多态复习

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)**主要就是我们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(动态多态)，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”(>^ω^<)喵“，传狗对象过去，就是"汪汪"。

2. 多态的定义及实现

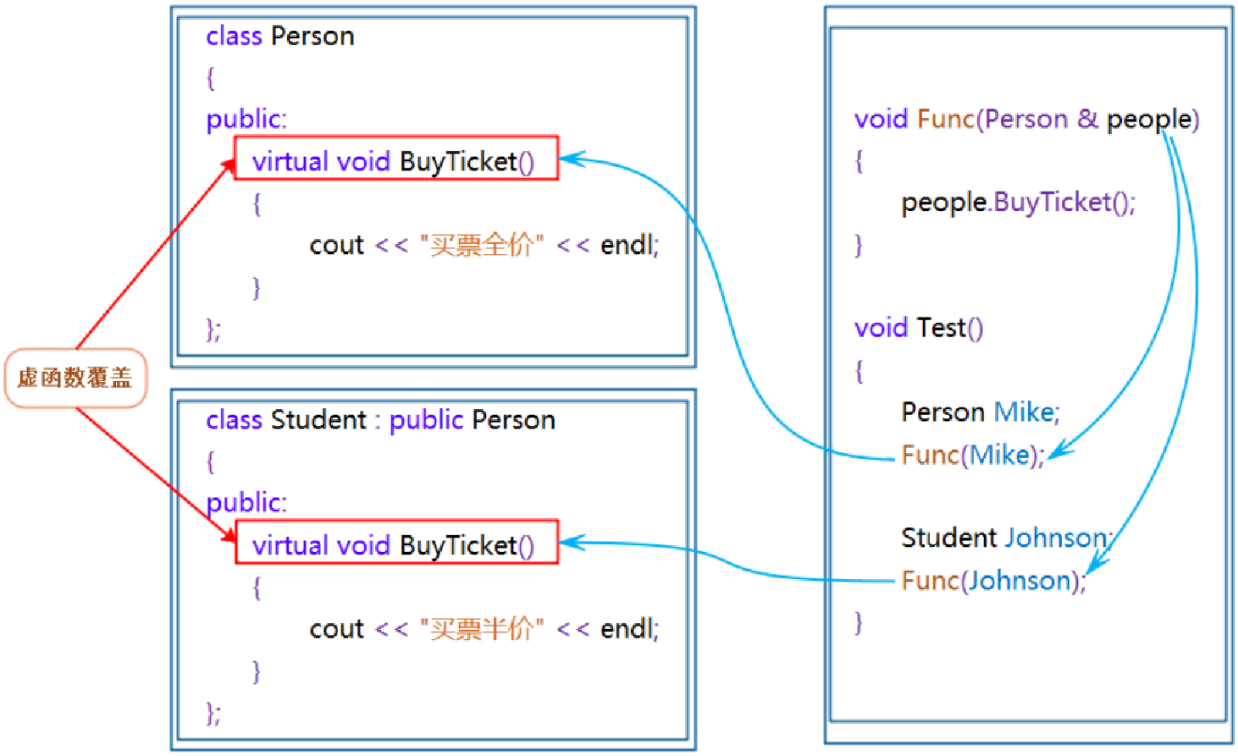
2.1 多态的构成条件

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 实现多态的两个必须条件

- 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，并且完成了虚函数的重写/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的函数，多态的不同形态效果才能达到。



2.1.2 虚函数

类成员函数前加virtual修饰，那么这个成员函数就被成为虚函数。非成员函数前不能加virtual修饰。

代码块

```
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(即派生类和基类虚函数的返回值类型，函数名，参数列表完全相同)，称为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，虽然可以构成重写(因为继承后基类的虚函数被继承下来，在派生类中保持虚函数特性)，但是这种写法很不规范，不建议这样使用。

代码块

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

class Student : public Person
{
public:
    virtual void BuyTicket() { cout << "买票-打折" << endl; }
};

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;
}
```

