第八章 多态

1. 什么叫多态?

多态（ 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。多态是面向对象的精髓。多态可以简单地概括为“一个接口，多种方法”。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。

通常是指对于同一个消息、同一种调用，在不同的场合，不同的情况下，执行不同的行为

。

2. 为什么需要多态性?

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，又称为静态联编。

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，也叫动态联编。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。

虚函数

虚函数的定义在一个成员函数的前面加上virtual关键字，该函数就成为虚函数

看这样一个例子：

基类和派生类中定义了同名的display函数

```
class Base{
public:
    Base(long x)
    : _base(x)
    {}

    void display() const{
        cout << "Base::display()" << endl;
    }
private:
    long _base;
};


class Derived
: public Base
{
public:
    Derived(long base,long derived)
    : Base(base)//创建基类子对象
    , _derived(derived)
    {}

    void display() const{
        cout << "Derived::display()" << endl;
    }
private:
    long _derived;

};

void print(Base * pbase){
    pbase->display();
}

void test0(){
    Base base(10);
    Derived dd(1,2);

    print(&base);
    cout << endl;
```

```
42      //用一个基类指针指向派生类对象
43      //能够操纵的只有基类部分
44      print(&dd);
45
46      cout << "sizeof(Base):" << sizeof(Base) << endl;
47      cout << "sizeof(Derived):" << sizeof(Derived) << endl;
48  }
49
```

得到的结果

```
Base::display()
Base::display()
sizeof(Base):8
sizeof(Derived):16
```

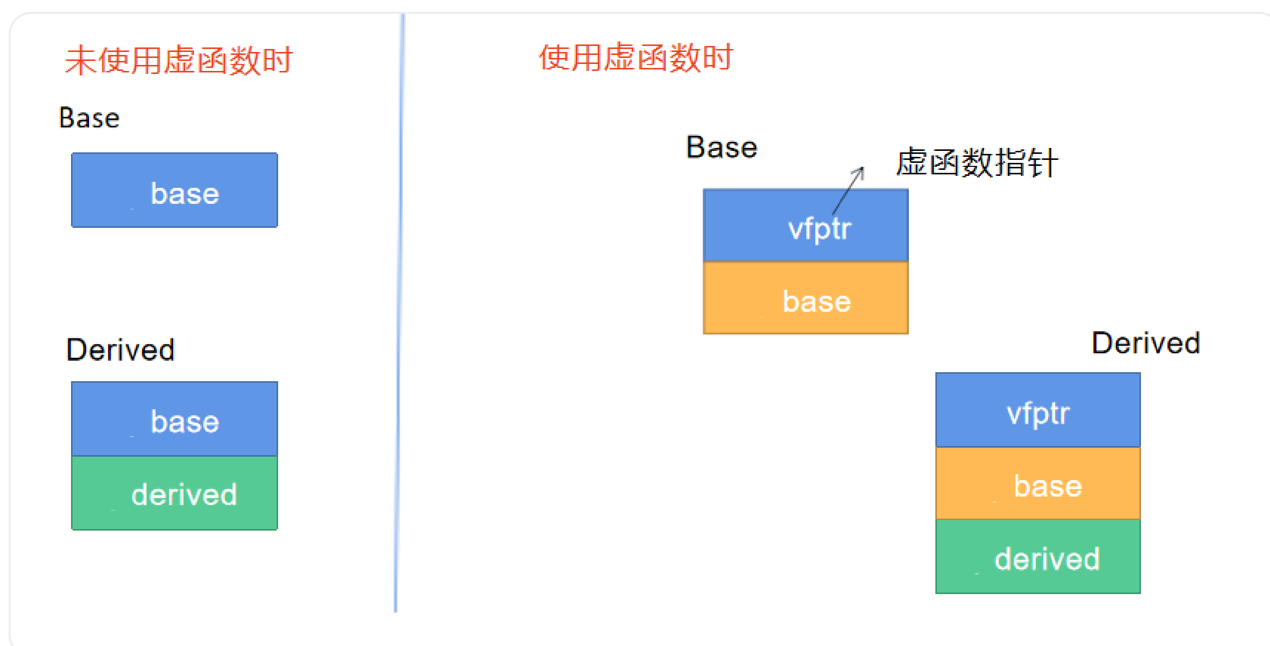
——给Base中的display函数加上virtual关键字修饰，得到的结果

```
Base::display()
Derived::display()
sizeof(Base):16
sizeof(Derived):24
```

