C++的多态性用一句话概括就是：在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数

  1：用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。

  2：存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。

  3：多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。

  4：多态用虚函数来实现，结合动态绑定.

  5:纯虚函数是虚函数再加上 = 0；

  6：抽象类是指包括至少一个纯虚函数的类。

纯虚函数:virtual void fun()=0;即抽象类！必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

我们先看个例子

[复制代码](javascript:void(0);)

#include "stdafx.h"

#include <iostream>

#include <stdlib.h>

using namespace std;

class Father

{

public:

void Face()

{

cout << "Father's face" << endl;

}

void Say()

{

cout << "Father say hello" << endl;

}

};

class Son:public Father

{

public:

void Say()

{

cout << "Son say hello" << endl;

}

};

void main()

{

Son son;

Father \*pFather=&son; // 隐式类型转换

pFather->Say();

}

[复制代码](javascript:void(0);)

输出的结果为:  
https://images2015.cnblogs.com/blog/1019006/201611/1019006-20161117135152717-1779221032.png

我们在main()函数中首先定义了一个Son类的对象son，接着定义了一个指向Father类的指针变量pFather，然后利用该变量调用pFather->Say().估计很多人往往将这种情况和c++的多态性搞混淆，认为son实际上是Son类的对象

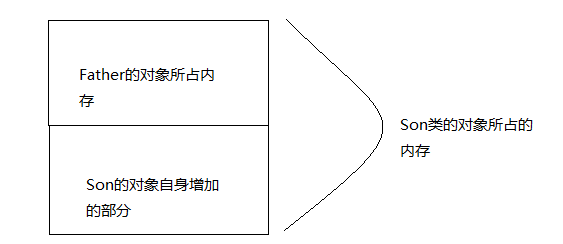
，应该是调用Son类的Say，输出"Son say hello",然而结果却不是.

　　从编译的角度来看:

　　　　c++编译器在编译的时候，要确定每个对象调用的函数（非虚函数）的地址，这称为早期绑定，当我们将Son类的对象son的地址赋给pFather时，c++编译器进行了类型转换，此时c++编译器认为变量pFather保存的就是Father对象的地址，当在main函数中执行pFather->Say(),

调用的当然就是Father对象的Say函数

　从内存角度看



Son类对象的内存模型如上图

我们构造Son类的对象时，首先要调用Father类的构造函数去构造Father类的对象，然后才调用Son类的构造函数完成自身部分的构造，从而拼接出一个完整的Son类对象。当我们将Son类对象转换为Father类型时，该对象就被认为是原对象整个内存模型的上半部分，

也就是上图中“Father的对象所占内存”，那么当我们利用类型转换后的对象指针去调用它的方法时，当然也就是调用它所在的内存中的方法，因此，输出“Father Say hello”，也就顺理成章了。

　　正如很多人那么认为，在上面的代码中，我们知道pFather实际上指向的是Son类的对象，我们希望输出的结果是son类的Say方法，那么想到达到这种结果，就要用到虚函数了。

前面输出的结果是因为编译器在编译的时候，就已经确定了对象调用的函数的地址，要解决这个问题就要使用晚绑定，当编译器使用晚绑定时候，就会在运行时再去确定对象的类型以及正确的调用函数，而要让编译器采用晚绑定，就要在基类中声明函数时使用virtual关键字，

这样的函数我们就称之为虚函数，一旦某个函数在基类中声明为virtual，那么在所有的派生类中该函数都是virtual，而不需要再显式地声明为virtual。

　　代码稍微改动一下，看一下运行结果

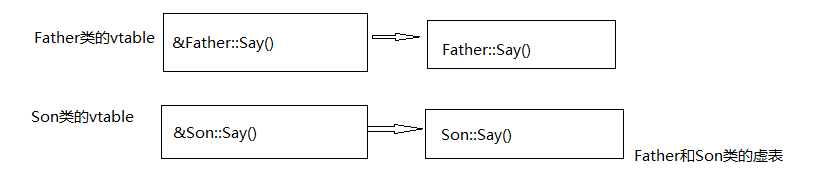
|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include <stdlib.h>  using namespace std;    class Father  {  public:  void Face()  {  cout << "Father's face" << endl;  }    virtual void Say()  {  cout << "Father say hello" << endl;  }  };  class Son:public Father  {  public:  void Say()  {  cout << "Son say hello" << endl;  }  };  void main()  {  Son son;  Father \*pFather=&son; // 隐式类型转换  pFather->Say();  } |

[复制代码](javascript:void(0);)

https://images2015.cnblogs.com/blog/1019006/201611/1019006-20161117142802076-1508128100.png

我们发现结果是"Son say hello"也就是根据对象的类型调用了正确的函数，那么当我们将Say()声明为virtual时，背后发生了什么。

　　编译器在编译的时候，发现Father类中有虚函数，此时编译器会为每个包含虚函数的类创建一个虚表(即 vtable)，该表是一个一维数组，在这个数组中存放每个虚函数的地址，



那么如何定位虚表呢？编译器另外还为每个对象提供了一个虚表指针(即vptr)，这个指针指向了对象所属类的虚表，在程序运行时，根据对象的类型去初始化vptr，从而让vptr正确的指向了所属类的虚表，从而在调用虚函数的时候，能够找到正确的函数，对于第二段代码程序，由于

pFather实际指向的对象类型是Son，因此vptr指向的Son类的vtable，当调用pFather->Son()时，根据虚表中的函数地址找到的就是Son类的Say()函数.