代码块

```
class Animal
{
public:
    virtual void talk() const {}
};

class Dog : public Animal
{
public:
    virtual void talk() const { std::cout << "汪汪" << std::endl; }
};

class Cat : public Animal
{
public:
```

```
16      virtual void talk() const { std::cout << "(>^ω^<)喵" << std::endl; }
17  };
18
19  void letsHear(const Animal& animal)
20  {
21      animal.talk();
22  }
23
24  int main()
25  {
26      Cat cat;
27      Dog dog;
28      letsHear(cat);
29      letsHear(dog);
30      return 0;
31  }
```

多态的构建需要：

1. 基类函数用 `virtual` 声明，
2. 派生类重写该函数，
3. 通过基类指针或引用调用。

用自己的话来总结一下：实际上就是对一个已经存在继承关系的两个类，让派生类对基类中需要多态语境的函数进行重写，并且在两个已经重写的函数前加上 `virtual` 关键字进行修饰(基类的重写虚函数前必须加上 `virtual`)，至此，多态构建的前两步就已经完成了。

对于多态的调用，即是在一个函数中，让其参数列表带有基类的指针或者引用，在主函数中构建基类或者派生类传进该函数，编译器在运行时，根据传入对象的实际类型（而非指针/引用的静态类型）决定调用哪个类的虚函数，进而实现了——根据传入类型不同而实现不同的功能——多态。

2.1.4 多态的一道选择题

代码块

```
1   class A
2   {
3   public:
4       virtual void func(int val = 1) override { std::cout << "A->" << val <<
    std::endl; }
5
6       virtual void test() { func(); }
7   };
```

```
8
9   class B : public A
10  {
11  public:
12      void func(int val = 0) override { std::cout << "B->" << val <<
    std::endl; }
13  };
14
15  int main(int argc, char* argv[])
16  {
17      B* p = new B;
18      p->test();
19      return 0;
20  }
```

代码块

```
1   // 结果
2   B -> 1
```

多态调用过程：

- `p->test()` 调用继承自 `A` 的 `test()` 方法。
- `test()` 内部调用 `func()`，由于 `func` 是虚函数，且 `p` 实际指向的是 `B` 对象，所以会调用 `B::func(int)`。

默认参数的值：

- 重要规则：虚函数的默认参数值是静态绑定（编译时确定），而函数实现是动态绑定（运行时确定）。
- `test()` 中调用 `func()` 没有显式传参，因此使用默认参数。
- 由于 `test()` 是在 `A` 中定义的，它看到的 `func` 的默认参数是 `A::func` 的默认值 `1`（尽管最终调用的是 `B::func`）。

执行流程：

- `p->test()` → `A::test()` → `func()`（多态调用 `B::func`，但默认参数用 `A::func` 的 `1`）→ 输出 `B->1`。

2.1.5 虚函数重写的一些其他问题

- **协变**

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。协变实际意义不大，只用了解即可。

代码块

```
class A {};
class B : public A {};

class Person {
public:
    virtual A* BuyTicket()
    {
        cout << "买票-全价" << endl;
        return nullptr;
    }
};

class Student : public Person
{
public:
    virtual B* BuyTicket()
    {
        cout << "买票-打折" << endl;
        return nullptr;
    }
};

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

int main()
{
    Person ps;
    Student st;
    Func(&ps);
    Func(&st);

    return 0;
}
```

- **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不符合重写规则，实际上编译器对析构函数的名称做了特殊处理，编译后的析构函数的名称统一处理为 `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。

代码块

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

class B : public A {
public:
    ~B()
    {
        cout << "~B()->delete:" << _p << endl;
        delete _p;
    }
protected:
    int* _p = new int[10];
};

// 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。
int main()
{
    A* p1 = new A;
    A* p2 = new B;
    delete p1;
    delete p2;

    return 0;
}
```

2.1.6 override 和 final关键字

C++对虚函数的重写要求比较严格，但是在某些情况下由于疏忽，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，反而进行Debug调试时会得不偿失，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`修饰。

代码块