从运行结果中我们发现，virtual关键字加入后，发生了一件“奇怪”的事情 —— 用基类指针指向派生类对象后，通过这个基类对象竟然可以调用派生类的成员函数。

而且，基类和派生类对象所占空间的大小都改变了，说明其内存结构发生了变化。

内存结构如下所示：

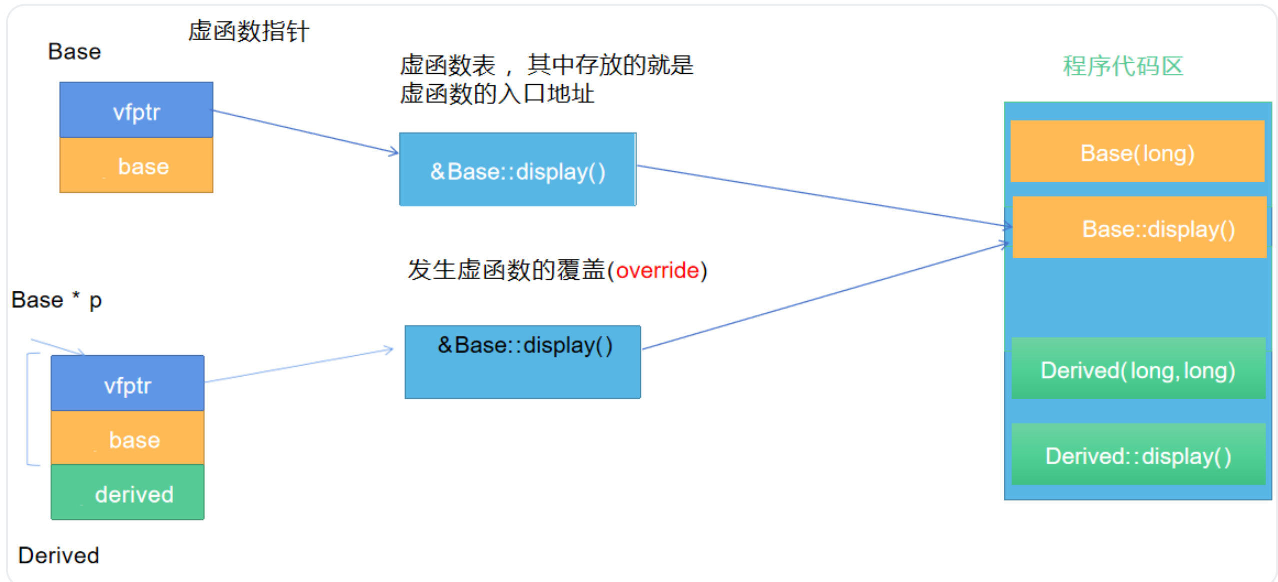


虚函数的实现原理

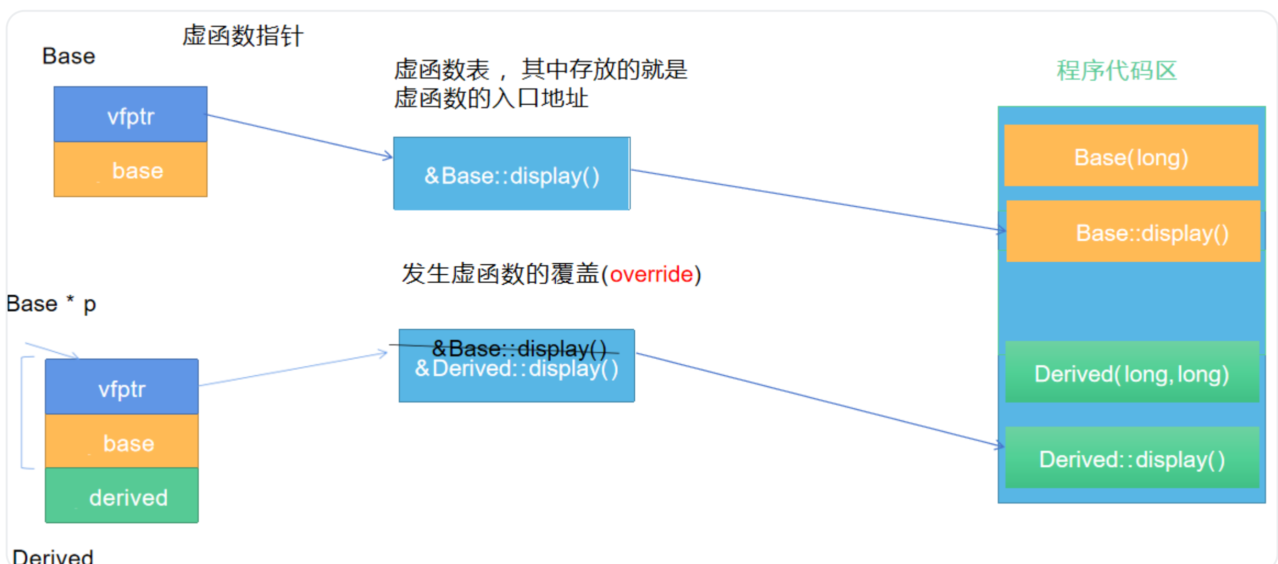
虚函数指针

当Base的display函数加上了virtual关键字，变成了一个虚函数，Base对象的存储布局就改变了。在存储的开始位置会多加一个虚函数指针，**该虚函数指针指向一张虚函数表**（简称虚表），其中存放的是虚函数的入口地址

Derived继承了Base类，那么创建一个Derived对象，依然会创建出一个Base类的基类子对象



在Derived类中又定义了display函数，发生了覆盖的机制（override），覆盖的是虚函数表中虚函数的入口地址



Base* p 去指向Derived对象，**依然只能访问到基类的部分**。用指针p去调用display函数，发现是一个虚函数，那么会**通过vfptr找到虚表**，此时虚表中存放的是Derived::display的入口地址，所以调用到Derived的display函数。

虚函数的覆盖

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。虚函数一般用于灵活拓展，所以需要派生类中对此虚函数进行覆盖。覆盖的格式有一定的要求：

- 与基类的虚函数有相同的函数名；
- 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。

我们在派生类中对虚函数进行覆盖时，很有可能写错函数的形式（函数名、返回类型、参数个数），等到要使用时才发现没有完成覆盖。这种错误很难发现，所以C++提供了关键字override来解决这一问题。

关键字override的作用：

在虚函数的函数参数列表之后，函数体的大括号之前，加上override关键字，告诉编译器此处定义的函数是要对基类的虚函数进行覆盖。

```
virtual void display() const{
    cout << "Base::display()" << endl;
}
```

```
27
28      void dispaly() const override{
29          cout << "Derived::display()" << endl;
30      }
31
```

```
1   class Base{
2   public:
3       virtual void display() const{
4           cout << "Base::display()" << endl;
5       }
6   private:
7       long _base;
8   };
9
10
```

```
class Derived
: public Base
{
public:
    //想要在派生类中定义虚函数覆盖基类的虚函数
    //很容易打错函数名字，同时又不会报错
    //没有完成有效的覆盖
    /* void dispaly() const{    //不会报错      */
    /* void dispaly() const override    //编译器会报错    */
    void display() const override
    {
        cout << "Derived::display()" << endl;
    }
private:
    long _derived;

};

```

覆盖 总结：

（1）覆盖是在虚函数之间的概念，需要派生类对象中定义的虚函数与基类中定义的虚函数的形式完全相同；

（2）当基类中定义了虚函数时，派生类去进行覆盖，即使在派生类的同名的成员函数前不加virtual，依然是虚函数；

（3）发生在基类派生类之间，基类与派生类中同时定义相同的虚函数 覆盖的是虚函数表中的入口地址，并不是覆盖函数本身。

动态多态（虚函数机制）被激活的条件（重点*）

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？其实激活条件还是比较严格的，需要满足以下全部要求：

1. 基类定义虚函数
2. **派生类中要覆盖虚函数** （覆盖的是虚函数表中的地址信息）
3. 创建派生类对象
4. **基类的指针指向派生类对象（或基类引用绑定派生类对象）**
5. **通过基类指针（引用）调用虚函数**

最终的效果：基类指针调用到了派生类实现的虚函数。

虚函数表*

在虚函数机制中virtual关键字的含义

1、虚函数是存在的；**（存在）**

2、通过间接的方式去访问；**（间接）**

3、通过基类的指针访问到派生类的函数，基类的指针共享了派生类的方法**（共享）**

如果没有虚函数，当通过pbase指针去调用一个普通的成员函数，那么就不会通过虚函数指针和虚表，直接到程序代码区中找到该函数；

有了虚函数，去找这个虚函数的方式就成了间接的方式。

对虚函数和虚函数表有了基本认知后，我们可以思考这样几个问题**（面试常考题）**

1、虚表存放在哪里?

编译完成时，虚表应该已经存在；在使用的过程中，虚函数表不应该被修改掉（如果能修改，将会找不到对应的虚函数）——应该存在只读段——具体位置不同厂家有不同实现。

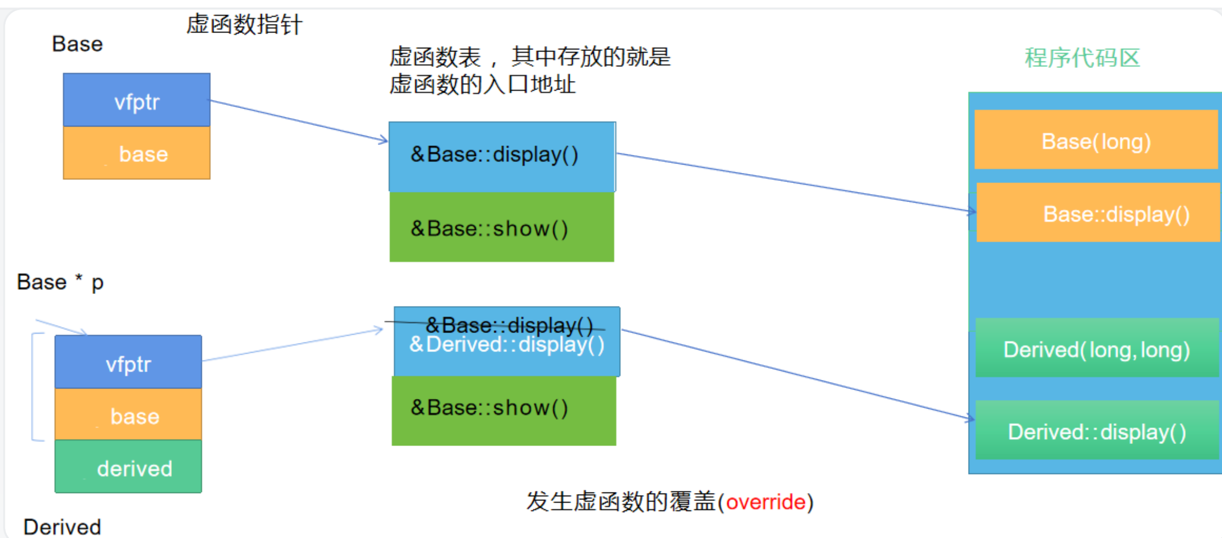
2、一个类中虚函数表有几张?

