C++ 多态的实现原理分析

一、什么是多态

在面向对象开发中，多态是一个很重要的特性。

什么是多态呢？就是程序运行时，父类指针可以根据具体指向的子类对象，来执行不同的函数，表现为多态。

二、C++ 多态的实现原理

1. 实现原理

当类中存在虚函数时，编译器会在类中自动生成一个虚函数表

虚函数表是一个存储类成员函数指针的数据结构

虚函数表由编译器自动生成和维护

virtual 修饰的成员函数会被编译器放入虚函数表中

存在虚函数时，编译器会为对象自动生成一个指向虚函数表的指针（通常称之为 vptr 指针）

2. 举个例子

看完上面的实现原理，你可能会觉得有点懵，接下来我们就一点点分析和验证上面的结论。

#include <iostream>

using namespace std;

class Parent

{

public:

// 父类虚函数必须要有 virtual 关键字

virtual void fun()

{

cout << "父类" << endl;

}

};

class Child : public Parent

{

public:

// 子类有没有 virtual 关键字都可以

void fun()

{

cout << "子类" << endl;

}

};

int main()

{

Parent \*p = NULL; // 创建一个父类的指针

Parent parent;

Child child;

p = &parent; // 指向父类的对象

p->fun(); // 执行的是父类的 fun() 函数

p = &child; // 指向子类的对象

p->fun(); // 执行的是子类的 fun() 函数

return 0;

}

如上例代码所示，当我们传入父类对象时，将调用和执行父类的函数，当我们传入子类对象时，将调用和执行子类的函数。而 C++ 编译器的执行过程其实是这样的：

父类的 fun() 是个虚函数，所以编译器给父类对象自动添加了一个 vptr 指针，指向父类的虚函数表，这个虚函数表里存放了父类的 fun() 函数的函数指针

子类的 fun() 函数是重写了父类的，即写不写 virtual 编译器都会为其自动添加一个 virtual，然后编译器给子类对象自动添加了一个 vptr 指针，指向子类的虚函数表，这个虚函数表里存放了子类的

fun() 函数的函数指针

执行 p->fun() 时，编译器检测到 fun() 是一个虚函数，所以不会静态的将 Parent 类的 fun() 方法直接编译过来，而是是运行的时候，动态的根据 base 指向的对象，找到这个对象的 vptr 指针，

然后找到这个对象的虚函数表，最后调用虚函数表里对应的函数，实现多态

3. 证明 vptr 的存在

上面说了这么多，那么怎么证明说的都是对的呢？vptr 指针真的存在么？

其实要证明 vptr 的存在很简单，我们只需要创建两个相同的类，一个类有虚函数，一个类没有虚函数，然后通过 sizeof() 方法打印出类对象的大小就行。如下：

#include <iostream>

using namespace std;

class Parent1

{

public:

int a;

void fun() {} // 非虚函数

};

class Parent2

{

public:

int a;

virtual void fun() {} // 虚函数

};

int main()

{

Parent1 p1;

Parent2 p2;

cout << sizeof(p1) << endl;

cout << sizeof(p2) << endl;

return 0;

}

运行结果：

4

8

可以看到，存在虚函数的类的对象，大小大了4个字节，这正好是一个指针对象的大小（指针对象的大小可能会根据运行环境而改变，32位的系统指针的大小是4个字节），这说明编译器确实给我们

添加了这么一个指针对象 vptr。

4. 父类的构造方法中调用虚函数，会发生多态吗

看下面这个例子：

#include <iostream>

using namespace std;

class Parent

{

public:

Parent()

{

// 父类的构造方法中执行虚函数，会发生多态吗？

fun();

}

virtual void fun()

{

cout << "父类" << endl;

}

};

class Child : public Parent

{

public:

Child()

{

fun();

}

void fun()

{

cout << "子类" << endl;

}

};

void main()

{

Child c;

return 0;

}

执行程序会发现，创建子类对象时，会先创建父类对象，而父类的构造方法中调用虚函数，执行的并不是子类的 fun() 函数，而是父类自己的 fun() 函数，并没有发生多态。

即答案是：父类的构造方法中调用虚函数，不会发生多态。这个和 vptr 的分步初始化有关。

5. vptr 的分步初始化

从上例中我们看到，在父类中调用虚函数时，执行的还是父类的函数，没有发生多态。这是因为当创建子类对象时，编译器的执行顺序其实是这样的：

对象在创建时，由编译器对 vptr 进行初始化

子类的构造会先调用父类的构造函数，这个时候 vptr 会先指向父类的虚函数表

子类构造的时候，vptr 会再指向子类的虚函数表

对象的创建完成后，vptr 最终的指向才确定