```
// 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
class Car
{
public:
    virtual void Dirve()
    {
    }
};

class Benz :public Car
{
public:
    // 此处不加override时，不会飘红报错
    virtual void Drive() override { cout << "Benz-舒适" << endl; }
};

int main()
{
    return 0;
}
```

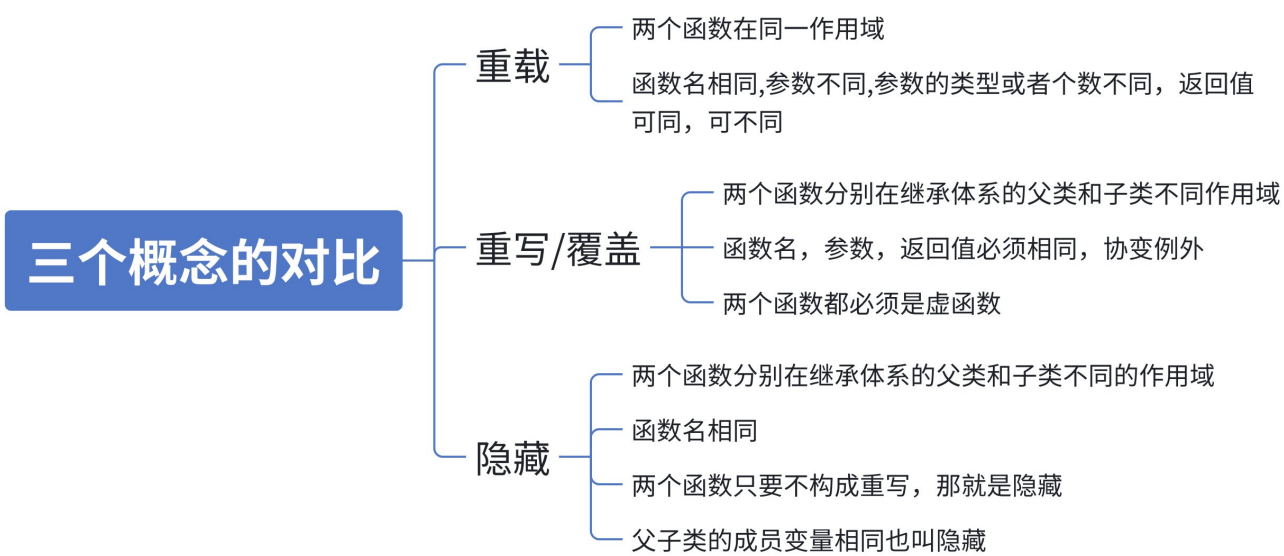
代码块

```
// error C3248: “Car::Drive”: 声明为“final”的函数无法被“Benz::Drive”重写
class Car
{
public:
    virtual void Drive() final {}
};

class Benz :public Car
{
public:
    virtual void Drive() { cout << "Benz-舒适" << endl; }
};
```

```
13
14  int main()
15  {
16      return 0;
17  }
```

2.1.7 重载/重写/隐藏的对比



3. 纯虚函数和抽象类

在虚函数的后面加上 =0，则这个函数为纯虚函数，纯虚函数不需要定义实现(实现没有意义，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。

代码块

```
1  class Car
2  {
3  public:
4      virtual void Drive() = 0;
5  };
6
7  class Benz :public Car
8  {
9  public:
```

```cpp
    virtual void Drive()
    {
        cout << "Benz-舒适" << endl;
    }
};

class BMW :public Car
{
public:
    virtual void Drive()
    {
        cout << "BMW-操控" << endl;
    }
};

int main()
{
    // 编译报错：error C2259: “Car”: 无法实例化抽象类
    Car car;
    Car* pBenz = new Benz;
    pBenz->Drive();
    Car* pBMW = new BMW;
    pBMW->Drive();
    return 0;
}
```

4. 多态的原理

4.1 虚函数表指针

代码块