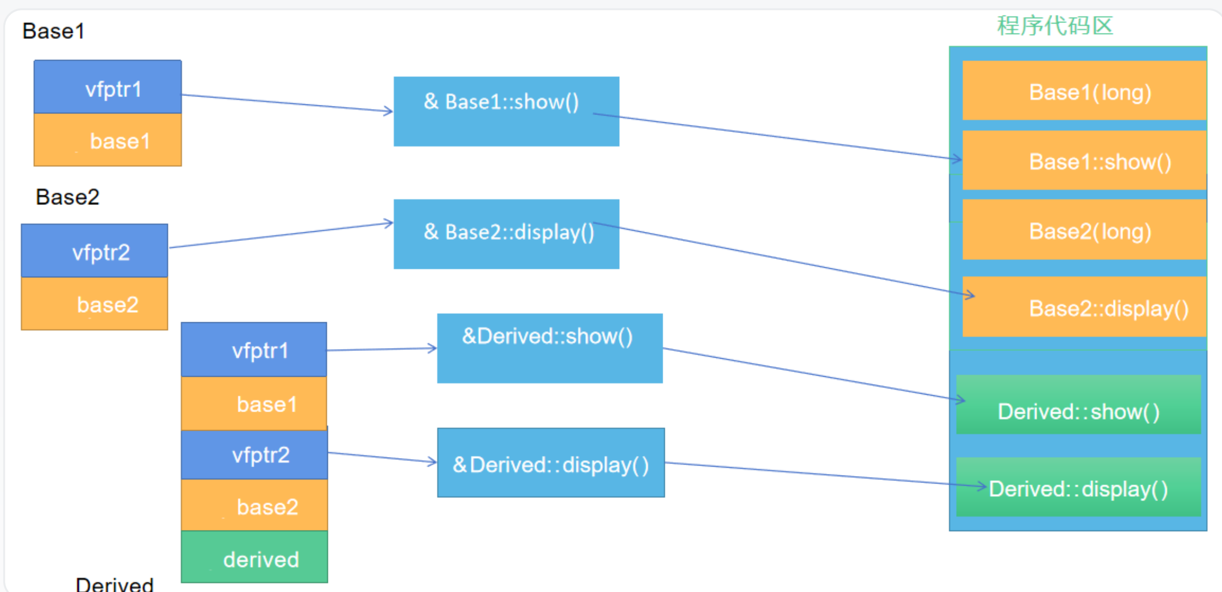
虚函数表（虚表）可以理解为是一个数组，存放的是一个个虚函数的地址

一个类可以没有虚函数表（没有虚函数就没有虚函数表）；

可以有一张虚函数表（即使这个类有多个虚函数，将这些虚函数的地址都存在虚函数表中）；

也可以有多张虚函数表（继承多个有虚函数的基类）





3、虚函数的底层实现是怎样的?

虚函数的底层是通过虚函数表实现的。当类中定义了虚函数之后，就会在对象的存储开始位置，多一个虚函数指针，该虚函数指针指向一张虚函数表，虚函数表中存储的是虚函数入口地址。

4. 三个概念的区分

重载（overload）： 发生在同一个类中， 当函数名称相同时， 函数参数类型、顺序、个数任一不同；

隐藏（oversee）： 发生在基类派生类之间， 函数名称相同时，就构成隐藏（参数不同也能构成隐藏）；

覆盖(override)： 发生在基类派生类之间，基类与派生类中同时定义相同的虚函数，覆盖的是虚函数表中的入口地址，并不是覆盖函数本身

虚函数的限制

虚函数机制给C++提供了灵活的用法，但仍然受到了一些约束，以下几种函数不能设为虚函数：

1.构造函数不能设为虚函数

构造函数的作用是创建对象，完成数据的初始化，而虚函数机制被激活的条件之一就是要先创建对象，有了对象才能表现出动态多态。如果将构造函数设为虚函数，那此时构造未执行完，对象还没创建出来，存在矛盾。

2.静态成员函数不能设为虚函数

虚函数的实际调用： ** this -> vfptr -> vtable -> virtual function **，但是静态成员函数没有this指针，所以无法访问到vfptr

3.Inline函数不能设为虚函数

因为inline函数在编译期间完成替换，而在编译期间无法展现动态多态机制，所以效果是冲突的如果同时存在，inline失效

4.普通函数不能设为虚函数

虚函数要解决的是对象多态的问题，与普通函数无关

虚函数的各种访问情况

虚函数机制的触发条件中规定了要**使用基类指针（或引用）来调用虚函数**，那么其他的调用方式会是什么情况呢？

1. **通过派生类对象直接调用虚函数**

并没有满足动态多态触发机制的条件，此时只是Derived中定义display函数对Base中的display函数发生了隐藏。

2. **在构造函数和析构函数中访问虚函数**

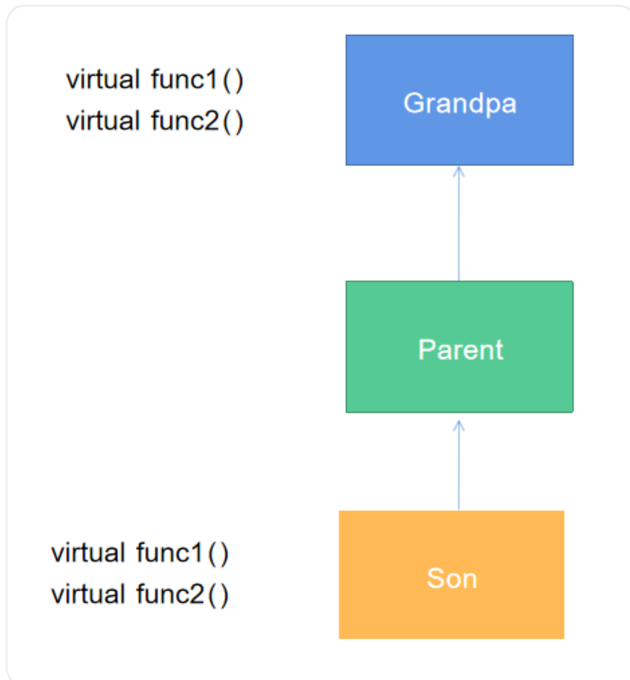
```
class Grandpa
{
public:
```

```
4       Grandpa(){ cout << "Grandpa()" << endl; }
5       ~Grandpa(){ cout << "~Grandpa()" << endl; }
6
7       virtual void func1() {
8           cout << "Grandpa::func1()" << endl;
9       }
10
11      virtual void func2(){
12          cout << "Grandpa::func2()" << endl;
13      }
14  };
15
16  class Parent
17  : public Grandpa
18  {
19  public:
20      Parent(){
21          cout << "Parent()" << endl;
22          //func1();//构造函数中调用虚函数
23      }
24
25      ~Parent(){
26          cout << "Parent()" << endl;
27          //func2();//析构函数中调用虚函数
28      }
29  };
30
31  class Son
32  : public Parent
33  {
34  public:
35      Son() { cout << "Son()" << endl; }
36      ~Son() { cout << "~Son()" << endl; }
37
38      virtual void func1() override {
39          cout << "Son::func1()" << endl;
40      }
41
42      virtual void func2() override{
43          cout << "Son::func2()" << endl;
44      }
45  };
46
47  void test0(){
48      Son ss;
49      Grandpa * p = &ss;
50      p->func1();
51      p->func2();
```

```
52  }
```

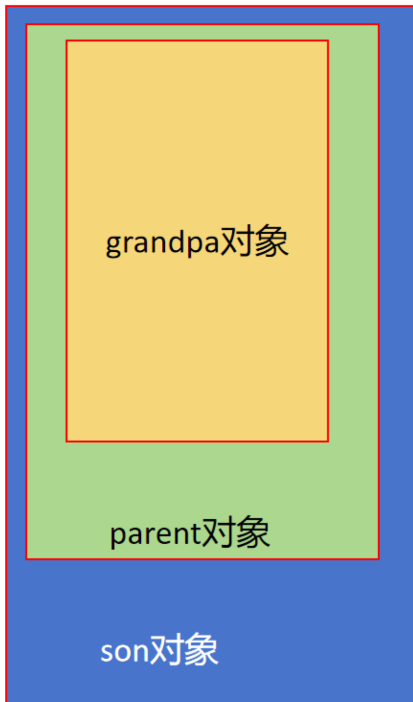
用Grandpa类指针p指向Son类对象，用这个指针p调用func1/func2.结果是指针p调用到的是Son类的func1和func2函数。

说明**即使Parent中没有对func1和fucn2覆盖，在Son中也可以对func1和func2覆盖。**



—— **如果在Parent类的构造和析构函数中调用虚函数**

创建一个Son对象




```
class Parent
: public Grandpa
{
public:
    Parent(){
        cout << "Parent()" << endl;
        func1();//构造函数中调用虚函数
    }

    ~Parent(){
        cout << "~parent()" << endl;
        func2();//析构函数中调用虚函数
    }
};
```

```
Grandpa()
Parent()
Grandpa::func1()
Son()
~Son()
~parent()
Grandpa::func2()
~Grandpa()
```

