多态

多态性（polymorphism）是面向对象设计语言的基本特征之一。仅仅是将数据和函数捆绑在一起，进行类的封装，使用一些简单的继承，还不能算是真正应用了面向对象的设计思想。多态性是面向对象的精髓。多态性可以简单地概括为“一个接口，多种方法”。

多态按字面意思就是多种形态。比如说：警车鸣笛，普通人反应一般，但逃犯听见会大惊失色，拔腿就跑。又比如说，...

通常是指对于同一个消息、同一种调用，在不同的场合，针对不同的对象下，执行不同的行为 。

1、为什么要用多态?

我们知道，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）。它们的目的都是为了代码重用。而多态除了代码的复用性外，还可以解决项目中紧偶合的问题，提高程序的可扩展性。

如果项目耦合度很高的情况下，维护代码时修改一个地方会牵连到很多地方，会无休止的增加开发成本。而降低耦合度，可以保证程序的扩展性。而多态对代码具有很好的可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

C++支持两种多态性：编译时多态和运行时多态。

编译时多态：也称为静态多态，我们之前学习过的**函数重载**、**运算符重载**就是采用的静态多态，C++编译器根据传递给函数的参数和函数名决定具体要使用哪一个函数，又称为先期联编（early binding）。

运行时多态：在一些场合下，编译器无法在编译过程中完成联编，必须在程序运行时完成选择，因此编译器必须提供这么一套称为“动态联编”（dynamic binding）的机制，也叫晚期联编（late binding）。C++通过虚函数来实现动态联编。

接下来，我们提到的多态，不做特殊说明，指的就是动态多态。

2、虚函数的定义

什么是虚函数呢？虚函数就是在基类中被声明为virtual，并在一个或多个派生类中被重新定义的成员函数。其形式如下:

```
// 类内部
class 类名
{
    virtual 返回类型 函数名(参数表)
    {
        //...
    }
};

//类之外
virtual 返回类型 类名::函数名(参数表)
{
    //...
}
```

如果一个基类的成员函数定义为虚函数，那么它在所有派生类中也保持为虚函数，即使在派生类中省略了virtual关键字，也仍然是虚函数。派生类要对虚函数进行中可根据需重定义，重定义的格式有一定的要求：

- 与基类的虚函数有相同的参数个数；
- 与基类的虚函数有相同的参数类型；
- 与基类的虚函数有相同的返回类型。

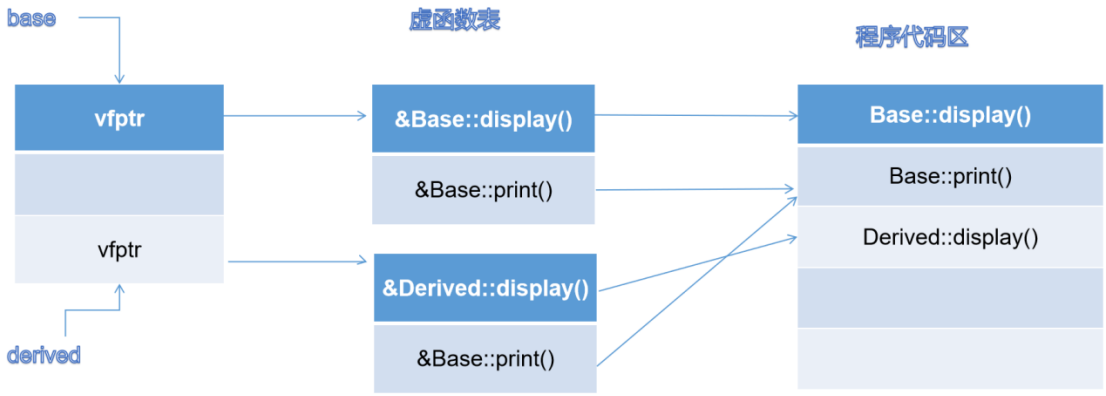
```
1  class Base
2  {
3  public:
4      virtual void display()
5      {
6          cout << "Base::display()" << endl;
7      }
8
9      virtual void print()
10     {
11         cout << "Base::print()" << endl;
12     }
13 };
14
15 class Derived
16 : public Base
17 {
18 public:
19     virtual void display()
20     {
21         cout << "Derived::display()" << endl;
22     }
23 };
24
25 void test(Base *pbase)
26 {
27     pbase->display();
28 }
29
30 int main()
31 {
32     Base base;
33     Derived derived;
34     test(&base);
35     test(&derived);
36     return 0;
37 }
```