```cpp
class Base
{
public:
    virtual void Func1()
    {
        cout << "Func1()" << endl;
    }

protected:
    int _b = 1;
    char _ch = 'x';
```

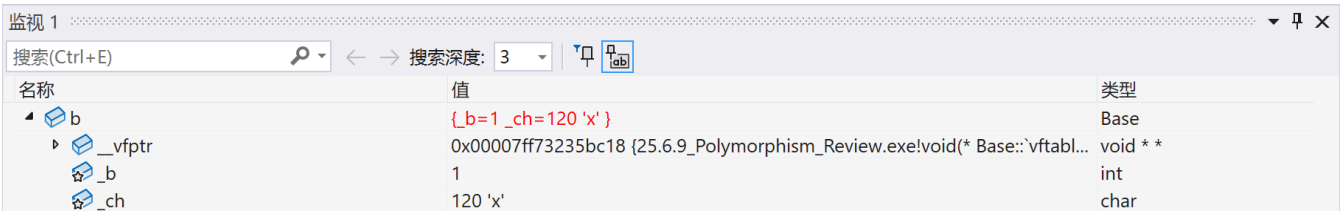
```
};

int main()
{
    Base b;
    cout << sizeof(b) << endl;

    return 0;
}
```

上述代码在VS2022 x64环境下的结果是16，如果注释掉`Base`类中的`Func1`虚函数，则为8。

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。