在parent的构造函数执行时，并不知道是在构造Son的对象，在此过程中，**只能看到本层及以上的部分**（因为Grandpa类的基类子对象已经创建完毕，虚表中记录了Grandpa::func1和func2的地址）

在Parent的析构函数执行时，此时Son的析构函数已经执行完了，**也只能看到本层及以上的部分。**

（表现的是静态联编）

——如果Parent类中也覆盖了func1和func2，那么会调用Parent本层的虚函数。

3. 在普通成员函数中调用虚函数

```
class Base{
public:
    Base(long x)
    : _base(x)
    {}

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

    void func1(){
        display();
        cout << _base << endl;
    }

```

```cpp
    void func2(){
        Base::display();
    }
private:
    long _base = 10;
};


class Derived
: public Base
{
public:
    Derived(long base,long derived)
    : Base(base)
    , _derived(derived)
    {}

    void display() const override{
        cout << "Derived::display()" << endl;
    }
private:
    long _derived;

};

void test0(){
    Base base(10);
    Derived derived(1,2);

    base.func1();
    base.func2();

    derived.func1(); //调用了Derived::display();
    derived.func2();

}
```

第1/2/4次调用，显然调用Base的display函数。

第3次调用的情况比较特殊：

derived对象调用func1函数，因为Derived类中没有重新定义自己的func1函数，所以回去调用基类子对象的func1函数。

可以理解为this指针此时发生了向上转型，成为了Base*类型。此时this指针还是指向的derived对象，就符合基类指针指向派生类对象的条件，在func1中调用虚函数display，触发动态多态机制。

抽象类

抽象类有两种形式:

1. 定义了纯虚函数的类，称为抽象类

2. 只定义了protected型构造函数的类，也称为抽象类

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 ...) = 0;
};
```

在基类中声明纯虚函数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。

多个派生类可以对纯虚函数进行多种不同的实现，但是都需要遵循基类给出的接口（纯虚函数的声明）。

定义了纯虚函数的类成为抽象类，抽象类不能实例化对象。

看一个简单例子:

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

class B
: public A
{
public:
    virtual void print() override{
        cout << "B::print()" << endl;
    }
```

```
15 };
16 
17 class C
18 : public B
19 {
20 public:
21     virtual void display() override{
22         cout << "C::display()" << endl;
23     }
24 };
25 
26 void test0(){
27     //A类定义了纯虚函数，A类是抽象类
28     //抽象类无法创建对象
29     //A a;//error
30 
31     //B b;//error
32     C c;
33     A * pa2 = &c;
34     pa2->print();
35     pa2->display();
36 }
```

在A类中声明纯虚函数，A类就是抽象类，无法创建对象；

在B类中去覆盖A类的纯虚函数，如果把所有的纯虚函数都覆盖了（都实现了），B类可以创建对象；只要还有一个纯虚函数没有实现，B类也会是抽象类，也无法创建对象；

再往下派生C类，完成所有的纯虚函数的实现，C类才能够创建对象。

最顶层的基类（定义纯虚函数的类）虽然无法创建对象，但是可以定义此类型的指针，指向派生类对象，去调用实现好的纯虚函数。

纯虚函数使用案例：

实现一个图形库，获取图形名称，获取图形之后获取它的面积

```
1  #define PI 3.141592653
2  class Figure{
3  public:
4      virtual string getName() const = 0;
5      virtual double getArea() const = 0;
6  };
7  
8  class Rectangle//矩形
9  : public Figure
10 {
```

```cpp
public:
    Rectangle(double len,double wid)
    : _length(len)
    , _width(wid)
    {}

    string getName() const override
    {
        return "矩形";
    }
    double getArea() const override
    {
        return _length * _width;
    }
private:
    double _length;
    double _width;
};

class Circle
: public Figure
{
public:
    Circle(double r)
    : _radius(r)
    {}

    string getName() const override
    {
        return "圆形";
    }
    double getArea() const override
    {
        return PI * _radius * _radius;
    }
private:
    double _radius;
};

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

```
59
60      string getName() const override
61      {
62          return "三角形";
63      }
64      double getArea() const override
65      {
66          double p = (_a + _b + _c)/2;
67          return sqrt(p * (p -_a) * (p - _b)* (p - _c));
68      }
69  private:
70      double _a,_b,_c;
71  };
72
```

基类Figure中定义纯虚函数，交给多个派生类去实现，最后可以使用基类的指针（引用）指向（绑定）不同类型的派生类对象，再去调用已经被实现的虚函数。

纯虚函数就是为了后续扩展而预留的接口。

只定义了protected构造函数的类

如果一个类只定义了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。

Derived类定义了protected属性的构造函数，Derived类也是抽象类，无法创建对象，**但是可以定义指针指向派生类对象**

```
1   class Base {
2   protected:
3       Base(int base): _base(base) { cout << "Base()" << endl; }
4   private:
5       int _base;
6   };
7   class Derived
8   : public Base {
9   public:
10      Derived(int base, int derived)
11      : Base(base)
12      , _derived(derived)
13      { cout << "Derived(int,int)" << endl; }
14
15      void print() const
16      {
17          cout << "_base:" << _base
```

```
18            << ", _derived:" << _derived << endl;
19        }
20    private:
21        int _derived;
22    };
23
24    void test()
25    {
26        Base base(1);//error
27        Derived derived(1, 2);
28    }
```

析构函数设为虚函数（重点）

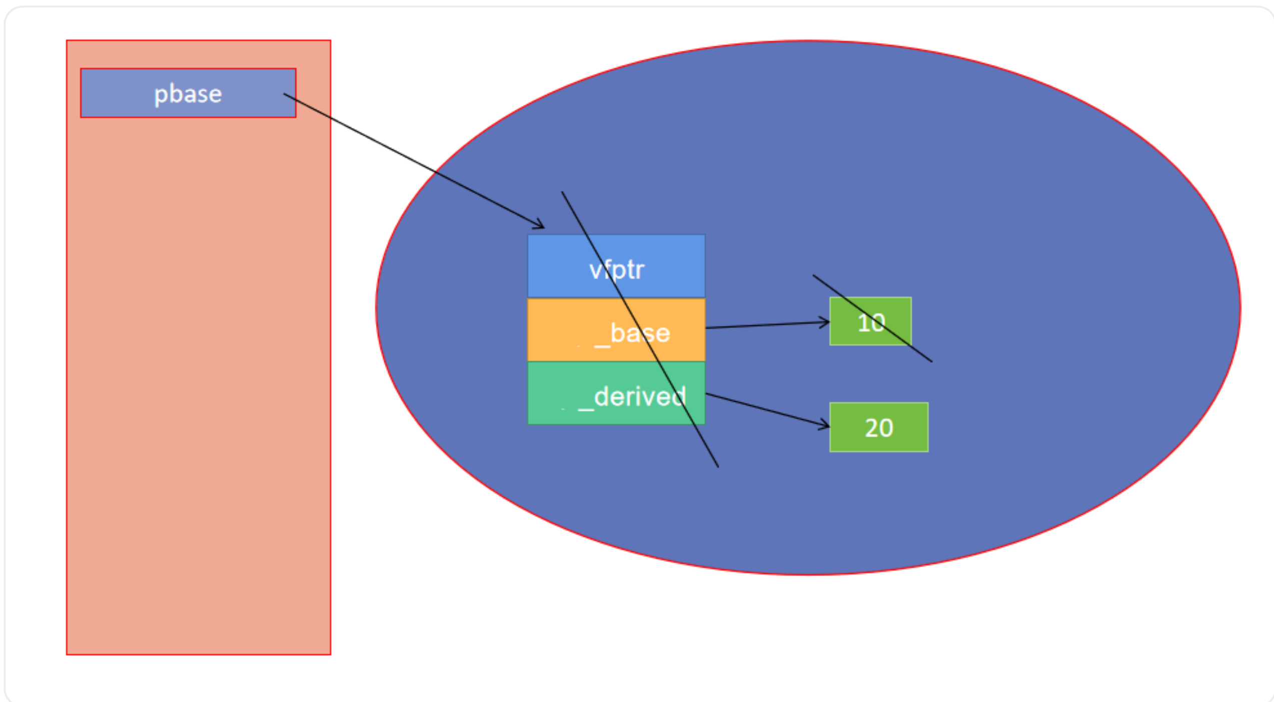
虽然构造函数不能被定义成虚函数，但析构函数可以定义为虚函数，一般来说，如果类中定义了虚函数，析构函数也应被定义为虚析构函数，尤其是类内有申请的动态内存，需要清理和释放的时候。