上面的例子中，对于test()函数，如果不管测试的结果，从其实现来看，通过类Base的指针pbase只能调用到Base类型的display函数；但最终的结果是25行的test调用，最终会调用到Derived类的display函数，这里就体现出虚函数的作用了，这是怎么做到的呢，或者说虚函数底层是的怎么实现的呢？

3、虚函数的实现机制

虚函数的实现是怎样的呢？简单来说，就是通过一张**虚函数表**（Virtual Fucntion Table）实现的。具体地讲，当类中定义了一个虚函数后，会在该类创建的对象的存储布局的开始位置多一个虚函数指针（vfptr），该虚函数指针指向了一张虚函数表，而该虚函数表就像一个数组，表中存放的就是各虚函数的入口地址。如下图

当一个基类中设有虚函数，而一个派生类继承了该基类，并对虚函数进行了重定义，我们称之为**覆盖**(override). 这里的覆盖指的是派生类的虚函数表中相应虚函数的入口地址被覆盖。



虚函数机制是如何被激活的呢，或者说**动态多态是怎么表现出来的呢**？从上面的例子，可以得出结论：

1. 基类定义虚函数
2. 派生类重定义（覆盖、重写）虚函数
3. 创建派生类对象
4. 基类的指针指向派生类对象
5. 基类指针调用虚函数

4、哪些函数不能被设置为虚函数?

1. 普通函数（非成员函数）：定义虚函数的主要目的是为了重写达到多态，所以普通函数声明为虚函数没有意义，因此编译器在编译时就绑定了它。
2. 静态成员函数：静态成员函数对于每个类都只有一份代码，所有对象都可以共享这份代码，他不归某一个对象所有，所以它也没有动态绑定的必要。(静态函数发生在编译时，虚函数体现多态发生在运行时)
3. 内联成员函数：内联函数本就是为了减少函数调用的代价，所以在代码中直接展开。但虚函数一定要创建虚函数表，这两者不可能统一。另外，内联函数在编译时被展开，而虚函数在运行时才动态绑定。
4. 构造函数：这个原因很简单，主要从语义上考虑。因为构造函数本来是为了初始化对象成员才产生的，然而虚函数的目的是为了在完全不了解细节的情况下也能正确处理对象，两者根本不能“ 好好相处 ”。因为虚函数要对不同类型的对象产生不同的动作，如果将构造函数定义成虚函数，那么对象都没有产生，怎么完成想要的动作呢
5. 友元函数：当我们把一个函数声明为一个类的友元函数时，它只是一个可以访问类内成员的普通函数，并不是这个类的成员函数，自然也不能在自己的类内将它声明为虚函数。(当友元函数是成员函数的时候是可以设置为虚函数的，比如在自己类里面设置为虚函数，但是作为另一个类的友元)

5、虚函数的访问

5.1、指针访问

使用指针访问非虚函数时，编译器根据指针本身的类型决定要调用哪个函数，而不是根据指针指向的对象类型；
使用指针访问虚函数时，编译器根据指针所指对象的类型决定要调用哪个函数(**动态联编**)，而与指针本身的类型无关。

5.2、引用访问

使用引用访问虚函数，与使用指针访问虚函数类似，**表现出动态多态**特性。不同的是，引用一经声明后，引用变量本身无论如何改变，其调用的函数就不会再改变，始终指向其开始定义时的函数。因此在使用上有一定限制，但这在一定程度上提高了代码的安全性，特别体现在函数参数传递等场合中，可以将引用理解成一种“受限制的指针”。

5.3、对象访问

和普通函数一样，虚函数一样可以通过对象名来调用，此时编译器采用的是**静态联编**。通过对象名访问虚函数时，调用哪个类的函数取决于定义对象名的类型。对象类型是基类时，就调用基类的函数；对象类型是子类时，就调用子类的函数。