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，

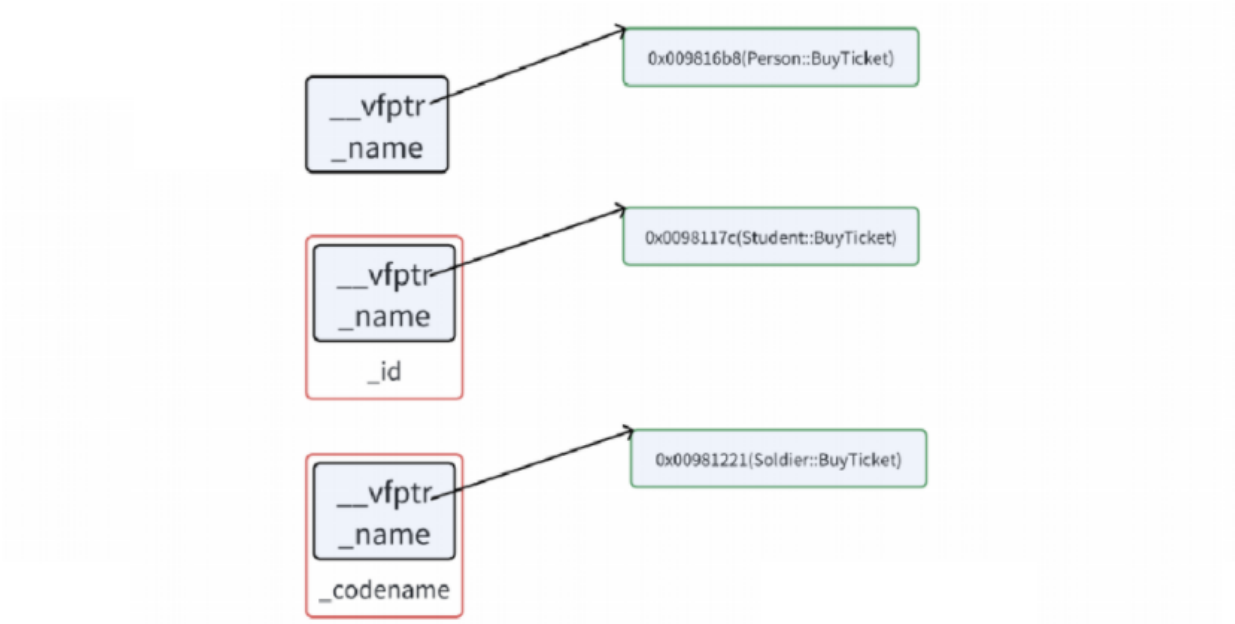
ptr指向Student对象调用Student::BuyTicket的呢？通过下图我们可以看到，满足多态条件后，底层

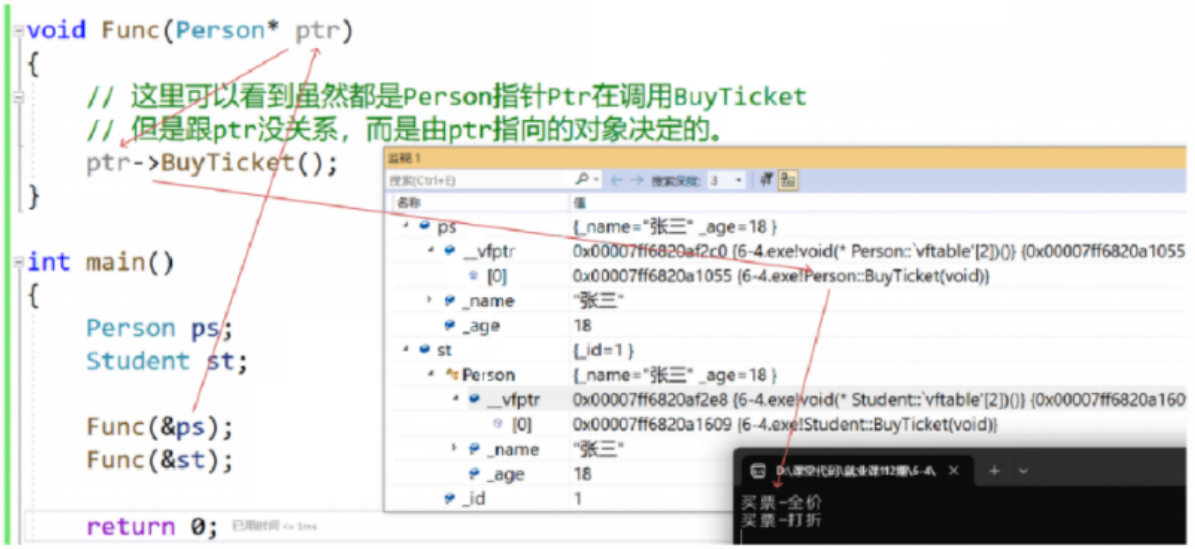
不再是编译时通过调用对象确定函数的地址，而是运行时到指向的对象的虚表中确定对应的虚函数的

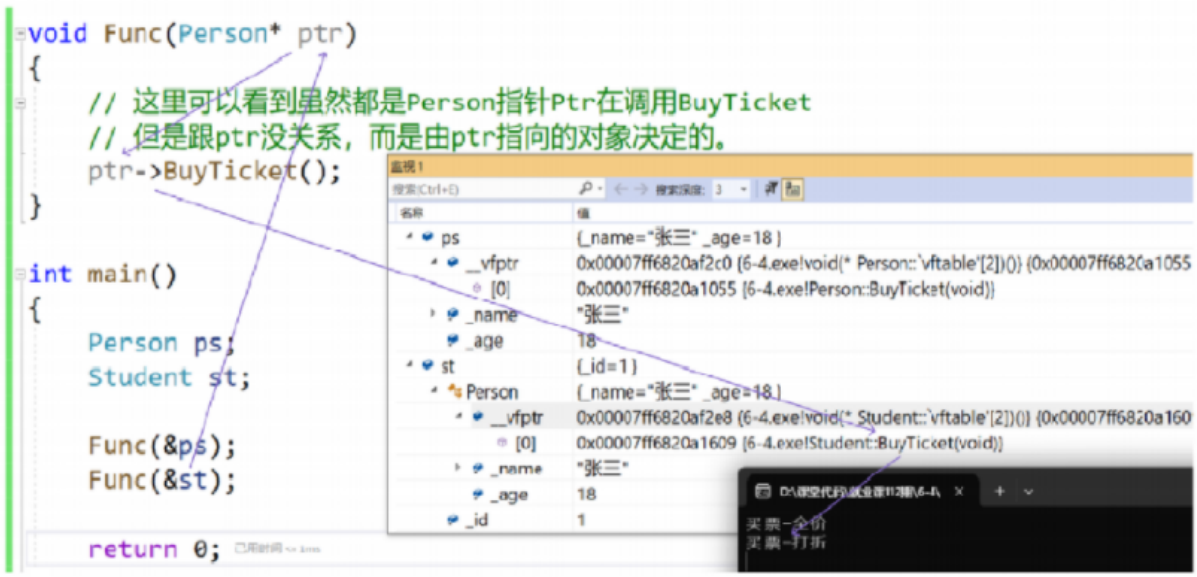
地址，这样就实现了指针或引用指向基类就调用基类的虚函数，指向派生类就调用派生类对应的虚函

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。







代码块