```
1    class Base
2    {
3    public:
4        Base()
5        : _base(new int(10))
6        { cout << "Base()" << endl; }
7
8        virtual void display() const{
9            cout << "*_base:" << *_base << endl;
10       }
11
12       ~Base(){
13           if(_base){
14               delete _base;
15               _base = nullptr;
16           }
17           cout << "~Base()" << endl;
18       }
19
20   private:
21       int * _base;
22   };
23
24   class Derived
25   : public Base
```

```
26  {
27  public:
28      Derived()
29      : Base()
30      , _derived(new int(20))
31      {
32          cout << "Derived()" << endl;
33      }
34  
35      virtual void display() const override{
36          cout << "*_derived:" << *_derived << endl;
37      }
38  
39      ~Derived(){
40          if(_derived){
41              delete _derived;
42              _derived = nullptr;
43          }
44          cout << "~Derived()" << endl;
45      }
46  
47  private:
48      int * _derived;
49  };
50  
51  void test0(){
52      Base * pbase = new Derived();
53      pbase->display();
54  
55      delete pbase;
56  }
```

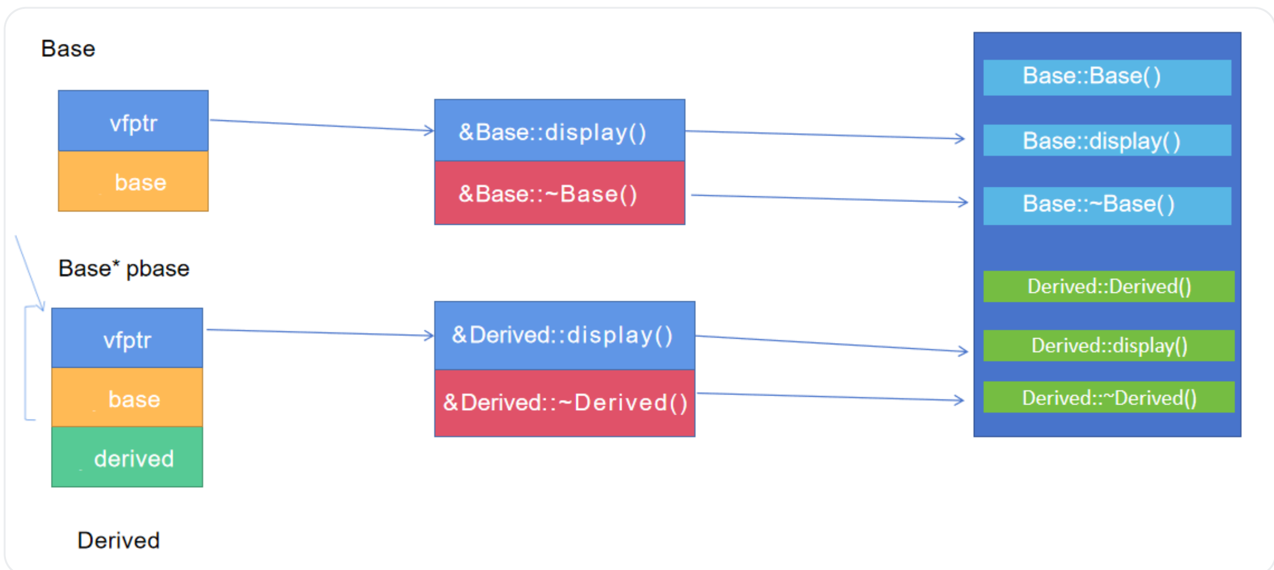
在执行delete pbase时的步骤：

首先会去调用Derived的析构函数，但是此时是通过一个Base类指针去调用，无法访问到，只能跳过，再去调用Base的析构函数，回收掉存放10这个数据的这片空间，最后调用operator delete回收掉堆对象本身所占的整片空间（编译器知道需要回收的是堆上的Derived对象，会自动计算应该回收多大的空间，与delete语句中指针的类别没有关系 —— delete pbase）



为了让基类指针能够调用派生类的析构函数，需要将Base的析构函数也设为虚函数。

Derived类中发生虚函数的覆盖，将Derived的虚函数表中记录的虚函数地址改变了。析构函数尽管不重名，也认为发生了覆盖。



总结：

在实际的使用中，如果有通过基类指针回收派生类对象的需求，都要将基类的析构函数设为虚函数。

建议：一个类定义了虚函数，就将它的析构函数设为虚函数。

验证虚表的存在（重点）

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？——当然可以

```cpp
class Base{
public:
 virtual void print() {
     cout << "Base::print()" << endl;
 }
 virtual void display() {
     cout << "Base::display()" << endl;
 }
 virtual void show() {
     cout << "Base::show()" << endl;
 }
private:
 long _base = 10;
};

class Derived
: public Base
{
public:
 virtual void print() {
     cout << "Derived::print()" << endl;
 }
 virtual void display() {
     cout << "Derived::display()" << endl;
 }
 virtual void show() {
     cout << "Derived::show()" << endl;
 }
private:
 long _derived = 100;
};

void test0(){
    Derived d;
 long * pDerived = reinterpret_cast<long*>(&d);
 cout << pDerived[0] << endl;
 cout << pDerived[1] << endl;
 cout << pDerived[2] << endl;

 cout << endl;
```

```
41    long * pVtable = reinterpret_cast<long*>(pDerived[0]);
42    cout << pVtable[0] << endl;
43    cout << pVtable[1] << endl;
44    cout << pVtable[2] << endl;
45
46    cout << endl;
47    typedef void (*Function)();
48    Function f = (Function)(pVtable[0]);
49    f();
50    f = (Function)(pVtable[1]);
51    f();
52    f = (Function)(pVtable[2]);
53    f();
54  }
```

创建一个Derived类对象d，这个对象的内存结构是由三个内容构成的，开始位置是虚函数指针，第二个位置是long型数据_base，

第三个位置是long型数据_derived.

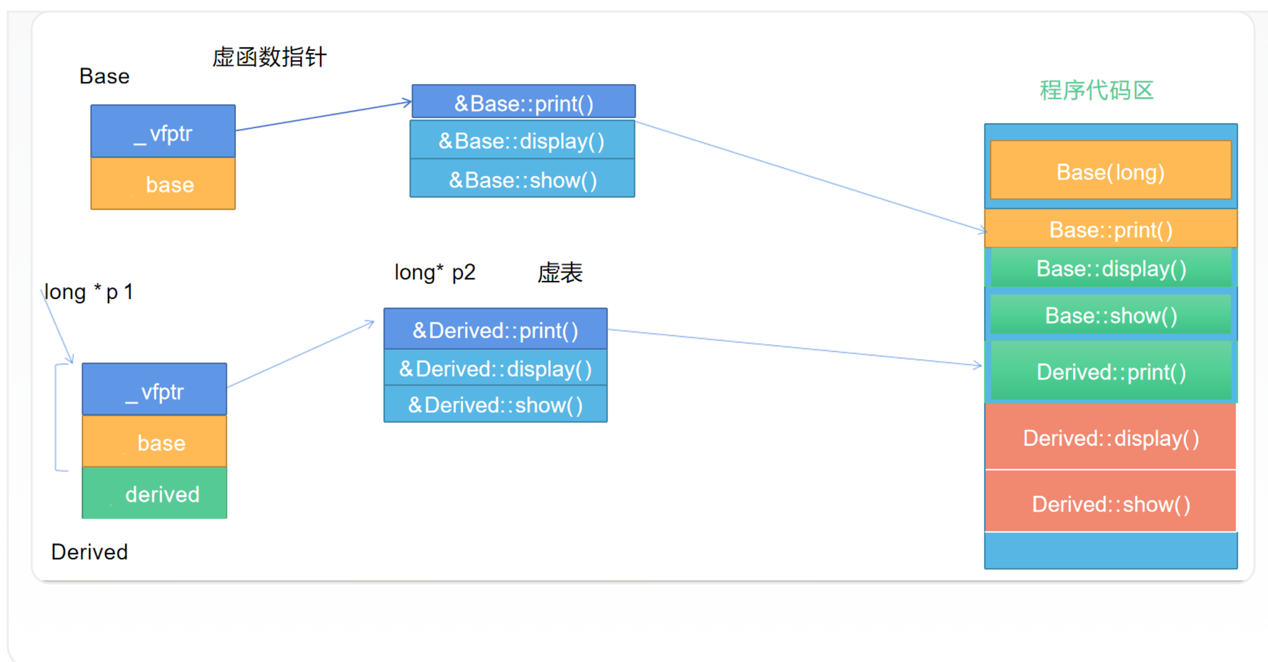
第一次强转将这个Derived类对象视为了存放三个long型元素的数组，打印这个数组中的三个元素，后两个本身就是long型数据，输出其值，第一个本身是指针（地址），打印出来的结果是编译器以long型数据来看待这个地址的值。

这个虚函数指针指向虚表，虚表中存放三个虚函数的入口地址（3 * 8字节），那么再将虚表视为存放三个long型元素的数组，第二次强转，直接输出数组的三个元素，得到的结果是编译器以long型数据来看待这三个函数地址的值。

虚表中的三个元素本身是函数指针，那么再将这个三个元素强转成相应类型的函数指针，就可以通过函数指针进行调用了。

——验证了虚表中存放虚函数的顺序，是按照声明顺序去存放的。