5.4、成员函数中访问

在类内的成员函数中访问该类层次中的虚函数，采用**动态联编**，要使用this指针。

5.5、构造函数和析构函数中访问

构造函数和析构函数是特殊的成员函数，在其中访问虚函数时，C++采用**静态联编**，即在构造函数或析构函数内，即使是使用“this->虚函数名”的形式来调用，编译器仍将其解释为静态联编的“本类名::虚函数名”。即它们所调用的虚函数是自己类中定义的函数，如果在自己的类中没有实现该函数，则调用的是基类中的虚函数。但绝不会调用任何在派生类中重定义的虚函数。

```
class Grandpa
{
public:
    Grandpa()
    {
        cout << "Grandpa()" << endl;
    }

    ~Grandpa()
    {
        cout << "~Grandpa()" << endl;
    }

    virtual
    void func1()
    {
        cout << "Grandpa::func1()" << endl;
    }

    virtual
    void func2()
    {
        cout << "Grandpa::func2()" << endl;
    }
};
```

```
26
27  class Father
28  : public Grandpa
29  {
30  public:
31      Father()
32      {
33          cout << "Father()" << endl;
34          func1();
35      }
36
37      ~Father()
38      {
39          cout << "~Father()" << endl;
40          func2();
41      }
42
43      virtual
44      void func1()
45      {
46          cout << "Father::func1()" << endl;
47      }
48
49      virtual
50      void func2()
51      {
52          cout << "Father::func2()" << endl;
53      }
54  };
55
56  class Son
57  : public Father
58  {
59  public:
60      Son()
61      {
62          cout << "Son()" << endl;
63      }
64
65      ~Son()
66      {
67          cout << "~Son()" << endl;
68      }
69
70      virtual void func1()
71      {
72          cout << "Son::func1()" << endl;
73      }
74
75      virtual void func2()
76      {
77          cout << "Son::func2()" << endl;
78      }
79  };
80
81  void test()
82  {
83      Son son;
```

```
}
```

接下来，我们看一个练习，大家思考一下，会输出什么呢?

```
//主要考虑点在绑定时机
class A
{
public:
    virtual
    void func(int val = 1)
    {
        cout << "A->" << val << endl;
    }

    virtual void test()
    {
        func();
    }
private:
    long _a;
};

class B
: public A
{
public:
    virtual
    void func(int val = 10)
    {
        cout << "B->" << val << endl;
    }
private:
    long _b;
};

int main(void)
{
    B b;
    A *p1 = (A*)&b;
    B *p2 = &b;
    p1->func();
    p2->func();

    return 0;
}
```

6、纯虚函数

纯虚函数是一种特殊的虚函数，在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，**它的实现留给该基类的派生类去做。**这就是纯虚函数的作用。纯虚函数的格式如下：

```
class 类名
{
public:
    virtual 返回类型 函数名(参数包) = 0;
};
```

设置纯虚函数的意义，就是让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“**你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它**”。

```
class Base
{
public:
    virtual void display() = 0;
};

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

声明纯虚函数的目的在于，提供一个与派生类一致的接口。

```
class Figure
{
public:
    virtual void display() const = 0;
    virtual double area() const = 0;
};

class Circle
: public Figure
{
public:
    explicit Circle(double radius)
    : _radius(radius)
    {

    }

    void display() const
    {
        cout << "Circle";
    }

    double area() const
    {
        return 3.14159 * _radius * _radius;
    }
private:
    double _radius;
```

```
};

class Rectangle
: public Figure
{
public:
    Rectangle(double length, double width)
    : _length(length)
    , _width(width)
    {

    }

    void display() const
    {
        cout << "Rectangle";
    }

    double area() const
    {
        return _length * _width;
    }
private:
    double _length;
    double _width;
};

class Triangle
: public Figure
{
public:
    Triangle(double a, double b, double c)
    : _a(a)
    , _b(b)
    , _c(c)
    {