```
1   class Person
2   {
```

```cpp
public:
    virtual void BuyTicket() { cout << "买票-全价" << endl; }

private:
    string _name;
};

class Student : public Person
{
public:
    virtual void BuyTicket() { cout << "买票-打折" << endl; }

private:
    string _id;
};

class Soldier : public Person
{
public:
    virtual void BuyTicket() { cout << "买票-优先" << endl; }

private:
    string _codename;
};

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()
{
    // 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
    // 多态也会发生在多个派生类之间。
    Person ps;
    Student st;
    Soldier sr;
    Func(&ps);
    Func(&st);
    Func(&sr);
    return 0;
}

```

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。

代码块

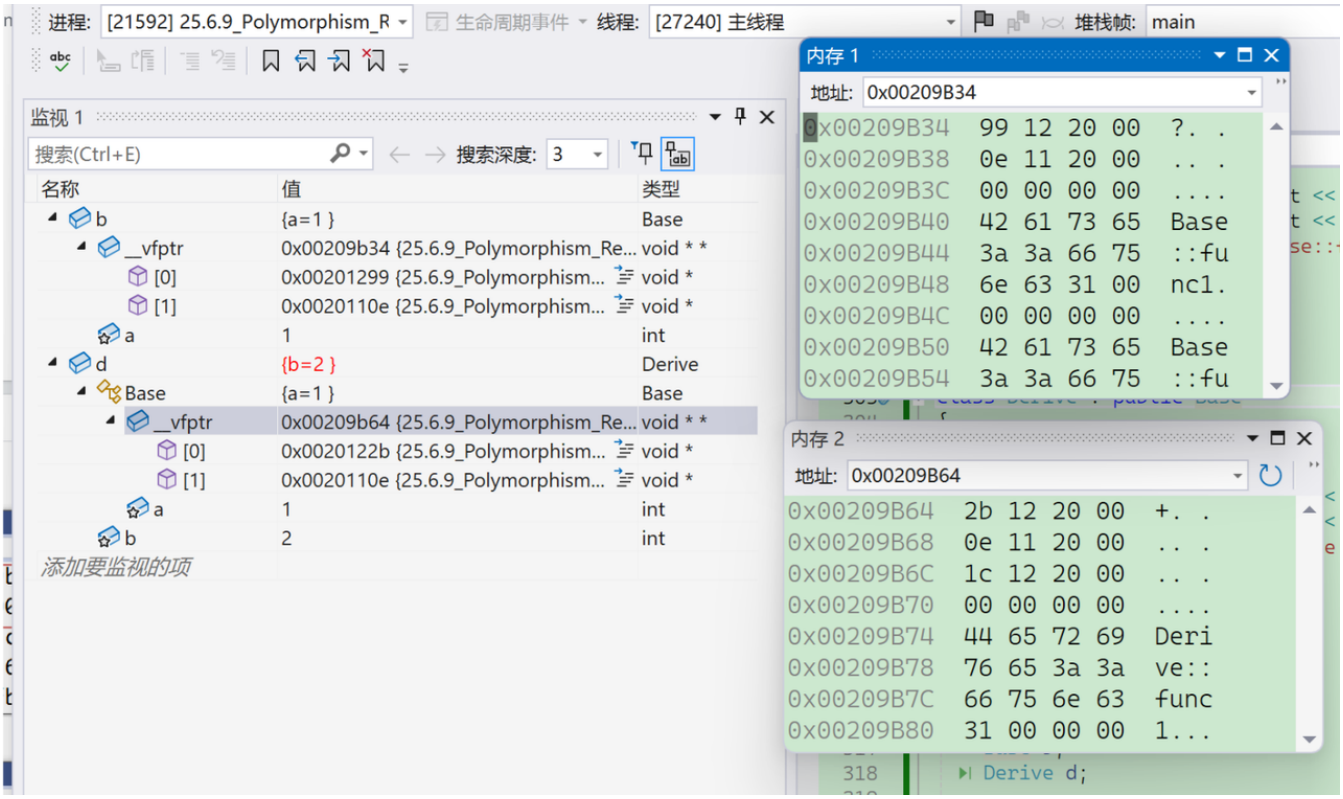
```
// ptr是指针+BuyTicket是虚函数满足多态条件。
// 这里就是动态绑定，编译在运行时到ptr指向对象的虚函数表中确定调用函数地址
    ptr->BuyTicket();
00EF2001 mov         eax,dword ptr [ptr]
00EF2004 mov         edx,dword ptr [eax]
00EF2006 mov         esi,esp
00EF2008 mov         ecx,dword ptr [ptr]
00EF200B mov         eax,dword ptr [edx]
00EF200D call        eax