```
16
24
94458508807480
10
100
94458506706446
94458506706334
94458506706390
Derived::show()
Derived::print()
Derived::display()
```



带虚函数的多继承

描述：先是Base1、Base2、Base3都拥有虚函数f、g、h，Derived公有继承以上三个类，在Derived中覆盖了虚函数f，还有一个普通的成员函数g1，四个类各有一个double成员。

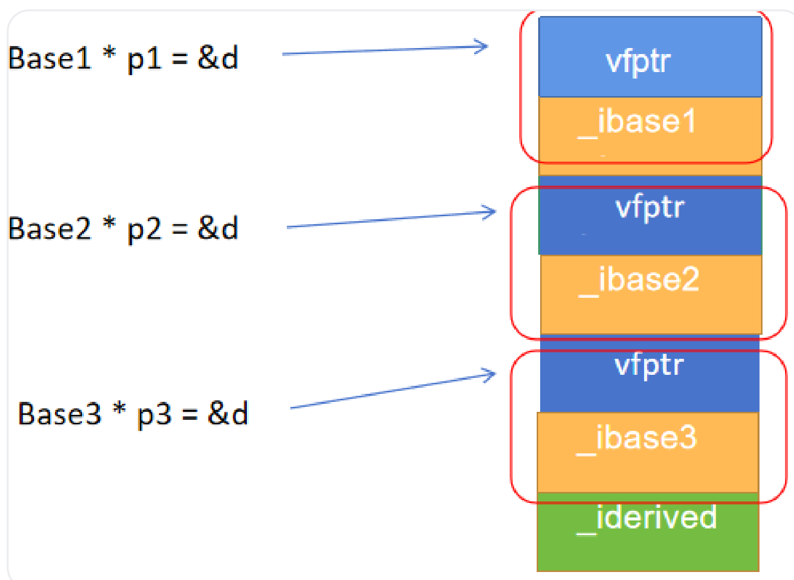
```
class Base1
{
public:
    Base1()
    : _iBase1(10)
    { cout << "Base1()" << endl; }
    virtual void f()
    {
        cout << "Base1::f()" << endl;
    }

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

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

```

```
22      virtual ~Base1() {}
23  private:
24      double _iBase1;
25  };
26
27  class Base2
28  {
29      //...
30  private:
31      double _iBase2;
32  };
33
34  class Base3
35  {
36  public:
37      //...
38  private:
39      double _iBase3;
40  };
41
42  class Derived
43      : public Base1
44      , public Base2
45      , public Base3
46  {
47  public:
48      Derived()
49      : _iDerived(10000)
50      { cout << "Derived()" << endl; }
51
52      void f()
53      {
54          cout << "Derived::f()" << endl;
55      }
56
57      void g1()
58      {
59          cout << "Derived::g1()" << endl;
60      }
61  private:
62      double _iDerived;
63  };
64
65  int main(void)
66  {
67      cout << sizeof(Derived) << endl;
68
69      Derived d;
```

```cpp
    Base1* pBase1 = &d;
    Base2* pBase2 = &d;
    Base3* pBase3 = &d;

    cout << "&Derived = " << &d << endl;
    cout << "pBase1 = " << pBase1 << endl;
    cout << "pBase2 = " <11 endl;

    return 0;
}
```

```
Base1()
Base2()
Base3()
Derived()
&Derived = 000000247157F6E8
pBase1 = 000000247157F6E8
pBase2 = 000000247157F6F8
pBase3 = 000000247157F708
```

三种不同的基类类型指针指向派生类对象时，实际指向的位置是基类子对象的位置



VS上验证布局和虚函数表存放的内容

```
1>class Derived size(56):
1>  +---
1> 0    | +--- (base class Base1)
1> 0    | | {vfptr}
1> 8    | | _iBase1
1>  | +---
1>16    | +--- (base class Base2)
1>16    | | {vfptr}
1>24    | | _iBase2
1>  | +---
1>32    | +--- (base class Base3)
1>32    | | {vfptr}
1>40    | | _iBase3
1>  | +---
1>48    | _iDerived
1>  +---
```

```
1>Derived::$vftable@Base1@:
1>  | &Derived_meta
1>  |  0
1> 0    | &Derived::f
1> 1    | &Base1::g
1> 2    | &Base1::h
1> 3    | &Derived::{dtor}
```

```
1>Derived::$vftable@Base2@:
1>  | -16
1> 0    | &thunk: this-=16; goto Derived::f
1> 1    | &Base2::g
1> 2    | &Base2::h
1> 3    | &thunk: this-=16; goto Derived::{dtor}
```

```
1>Derived::$vftable@Base3@:
1>  | -32
1> 0    | &thunk: this-=32; goto Derived::f
1> 1    | &Base3::g
1> 2    | &Base3::h
1> 3    | &thunk: this-=32; goto Derived::{dtor}
```

布局规则

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

```
1>Derived::$vftable@Base1@:
1>  | &Derived_meta
1>  |  0
1> 0    | &Derived::f
1> 1    | &Base1::g
1> 2    | &Base1::h
1> 3    | &Derived::{dtor}
1> 4    | &Derived::g1
```

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（有虚函数的基类会往上放——希望尽快访问到虚函数）

```
1>class Derived size(48):
1>  +---
1> 0     | +--- (base class Base2)
1> 0     | | {vfptr}
1> 8     | | _iBase2
1>   | +---
1>16     | +--- (base class Base3)
1>16     | | {vfptr}
1>24     | | _iBase3
1>   | +---
1>32     | +--- (base class Base1)
1>32     | | _iBase1
1>   | +---
1>40     | _iDerived
```

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；其它的虚函数表中对应位置存放的并不是真实的对应的虚函数的地址，而是一条跳转指令

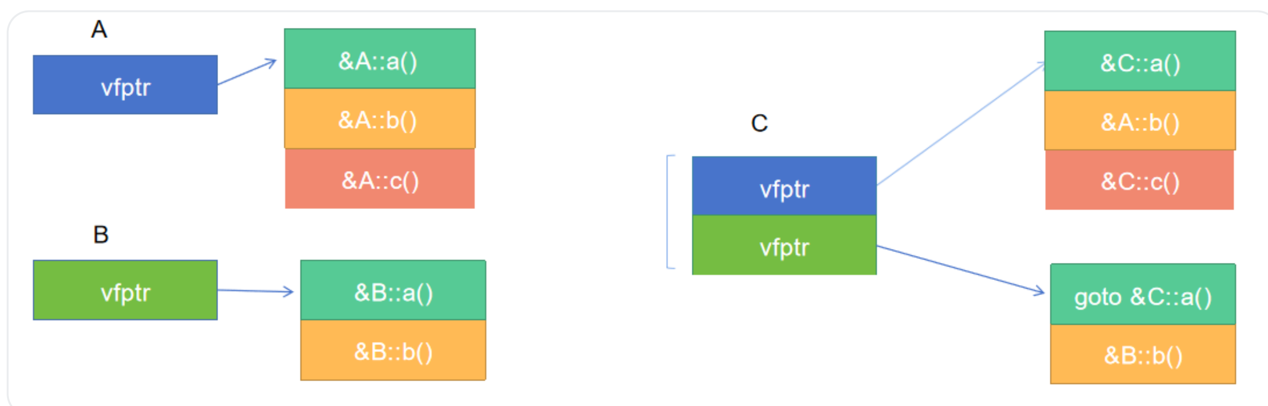
带虚函数的多重继承的二义性

例子：