    }

    void display() const
    {
        cout << "Triangle";
    }

    //海伦公式计算三角形的面积
    double area() const
    {
        double p = (_a + _b + _c) / 2;
        return  sqrt(p * (p - _a) * (p - _b) * (p - _c));
    }
private:
    double _a;
    double _b;
    double _c;
};
```

7、抽象类

一个类可以包含多个纯虚函数。只要类中含有一个纯虚函数，该类便为抽象类。一个抽象类只能作为基类来派生新类，不能创建抽象类的对象。

和普通的虚函数不同，在派生类中一般要对基类中纯虚函数进行重定义。如果该派生类没有对所有的纯虚函数进行重定义，则该派生类也会成为抽象类。这说明只有在派生类中给出了基类中所有纯虚函数的实现时，该派生类才不再是抽象类。

除此以外，还有另外一种形式的抽象类。对一个类来说，如果只定义了protected型的构造函数而没有提供public构造函数，无论是在外部还是在派生类中作为其对象成员都不能创建该类的对象，但可以由其派生出新的类，这种能派生新类，却不能创建自己对象的类是另一种形式的抽象类。

```
class Base
{
protected:
    Base(long base)
    : _base(base)
    {
        cout << "Base()" << endl;
    }
protected:
    long _base;
};

class Derived
: public Base
{
public:
    Derived(long base, long derived)
    : Base(base)
    , _derived(derived)
    {
        cout << "Derived(long, long)" << endl;
    }

    void print() const
    {
        cout << "_base:" << _base
             << ", _derived:" << _derived << endl;
    }
private:
    long _derived;
};

void test()
{
    Base base(1);//error
    Derived derived(1, 2);
}
```

8、虚析构函数

虽然构造函数不能被定义成虚函数，但析构函数可以定义为虚函数，一般来说，如果类中定义了虚函数，析构函数也应被定义为虚析构函数，尤其是类内有申请的动态内存，需要清理和释放的时候。

```cpp
class Base
{
public:
    Base(const char *pbase)
    : _pbase(new char[strlen(pbase) + 1]())
    {
        cout << "Base(const char *)" << endl;
        strcpy(_pbase, pbase);
    }

    /*virtual*/
    ~Base()
    {
        if(_pbase)
        {
            delete [] _pbase;
            _pbase = nullptr;
        }

        cout << "~Base()" << endl;
    }

private:
    char *_pbase;
};

class Derived
: public Base
{
public:
    Derived(const char *pbase, const char *pderived)
    : Base(pbase)
    , _pderived(new char[strlen(pderived) + 1]())
    {
        cout << "Derived(const char *, const char *)" << endl;
        strcpy(_pderived, pderived);
    }

    ~Derived()
    {
        cout << "~Derived()" << endl;
        if(_pderived)
        {
            delete [] _pderived;
            _pderived = nullptr;
        }
    }
private:
   char *_pderived;
};

void test()
{
    Base *pbase = new Derived("hello", "wuhan");
    pbase->print();
    delete pbase;
}
```

如上，在例子中，如果基类Base的析构函数没有设置为虚函数， 则在执行delete pbase;语句时，不会调用派生类Derived的析构函数，这样就会造成内存泄漏。此时，将基类Base的析构函数设置为虚函数，就可以解决该问题。

如果有一个基类的指针指向派生类的对象，并且想通过该指针delete派生类对象，系统将只会执行基类的析构函数，而不会执行派生类的析构函数。为避免这种情况的发生，往往把基类的析构函数声明为虚的，此时，系统将先执行派生类对象的析构函数，然后再执行基类的析构函数。

如果基类的析构函数声明为虚的，派生类的析构函数也将自动成为虚析构函数，无论派生类析构函数声明中是否加virtual关键字。

9、重载、隐藏、覆盖

重载：发生在同一个作用域中，函数名称相同，但参数的类型、个数、顺序不同（参数列表不一样）。

覆盖：发生在基类与派生类中，同名虚函数，参数列表亦完全相同。

隐藏：发生在基类与派生类中，指的是在某些情况下，派生类中的函数屏蔽了基类中的同名函数。(同名数据成员也有隐藏)

