("堆:%p\n", p1);  
10     printf("常量区:%p\n", p2);  
11  
12     Base b;  
13     Derive d;  
14     Base* p3 = &b;  
15     Derive* p4 = &d;  
16  
17     printf("Person虚表地址:%p\n", *(int*)p3);  
18     printf("Student虚表地址:%p\n", *(int*)p4);  
19     printf("虚函数地址:%p\n", &Base::func1);  
20     printf("普通函数地址:%p\n", &Base::func5);  
21  
22     return 0;  
23 }
```

25 运行结果:

```
26 栈:010FF954  
27 静态区:0071D000  
28 堆:0126D740  
29 常量区:0071ABA4  
30 Person虚表地址:0071AB44  
31 Student虚表地址:0071AB84  
32 虚函数地址:00711488  
33 普通函数地址:007114BF
```