```
1  class A{
2  public:
3      virtual void a(){ cout << "A::a()" << endl; }
4      virtual void b(){ cout << "A::b()" << endl; }
5      virtual void c(){ cout << "A::c()" << endl; }
6  };
7
8  class B{
9  public:
10     virtual void a(){ cout << "B::a()" << endl; }
11     virtual void b(){ cout << "B::b()" << endl; }
12     void c(){ cout << "B::c()" << endl; }
13     void d(){ cout << "B::d()" << endl; }
14 };
15
16 class C
17 : public A
18 , public B
19 {
20 public:
21     virtual void a(){ cout << "C::a()" << endl; }
```

```
22        void c(){ cout << "C::c()" << endl; }
23        void d(){ cout << "C::d()" << endl; }
24    };
25
26
27    //先不看D类
28    class D
29    : public C
30    {
31    public:
32        void c(){ cout << "D::c()" << endl; }
33    };
34
```

内存结构的示意图：



请分析以下各种调用情况的结果

```
1    void test0(){
2        C c;
3        c.a(); //C::a()  隐藏
4        c.b(); //冲突
5        c.c(); //C::c() 隐藏
6        c.d(); //C::d() 隐藏
7
8        cout << endl;
9        A* pa = &c;
10       pa->a(); //C::a() 覆盖
11       pa->b(); //A::b()
12       pa->c(); //C::c() 覆盖
13       pa->d(); //无法调用
14
15       cout << endl;
16       B* pb = &c;
17       pb->a(); //C::a() 覆盖
18       pb->b(); //B::b()
19       pb->c(); //B::c() 不是虚函数调用
```

```
    pb->d(); //B::d() 同上


    cout << endl;
    C * pc = &c;
    pc->a(); //C::a() 隐藏
    pc->b(); //冲突
    pc->c(); //C::c() 隐藏
    pc->d(); //C::d() 隐藏
}
```

——思考：pc->c() 这里的c函数是不是虚函数

从内存的角度分析，C::c()已经在第一张虚函数表中了，所以应该当成是虚函数处理。能否验证一下呢？

D类继承C类，重新定义c()函数，用C类指针指向D类对象，并调用c()函数

```
class D
: public C
{
public:
    void c(){ cout << "D::c()" << endl; }
};
```

```
void test1(){
    D d1;
    C * pc = &d1;
    pc->c();
}
```

```
ray@ubuntu:~/HaiBao/54th,
D::c()
```

如果将A类中c函数的virtual关键字去掉，毫无疑问C中c函数是一个普通函数（发生的是隐藏）

虚拟继承

虚函数 vs 虚拟继承

在虚函数机制（动态多态机制）中

1、虚函数是存在的；**（存在）**

2、通过间接的方式去访问；**（间接）**

3、通过基类的指针访问到派生类的函数，基类的指针共享了派生类的方法**（共享）**

（如果没有虚函数，当通过pbase指针去调用一个普通的成员函数，那么就不会通过虚函数指针和虚表，直接到程序代码区中找到该函数；有了虚函数，去找这个虚函数的方式就成了间接的方式）

虚拟继承同样使用virtual关键字（存在、间接、共享）

1、存在即表示虚继承体系和虚基类确实存在

2、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）

3、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

（虚基类的说法，如果B类虚拟继承了A类，那么说A类是B类虚基类，因为A类还可以以非虚拟的方式派生其他类）

补充：

（1）虚拟继承的内存结构

```cpp
class A
{
    void func() {}
    void run() { cout << "A::run()" << endl; }
    void run1() { cout << "A::run1()" << endl; }
    void run2() { cout << "A::run2()" << endl; }
    double a = 10;
};

class B : virtual public A
{
    void run() { cout << "B::run()" << endl; }
    void run1() { cout << "B::run1()" << endl; }
    double b = 1;
};
```

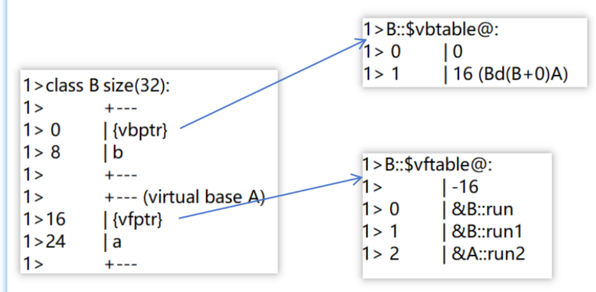
```
1>class B size(24):
1>        +---
1> 0      | {vbptr}
1> 8      | b
1>        +---
1>        +--- (virtual base A)
1>16      | a
1>        +---
```

```
1>B::$vbtable@:
1> 0      | 0
1> 1      | 16 (Bd(B+0)A)
```

虚基表中记录从虚基指针到基类子对象的成员需要偏转的信息

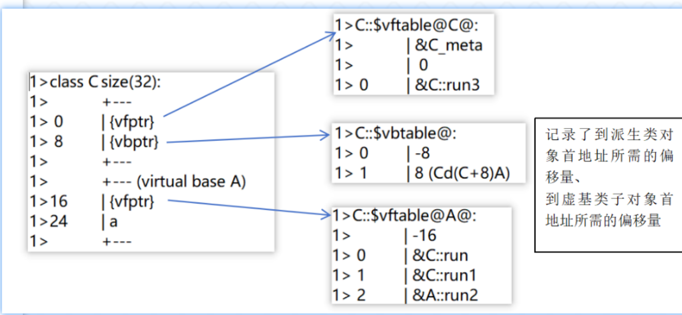
（2）如果虚基类中包含了虚函数

```cpp
class A
{
    void func() {}
    virtual void run() { cout << "A::run()" << endl; }
    virtual void run1() { cout << "A::run1()" << endl; }
    virtual void run2() { cout << "A::run2()" << endl; }
    double a = 10;
};

class B : virtual public A
{
    void run() { cout << "B::run()" << endl; }
    void run1() { cout << "B::run1()" << endl; }
    double b = 1;
};
```



（3）如果派生类中又定义了新的虚函数，会在内存中多出一个属于派生类的虚函数指针，指向一张新的虚表（VS的实现）

```cpp
class A
{
    void func() {}
    virtual void run() { cout << "A::run()" << endl; }
    virtual void run1() { cout << "A::run1()" << endl; }
    virtual void run2() { cout << "A::run2()" << endl; }
    double a = 10;
};