```
//针对于隐藏的例子
class Base
{
public:
    Base(int m)
    : _member(m)
    {
        cout << "Base(int)" << endl;
    }

    void func(int x)
    {
        cout << "Base::func(int)" << endl;
    }

    ~Base()
    {
        cout << "~Base()" << endl;
    }
protected:
    int _member;
};

class Derived
: public Base
{
public:
    Derived(int m1, int m2)
    : Base(m1)
    , _memeber(m2)
    {
        cout << "Derived(int, int)" << endl;
    }

    void func(int *)
    {
```

```
        cout << "_member: " << _member << endl;
        cout << "Derived::func(int*)" << endl;
    }

    ~Derived()
    {
        cout << "~Derived()" << endl;
    }
private:
    int _member;
};
```

10、测试虚表的存在

从前面的知识讲解，我们已经知道虚表的存在，但之前都是理论的说法，我们是否可以通过程序来验证呢？答案是肯定的。接下来我们看看下面的例子：

```
class Base
{
public:
    Base(long data1): _data1(data1)
    {

    }

    virtual
    void func1()
    {
        cout << "Base::func1()" << endl;
    }

    virtual
    void func2()
    {
        cout << "Base::func2()" << endl;
    }

    virtual
    void func3()
    {
        cout << "Base::func3()" << endl;
    }
protected:
    long _data1;
};

class Derived
: public Base
{
public:
    Derived(long data1, long data2)
    : _data1(data1)
    , _data2(data2)
    {

```

```
    }

    virtual
    void func1()
    {
        cout << "Derived::func1()" << endl;
    }

    virtual
    void func2()
    {
        cout << "Derived::func2()" << endl;
    }
private:
    long _data2;
};

void test()
{
    Derived derived(10, 100);
    long **pVtable = (long **)&derived;

    typedef void(* Function)();
    for(int idx = 0; idx < 3; ++idx)
    {
        Function f = (Function)pVtable[0][idx];
        f();
    }
}

```

以上例子充分说明了虚表的存在。那虚表到底存在什么位置呢？大家可以思考一下。

之前我们的例子都比较简单，接下来我们看看相对复杂一些的例子

11、带虚函数的多基派生

```
class Base1
{
public:
    Base1()
    : _iBase1(10)
    {

    }

    virtual
    void f()
    {
        cout << "Base1::f()" << endl;
    }

    virtual
    void g()
    {
```

```
        cout << "Base1::g()" << endl;
    }

    virtual
    void h()
    {
        cout << "Base1::h()" << endl;
    }
private:
    int _iBase1;
};

class Base2
{
public:
    Base2()
    : _iBase2(100)
    {

    }

    virtual void f()
    {
        cout << "Base2::f()" << endl;
    }

    virtual
    void g()
    {
        cout << "Base2::g()" << endl;
    }

    virtual
    void h()
    {
        cout << "Base2::h()" << endl;
    }
private:
    int _iBase2;
};

class Base3
{
public:
    Base3()
    : _iBase3(1000)
    {

    }

    virtual
    void f()
    {
        cout << "Base3::f()" << endl;
    }

    virtual
    void g()
```

```
    {
        cout << "Base3::g()" << endl;
    }

    virtual
    void h()
    {
        cout << "Base3::h()" << endl;
    }
private:
    int _iBase3;
};

class Derived
: public Base1
, public Base2
, public Base3
{
public:
    Derived()
    : _iDerived(10000)
    {

    }

    void f()
    {
        cout << "Derived::f()" << endl;
    }

    virtual
    void g1()
    {
        cout << "Derived::g1()" << endl;
    }
private:
    int _iDerived;
};

void test()
{
    Derived d;
    Base2 *pBase2 = &d;
    Base3 *pBase3 = &d;
    Derived *pDerived = &d;

    pBase2->f();
    cout << "sizeof(d) = " << sizeof(d) << endl;

    cout << "&Derived = " << &d << endl;
    cout << "pBase2 = " << pBase2 << endl;
    cout << "pBase3 = " << pBase3 << endl;
}

//结论：多重继承（带虚函数）
//1. 每个基类都有自己的虚函数表
//2. 派生类如果有自己的虚函数，会被加入到第一个虚函数表之中
//3. 内存布局中，其基类的布局按照基类被声明时的顺序进行排列
```