　　正是由于每个对象调用的虚函数都是通过虚表指针来索引的，也就决定了虚表指针的正确初始化是非常重要的，换句话说，在虚表指针没有正确初始化之前，我们不能够去调用虚函数，那么虚表指针是在什么时候，或者什么地方初始化呢？

答案是在构造函数中进行虚表的创建和虚表指针的初始化，在构造子类对象时，要先调用父类的构造函数，此时编译器只“看到了”父类，并不知道后面是否还有继承者，它初始化父类对象的虚表指针，该虚表指针指向父类的虚表，当执行子类的构造函数时，子类对象的虚表指针被初始化

，指向自身的虚表。

　　总结(基类有虚函数的):

　　1:每一个类都有虚表

2:虚表可以继承，如果子类没有重写虚函数，那么子类虚表中仍然会有该函数的地址，只不过这个地址指向的是基类的虚函数实现，如果基类有3个虚函数，那么基类的虚表中就有三项(虚函数地址)，派生类也会虚表，至少有三项，如果重写了相应的虚函数，那么虚表中的地址就会改变，

指向自身的虚函数实现，如果派生类有自己的虚函数，那么虚表中就会添加该项。

　　3：派生类的虚表中虚地址的排列顺序和基类的虚表中虚函数地址排列顺序相同。

这就是c++中的多态性，当c++编译器在编译的时候，发现Father类的Say()函数是虚函数，这个时候c++就会采用晚绑定技术，也就是编译时并不确定具体调用的函数，而是在运行时，依据对象的类型来确认调用的是哪一个函数，这种能力就叫做c++的多态性，我们没有在Say()函数

前加virtual关键字时，c++编译器就确定了哪个函数被调用，这叫做早期绑定。

　　c++的多态性就是通过晚绑定技术来实现的。

　　c++的多态性用一句话概括就是:在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

　　虚函数是在基类中定义的，目的是不确定它的派生类的具体行为，例如:

　　定义一个基类:class Animal //动物，它的函数为breathe()

　　再定义一个类class Fish //鱼。它的函数也为breathe()

　　再定义一个类class Sheep //羊，它的函数也为breathe()

将Fish，Sheep定义成Animal的派生类，然而Fish与Sheep的breathe不一样，一个是在水中通过水来呼吸，一个是直接呼吸，所以基类不能确定该如何定义breathe，所以在基类中只定义了一个virtual breathe，它是一个空的虚函数，具体的函数在子类中分别定义，程序一般运行时

，找到类，如果它有基类，再找到它的基类，最后运行的是基类中的函数，这时，它在基类中找到的是virtual标识的函数，它就会再回到子类中找同名函数，派生类也叫子类，基类也叫父类，这就是虚函数的产生，和类的多态性的体现。

　　这里的多态性是指类的多态性。

　　函数的多态性是指一个函数被定义成多个不同参数的函数。当你调用这个函数时，就会调用不同的同名函数。

一般情况下(不涉及虚函数)，当我们用一个指针/引用调用一个函数的时候，被调用的函数是取决于这个指针/引用的类型。

当设计到多态性的时候，采用了虚函数和动态绑定，此时的调用就不会在编译时候确定而是在运行时确定。不在单独考虑指针/引用的类型而是看指针/引用的对象的类型来判断函数的调用，根据对象中虚指针指向的虚表中的函数的地址来确定调用哪个函数

现在我们看一个体现c++多态性的例子，看看输出结果:

#include "stdafx.h"

#include <iostream>

#include <stdlib.h>

using namespace std;

class CA

{

public:

void f()

{

cout << "CA f()" << endl;

}

virtual void ff()

{

cout << "CA ff()" << endl;

f();

}

};

class CB : public CA

{

public :

virtual void f()

{

cout << "CB f()" << endl;

}

void ff()

{

cout << "CB ff()" << endl;

f();

CA::ff();

}

};

class CC : public CB

{

public:

virtual void f()

{

cout << "C f()" << endl;

}

};

int main()

{

CB b;

CA \*ap = &b;

CC c;

CB &br = c;

CB \*bp = &c;

ap->f();

cout << endl;

b.f();

cout << endl;

br.f();

cout << endl;

bp->f();

cout << endl;

ap->ff();

cout << endl;

bp->ff();

cout << endl;

return 0;

}

输出结果:  