class C : virtual public A
{
    virtual void run() { cout << "C::run()" << endl; }
    virtual void run1() { cout << "C::run1()" << endl; }
    virtual void run3() { cout << "C::run3()" << endl; }
};
```



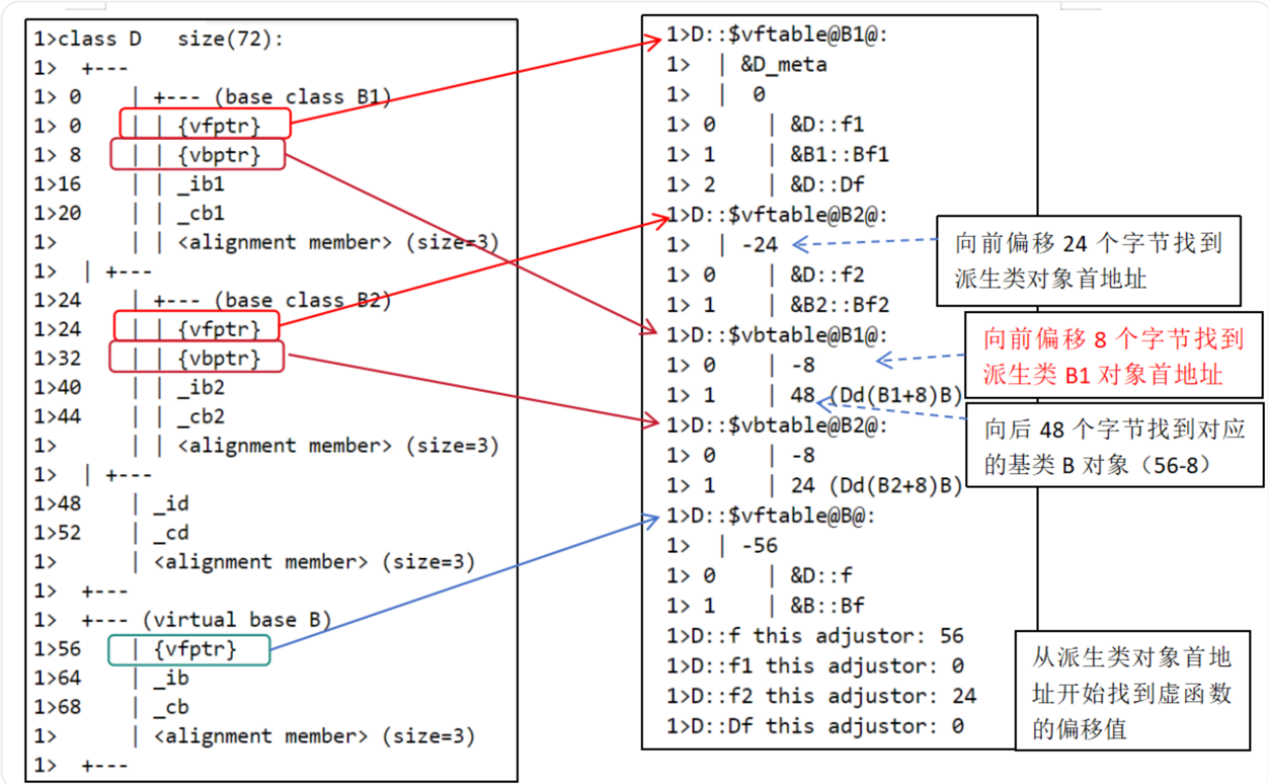
（4）带虚函数的菱形继承（拔高，不要求一定掌握）

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

    virtual void Bf()
    {
        cout << "B::Bf()" << endl;
    }
private:
    //double _b = 1;
    int _ib;
    char _cb;
};
```

```cpp
class B1 : virtual public B {
public:
    virtual void f() { cout << "B1::f()" << endl; }
    virtual void f1() { cout << "B1::f1()" << endl; }
    virtual void Bf1() { cout << "B1::Bf1()" << endl; }
private:
    //double _b1 = 10;
    int _ib1;
    char _cb1;
};
```

```cpp
class B2 : virtual public B {
public:
    virtual void f() { cout << "B2::f()" << endl; }
    virtual void f2() { cout << "B2::f2()" << endl; }
    virtual void Bf2() { cout << "B2::Bf2()" << endl; }
private:
    //double _b2 = 20;
    int _ib2;
    char _cb2;
```

```cpp
class D
: public B1
, public B2
{
public:
    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:
    //double _d = 100;
    int _id;
    char _cd;
};
```



虚拟继承时派生类对象的构造和析构

如下菱形继承的结构中，中间层基类虚拟继承了顶层基类，注意底层派生类的构造函数

```
1   class A
2   {
3   public:
4       A(double a)
5       : _a(a)
6       {
7           cout << "A(double)" << endl;
8       }
9
10      ~A(){cout << "~A()" << endl;}
11  private:
12      double _a = 10;
13  };
14
15  class B
16  : virtual public A
17  {
```

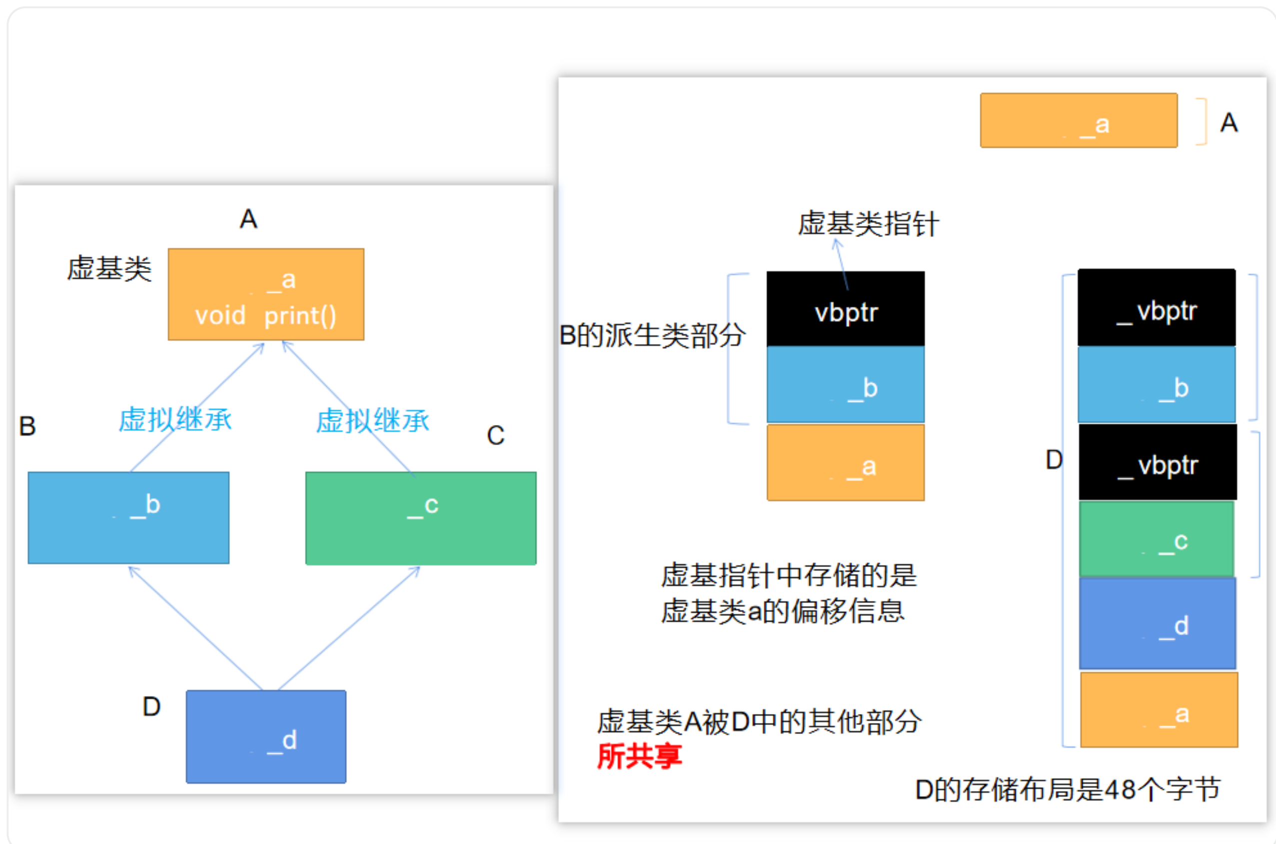
```
public:
    B(double a, double b)
    : A(a)
    , _b(b)
    {
        cout << "B(double,double)" << endl;
    }

    ~B(){ cout << "~B()" << endl; }
private:
    double _b;
};


class C
: virtual public A
{
public:
    C(double a, double c)
    : A(a)
    , _c(c)
    {
        cout << "C(double,double)" << endl;
    }

    ~C(){ cout << "~C()" << endl; }
private:
    double _c;
};

class D
: public B
, public C
{
public:
    D(double a,double b,double c,double d)
    : A(a)
    , B(a,b)
    , C(a,c)
    , _d(d)
    {
        cout << "D(double * 4)" << endl;
    }

    ~D(){ cout << "~D()" << endl; }
private:
    double _d;
};
```

```
class D
: public B
, public C
{
public:
    D(double a,double b,double c,double d)
    : A(a)
    , B(a,b)
    , C(a,c)
    , _d(d)
    {
        cout << "D(double * 4)" << endl;
    }
```

在虚拟继承的结构中，最底层的派生类不仅需要显式调用中间层基类的构造函数，还要在初始化列表最开始调用顶层基类的构造函数。

——那么A类构造岂不是会调用3次？

并不会，有了A类的构造之后会压抑B、C构造时调用A类构造，A类构造只会调用一次。可以对照菱形继承的内存模型理解，D类对象中只有一份A类对象的内容。



效率分析

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