```
//4. 派生类会覆盖基类的虚函数，只有第一个虚函数表中存放的是
//    真实的被覆盖的函数的地址；其它的虚函数表中存放的并不是真实的
//    对应的虚函数的地址，而只是一条跳转指令
```

12、多基派生的二义性

```
class A
{
public:
    virtual
    void a()
    {
        cout << "a() in A" << endl;
    }

    virtual
    void b()
    {
        cout << "b() in A" << endl;
    }

    virtual
    void c()
    {
        cout << "c() in A" << endl;
    }
};

class B
{
public:
    virtual
    void a()
    {
        cout << "a() in B" << endl;
    }

    virtual
    void b()
    {
        cout << "b() in B" << endl;
    }

    void c()
    {
        cout << "c() in B" << endl;
    }

    void d()
    {
        cout << "d() in B" << endl;
    }
};

class C
```

```
: public A
, public B
{
public:
    virtual
    void a()
    {
        cout << "a() in C" << endl;
    }
    void c()
    {
        cout << "c() in C" << endl;
    }
    void d()
    {
        cout << "d() in C" << endl;
    }
};

void test()
{
    C c;
    printf("&c: %p\n", &c);
    c.b();
    cout << endl;

    A  *pA = &c;
    printf("pA: %p\n", pA);
    pA->a();
    pA->b();
    pA->c();
    cout << endl;

    B *pB = &c;
    printf("pB: %p\n", pB);
    pB->a();
    pB->b();
    pB->c();
    pB->d();
    cout << endl;

    C *pC = &c;
    printf("pC: %p\n", pC);
    pC->a();
    pC->b();//此处就是二义性
    pC->c();//此处的c()走的是虚函数机制还是非虚函数机制，如何判别？
    pC->d();//此处是隐藏，不是重写

    return 0;
}

```

13、虚拟继承

在讲虚函数之前，我们首先问大家一个问题：从语义上来讲，为什么**动态多态**和**虚继承**都使用virtual关键字来表示?

virtual在词典中的解释有两种：

1. Existing or resulting in essence or effect though not in actual fact, form, or name.实质上的，实际上的：虽然没有实际的事实、形式或名义，但在实际上或效果上存在或产生的；
2. Existing in the mind, especially as a product of the imagination. Used in literary criticism of text. 虚的，内心的：在头脑中存在的，尤指意想的产物，用于文学批评中。

C++中的virtual关键字采用第一个定义，即被virtual所修饰的事物或现象在本质上是存在的，但是没有直观的形式表现，无法直接描述或定义，需要通过其他的间接方式或手段才能够体现出其实际上的效果。关键就在于**存在**、**间接**和**共享**这三种特征：

- 虚函数是存在的
- 虚函数必须要通过一种间接的运行时（而不是编译时）机制才能够激活（调用）的函数
- 共享性表现在基类会共享被派生类重定义后的虚函数

那**虚拟继承**又是如何表现这三种特征的呢?

- 存在即表示虚继承体系和虚基类确实存在
- 间接性表现在当访问虚基类的成员时同样也必须通过某种间接机制来完成（通过**虚基表**来完成）
- 共享性表现在虚基类会在虚继承体系中被共享，而不会出现多份拷贝

虚拟继承是指在继承定义中包含了virtual关键字的继承关系。**虚基类**是指在虚继承体系中的通过virtual继承而来的基类。语法格式如下：

```
class Baseclass;
class Subclass
: public/private/protected virtual Baseclass
{
public:
    //...
private:
    //...
protected:
    //...
};
//其中Baseclass称之为Subclass的虚基类，而不是说Baseclass就是虚基类
```

接下来，我们看看例子：

```
#pragma vtordisp(off)
#include <iostream>
using std::cout;
using std::endl;

class A
{
public:
    A()
    : _ia(10)
    {

    }

    virtual
```

```
    void f()
    {
        cout << "A::f()" << endl;
    }
private:
    int _ia;
};

class B
: virtual public A
{
public:
    B()
    : _ib(20)
    {

    }

    void fb()
    {
        cout << "A::fb()" << endl;
    }

    /*virtual*/
    void f()
    {
        cout << "B::f()" << endl;
    }

    virtual
    void fb2()
    {
        cout << "B::fb2()" << endl;
    }
private:
    int _ib;
};

void test(void)
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    B b;
    return 0;
}