    // BuyTicket不是虚函数，不满足多态条件。
    // 这里就是静态绑定，编译器直接确定调用函数地址
    ptr->BuyTicket();
00EA2C91 mov ecx,dword ptr [ptr]
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。
- 派生类有两部分构成，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。但需要注意的是，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也都是独立的。
- 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。

- 虚函数表本质就是一个存虚函数指针的指针数组，一般情况这个数组最后放了一个0x00000000标记。(这个C++并没有进行规定，各个编译器自行定义的，VS系列编译器会在最后放个0x00000000标记，g++系列编译不会放)
- 虚函数存在哪？虚函数和普通函数一样，编译后是一段指令，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？这个问题严格来说并没有标准答案。C++标准并没有规定。



代码块

```
class Base
{
public:
    virtual void func1() { cout << "Base::func1" << endl; }
    virtual void func2() { cout << "Base::func2" << endl; }
    void func5() { cout << "Base::func5" << endl; }

protected:
    int a = 1;
};

class Derive : public Base
{
public:
    // 重写基类的func1
    virtual void func1() { cout << "Derive::func1" << endl; }
    virtual void func3() { cout << "Derive::func1" << endl; }
    void func4() { cout << "Derive::func4" << endl; }
```

```
19
20  protected:
21      int b = 2;
22  };
23
24  int main()
25  {
26      Base b;
27      Derive d;
28
29      return 0;
30  }
31
32  int main()
33  {
34      int i = 0;
35      static int j = 1;
36      int* p1 = new int;
37      const char* p2 = "xxxxxxxx";
38      printf("栈:%p\n", &i);
39      printf("静态区:%p\n", &j);
40      printf("堆:%p\n", p1);
41      printf("常量区:%p\n", p2);
42
43      Base b;
44      Derive d;
45      Base* p3 = &b;
46      Derive* p4 = &d;
47
48      printf("Person虚表地址:%p\n", *(int*)p3);
49      printf("Student虚表地址:%p\n", *(int*)p4);
50      printf("虚函数地址:%p\n", &Base::func1);
51      printf("普通函数地址:%p\n", &Base::func5);
52
53      return 0;
54  }
55
56  //运行结果:
57  //栈 : 010FF954
58  //静态区 : 0071D000
59  //堆 : 0126D740
60  //常量区 : 0071ABA4
61  //Person虚表地址 : 0071AB44
62  //Student虚表地址 : 0071AB84
63  //虚函数地址 : 00711488
64  //普通函数地址 : 007114BF
```

