多态的实现

什么是多态?

1. 相同对象收到不同消息或不同对象收到相同消息时产生的不同的动作。
2. 一个接口，多种方法

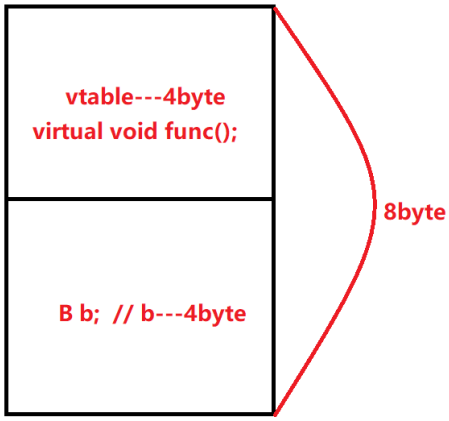
虚成员函数表（vtable）

这一节将介绍多态是如何实现的，关于如何实现多态，对于程序设计人员来说即使不知道也是完全没有关系的，但是对于加深对多态的理解具有重要意义，故而在此节中稍微阐述一下多态的实现机制。

在C++中通过虚成员函数表vtable实现多态，虚函数表中存储的是类中虚函数的入口地址。在普通的类中是没有虚函数表的，只有在具有虚函数的类中（无论是自身添加的虚函数还是继承过来的虚函数表）才会具有虚函数表，通常虚成员函数表的首地址将会被存入对象的最前面（在32位的操作系统中，存储地址是用4个字节，因此这个首地址就会占用对象的前四个字节的空间）。

```
// 在这个类中完全没有虚函数表的概念
class A{
public:
    void func(){}
private:
    int x;
};

// 这个类中有虚函数了，所以这个类中已经包含了虚函数表
class B{
public:
    virtual void func(){}        // 这个函数的入口地址就存在虚函数表中
private:
    int x;
};

// 这个类继承自B类，因为B类中含有虚函数表，所以该表也被继承到C类中，也有虚函数表
class C : public B{
public:
};

int main(){
    A a;    // a的大小为4byte
    B b;    // b大小为8byte，因为b中含有虚函数表(vtable),占用4个byte
    return 0;
}
```

```
// 这个类中有虚函数了，所以这个类中已经包含了虚函数表
class B{
public:
    virtual void func(){}       // 这个函数的入口地址就存在虚函数表中
private:
    int x;
};
```

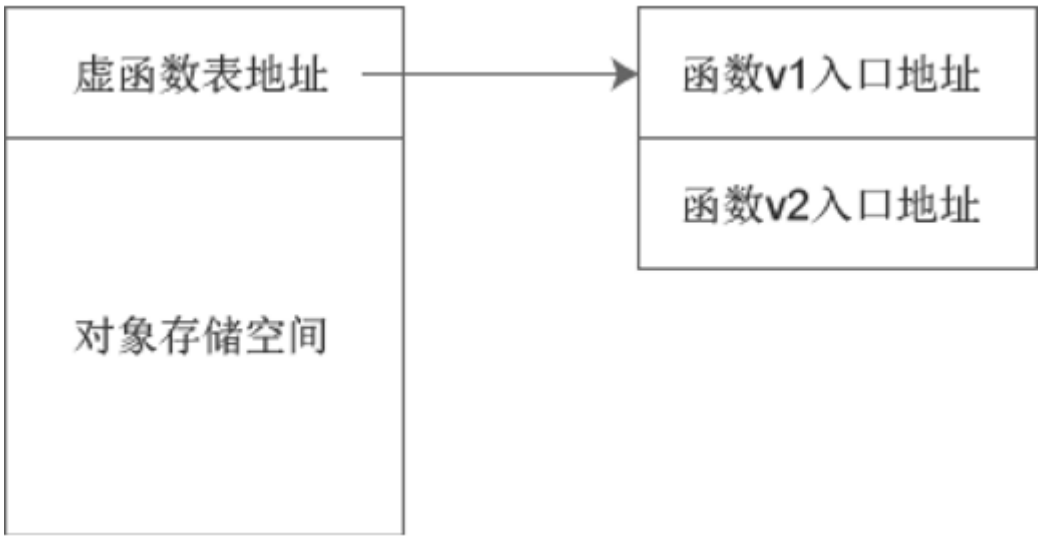



```
#include <iostream>

using namespace std;

class base{
public:
    virtual void v1(){ }
    virtual void v2(){ }
};

class derived : public base{
public:
    virtual void v1() override{ }
    virtual void v2() override{ }
};

int main(){
    base b;
    derived d;
    base *p;
    p = &b;
    p->v1();
    p->v2();
    p = &d;  // 基类指针指向子类对象
    p->v1();
    p->v2();

    return 0;
}
```

我们将两个类定义成例1所示形式，两个类中各有两个虚函数v1和v2，我们将其函数入口地址找到列于下表中：

虚成员函数	函数入口地址	虚成员函数	函数入口地址
base::v1	00D15834	derived::v1	00D15844
base::v2	00D15838	derived::v2	00D15848

虚函数表里面存储的就是虚函数的入口地址。我们再来看主函数，在主函数中先定义了base类对象b，因为b类中有虚函数，因此存在虚函数表，而虚函数表的首地址就存储在对象所在存储空间的最前。当然声明derived对象d之后，情况也跟下图中一样，同样在对象存储空间中包含虚成员函数表地址。



之后定义了一个基类类型的指针p，当我们通过基类指针p调用虚函数v1或v2时，系统会先去p所指向的对象的前四个字节中寻找到虚函数表地址，之后在内存中找到该虚函数表，然后在表中找到对应函数的入口地址，之后直接访问这个函数了。**当p指针指向的是基类对象时，基类的虚函数表将会被访问，基类中虚函数将会被调用。当p指针指向的是派生类对象，则访问的是派生类的虚函数表，派生类的虚函数表中存的是派生类中的虚函数入口地址，因此调用的是派生类中的虚函数。**

使用多态会降低程序运行效率，使用多态的程序会使用更多的存储空间，存储虚函数表等内容，而且在调用函数时需要去虚函数表中查询函数入口地址，这会增加程序运行时间。在设计程序时，程序设计人员可以选择性的使用多态，对于有需要的函数使用多态，对于其它的函数则不要采用多态。**通常情况下，如果一个类需要作为基类，并且期望在派生类中修改某成员函数的功能，并且在使用类对象的时候会采用指针或引用的形式访问该函数，则将该函数声明为虚函数。**

普通继承

```
class A{
};
// 用于取得基类中的成员方法和属性的过程，会产生拷贝，代码重用的过程
class B :  public A{
};
```

虚继承(一般用于解决菱形继承产生的相关问题:1. 多重拷贝问题 2. 二义性问题)

```
class A{
public:
    void setValue();
};
// 用于取得基类中的成员方法和属性的过程，会产生拷贝，代码重用的过程
class B : virtual public A{
};
class C : virtual public A{
};
class D : public B, public C{
};

int main(){
```

```
    D d;
    d.setValue();   // 虚继承才能不报错
    return 0;
}
```