// 结论一：单个虚继承，不带虚函数
//  虚继承与继承的区别
//  1. 多了一个虚基指针
//  2. 虚基类位于派生类存储空间的最末尾

// 结论二：单个虚继承，带虚函数
//  1. 如果派生类没有自己的虚函数，此时派生类对象不会产生虚函数指针
//  2. 如果派生类拥有自己的虚函数，此时派生类对象就会产生自己本身的虚函数指针，
//      并且该虚函数指针位于派生类对象存储空间的开始位置
```

13.1、虚拟继承时派生类对象的构造和析构

在普通的继承体系中，比如A派生出B，B派生出C，则创建C对象时，在C类构造函数的初始化列表中调用B类构造函数，然后在B类构造函数初始化列表中调用A类的构造函数，即可完成对象的创建操作。但在虚拟继承中，则有所不同。

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

    A(int ia)
    : _ia(ia)
    {
        cout << "A(int)" << endl;
    }

    void f()
    {
        cout << "A::f()" << endl;
    }
protected:
    int _ia;
};

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

    B(int ia, int ib)
    : A(ia)
    , _ib(ib)
    {
        cout << "B(int,int)" << endl;
    }

protected:
    int _ib;
};

class C
: public B
{
public:
    C(int ia, int ib, int ic)
    : B(ia, ib)
    , _ic(ic)
    {
        cout << "C(int,int,int)" << endl;
```

```
    }

    void show() const
    {
        cout << "_ia: " << _ia << endl
             << "_ib: " << _ib << endl
             << "_ic: " << _ic << endl;
    }
private:
    int _ic;
};

void test()
{
    C c(10, 20, 30);
    c.show();
}

```

从最终的打印结果来看，在创建对象c的过程中，我们看到C带三个参数的构造函数执行了，同时B带两个参数的构造函数也执行了，但A带一个参数的构造函数没有执行，而是执行了A的默认构造函数。这与我们的预期是有差别的。如果想要得到预期的结果，我们还需要在C的构造函数初始化列表最后，显式调用A的相应构造函数。那为什么需要这样做呢?

在 C++ 中，如果继承链上存在虚继承的基类，则最底层的子类要负责完成该虚基类部分成员的构造。即我们需要显式调用虚基类的构造函数来完成初始化，如果**不显式调用**，则编译器会调用虚基类的**缺省构造函数**，不管初始化列表中次序如何，对虚基类构造函数的调用总是先于普通基类的构造函数。如果虚基类中**没有定义**的缺省构造函数，则会**编译错误。因为如果不这样做，虚基类部分会在存在的多个继承链上被多次初始化。**很多时候，对于继承链上的中间类，我们也会在其构造函数中显式调用虚基类的构造函数，因为一旦有人要创建这些中间类的对象，我们要保证它们能够得到正确的初始化。这种情况在菱形继承中非常明显， 我们接下来看看这种情况