5. 多态详解

5.1 多态的基本条件

要实现运行时多态，必须满足：

i. 继承关系：派生类继承自基类。

ii. 虚函数：基类中至少有一个 `virtual` 修饰的成员函数，派生类可以重写（override）它。

iii. 基类指针/引用调用：必须通过 基类指针或引用 调用虚函数，才能触发动态绑定。

5.2 虚函数表（vtable）机制

C++ 编译器通过 虚函数表（vtable） 实现多态，具体流程如下：

(1) 虚函数表的创建

- 每个包含虚函数的类 都有一个 虚函数表（vtable），存储该类所有虚函数的地址。
- 每个对象 内部有一个隐藏的指针（`vptr`），指向该类的 `vtable`。

(2) 继承时的虚函数表

- 派生类继承基类时，会先复制基类的 `vtable`。
- 如果派生类重写了虚函数，则 `vtable` 中对应的函数地址会被替换为派生类的版本。
- 如果派生类新增了虚函数，则这些函数会被追加到 `vtable` 的末尾。

(3) 调用过程

当通过 基类指针/引用 调用虚函数时：

i. 程序先找到对象的 `vptr`（虚表指针）。

ii. 通过 `vptr` 找到 `vtable`。

iii. 从 `vtable` 中找到正确的函数地址并调用。

5.3 示例分析

代码块

```
1   #include <iostream>
```

```cpp

class Base
{
public:
    virtual void func1() { std::cout << "Base::func1" << std::endl; }
    virtual void func2() { std::cout << "Base::func2" << std::endl; }
};

class Derived : public Base
{
public:
    void func1() override { std::cout << "Derived::func1" << std::endl; } // 重写 func1
    virtual void func3() { std::cout << "Derived::func3" << std::endl; } // 新增虚函数
};

int main()
{
    Base* b = new Derived();
    b->func1();  // 输出 Derived::func1（多态调用）
    b->func2();  // 输出 Base::func2（未重写，调用基类版本）
    // b->func3();  // 错误！基类指针无法访问派生类新增的虚函数
    delete b;
    return 0;
}
```

对应的 `vtable` 结构：

类	`vtable` 内容
`Base`	`&Base::func1`, `&Base::func2`
`Derived`	`&Derived::func1`, `&Base::func2`, `&Derived::func3`

5.4 关键点总结

i. `virtual` 关键字：

- 只有用 `virtual` 修饰的函数才能被动态绑定。
- 派生类重写时可以用 `override`（C++11）显式标记。

ii. 动态绑定（运行时多态）：

- 通过 基类指针/引用 调用虚函数时，实际调用哪个版本由 对象的真实类型 决定。

iii. `vptr` 和 `vtable`：

- 每个对象存储一个 `vptr`，指向所属类的 `vtable`。
- `vtable` 在编译时生成，存储虚函数的地址。

iv. 默认参数是静态绑定的：

- 虚函数的默认参数在 编译时 确定，不会参与多态（见之前的例子）。

v. 虚析构函数：

- 如果基类可能被继承，并且可能通过基类指针删除派生类对象，则 基类析构函数必须是虚函数，否则会导致派生类析构不被调用，内存泄漏。

5.5 多态 VS 非多态

场景	是否多态	调用方式
`Base obj; obj.func();`	❌ 否	直接调用 `Base::func`
`Derived obj; obj.func();`	❌ 否	直接调用 `Derived::func`
`Base* p = new Derived(); p->func();`	✅ 是	通过 `vtable` 动态调用

5.6 总结

- 多态的核心：`virtual` + 继承 + 基类指针/引用调用。
- 底层实现：`vptr` + `vtable` 动态查找函数地址。
- 默认参数 是静态绑定的，不影响多态行为。
- 虚析构函数 必须用于基类，避免内存泄漏。