13.2、菱形继承

```
#include <iostream>
using std::cout;
using std::endl;

class B
{
public:
    B()
    : _ib(10)
    , _cb('B')
    {
        cout << "B()" << endl;
    }

    B(int ib, char cb)
    : _ib(ib)
    , _cb(cb)
    {
        cout << "B(int,char)" << endl;
    }
```

```
21
22      virtual
23      void f()
24      {
25          cout << "B::f()" << endl;
26      }
27
28      virtual
29      void Bf()
30      {
31          cout << "B::Bf()" << endl;
32      }
33  private:
34      int _ib;
35      char _cb;
36  };
37
38  class B1
39  : virtual public B
40  {
41  public:
42      B1()
43      : _ib1(100)
44      , _cb1('1')
45      {
46
47      }
48
49      B1(int ib, char ic, int ib1, char cb1)
50      : B(ib, ic)
51      , _ib1(ib1)
52      , _cb1(cb1)
53      {
54          cout << "B1(int,char,int,char)" << endl;
55      }
56
57      virtual
58      void f()
59      {
60          cout << "B1::f()" << endl;
61      }
62
63      virtual
64      void f1()
65      {
66          cout << "B1::f1()" << endl;
67      }
68
69      virtual
70      void Bf1()
71      {
72          cout << "B1::Bf1()" << endl;
73      }
74
75  private:
76      int _ib1;
77      char _cb1;
78  };
```

```
79
80  class B2
81  : virtual public B
82  {
83  public:
84      B2()
85      : _ib2(1000)
86      , _cb2('2')
87      {
88
89      }
90
91      B2(int ib, char ic, int ib2, char cb2)
92      : B(ib, ic)
93      , _ib2(ib2)
94      , _cb2(cb2)
95      {
96          cout << "B2(int,char,int,char)" << endl;
97      }
98
99      //virtual
100     void f()
101     {
102         cout << "B2::f()" << endl;
103     }
104
105     //virtual
106     void f2()
107     {
108         cout << "B2::f2()" << endl;
109     }
110
111     //virtual
112     void Bf2()
113     {
114         cout << "B2::Bf2()" << endl;
115     }
116 private:
117     int _ib2;
118     char _cb2;
119 };
120
121 class D
122 : public B1
123 , public B2
124 {
125 public:
126     D()
127     : _id(10000)
128     , _cd('3')
129     {
130
131     }
132
133     D(int ib,  char cb
134       , int ib1, char ib
135       , int ib2, char cb2
136       , int id, char cd)
```

```
    : B1(ib1, ib1)
    , B2(ib2, cb2)
    , B(ib, cb)
    , _id(id)
    , _cd(cd)
    {
        cout << "D(...)" << endl;
    }

    virtual
    void f()
    {
        cout << "D::f()" << endl;
    }

    virtual
    void f1()
    {
        cout << "D::f1()" << endl;
    }

    virtual
    void f2()
    {
        cout << "D::f2()" << endl;
    }

    virtual
    void Df()
    {
        cout << "D::Df()" << endl;
    }
private:
    int _id;
    char _cd;
};

void test(void)
{
    D d;
    cout << sizeof(d) << endl;
    B1 *pb1 = &d;
    pb1->f();

    B2 *pb2 = &d;
    pb2->f();

    d.f();
    return 0;
}
//结论：虚基指针所指向的虚基表的内容
//  1. 虚基指针的第一条内容表示的是该虚基指针距离所在的子对象的首地址的偏移
//  2. 虚基指针的第二条内容表示的是该虚基指针距离虚基类子对象的首地址的偏移

```

如果是在若干类层次中，从虚基类直接或间接派生出来的派生类的构造函数初始化列表均有对该虚基类构造函数的调用，那么创建一个派生类对象的时候**只有该派生类列出的虚基类的构造函数被调用**，**其他类列出的将被忽略**，这样就保证虚基类的唯一副本只被初始化一次。即虚基类的构造函数只被执行一次。

对于虚继承的派生类对象的析构，析构函数的调用顺序为：

- 先调用派生类的析构函数；
- 然后调用派生类中成员对象的析构函数；
- 再调用普通基类的析构函数；
- 最后调用虚基类 的析构函数。

14、效率分析

通过以上的学习我们可以知道，多重继承和虚拟继承对象模型较单一继承复杂的对象模型，造成了成员访问低效率，表现在两个方面：对象构造时vptr的多次设定，以及this指针的调整。对于多种继承情况的效率比较如下：

继承情况	vptr是否设定	数据成员访问	虚函数访问	效率
单一继承	无	指针/对象/引用访问效率相同	直接访问	效率最高
单一继承	一次	指针/对象/引用访问效率相同	通过vptr和vtable访问	多态的引入，带来了设定`vptr`和间接访问虚函数等效率的降低
多重继承	多次	指针/对象/引用访问效率相同	通过vptr和vtable访问;通过第二或后继Base类指针访问，需要调整this指针	除了单一继承效率降低的情形，调整this指针也带来了效率的降低
虚拟继承	多次	指针/对象/引用访问效率降低	通过vptr和vtable访问;访问虚基类需要调整this指针	除了单一继承效率降低的情形，调整this指针也带来了效率的降低