Lesson02--多态

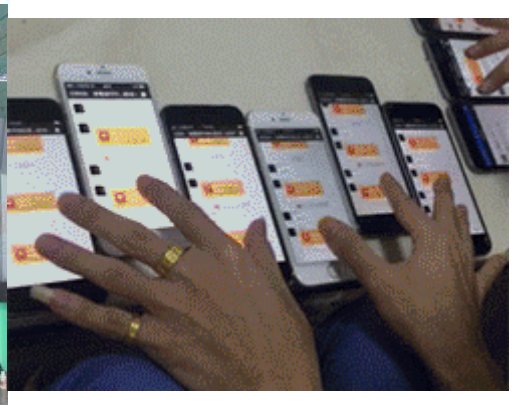
【本节目标】

- 1. 多态的概念
- 2. 多态的定义及实现
- 3. 抽象类
- 4. 多态的原理
- 5. 单继承和多继承关系中的虚函数表
- 6. 继承和多态常见的面试问题

前言

> **需要声明的，本节课件中的代码及解释都是在vs2013下的x86程序中，涉及的指针都是4bytes。如果要其他平台下，部分代码需要改动。比如：如果是x64程序，则需要考虑指针是8bytes问题等等**

1. 多态的概念

> ### 1.1 概念
>
> 多态的概念：通俗来说，就是多种形态，**具体点就是去完成某个行为，当不同的对象去完成时会产生出不同的状态。**
>
> 
>
> 
>
> 举个栗子：比如**买票这个行为**，当**普通人**买票时，是全价买票；**学生**买票时，是半价买票；**军人**买票时是优先买票。
>
> 再举个栗子： 最近为了**争夺在线支付市场**，支付宝年底经常会做诱人的**扫红包-支付-给奖励金**的活动。那么大家想想为什么有人扫的红包又大又新鲜8块、10块...，而有人扫的红包都是1毛，5毛....。其实这背后也是一个多态行为。支付宝首先会分析你的账户数据，比如你是新用户、比如你没有经常支付宝支付等等，那么你需要被鼓励使用支付宝，那么就你扫码金额 = random()%99；比如你经常使用支付宝支付或者支付宝账户中常年没钱，那么就不需要太鼓励你去使用支付宝，那么就你扫码金额 = random()%1；总结一下：**同样是扫码动作，不同的用户扫得到的不一样的红包，这也是一种多态行为。**ps：支付宝红包问题纯属瞎编，大家仅供娱乐。

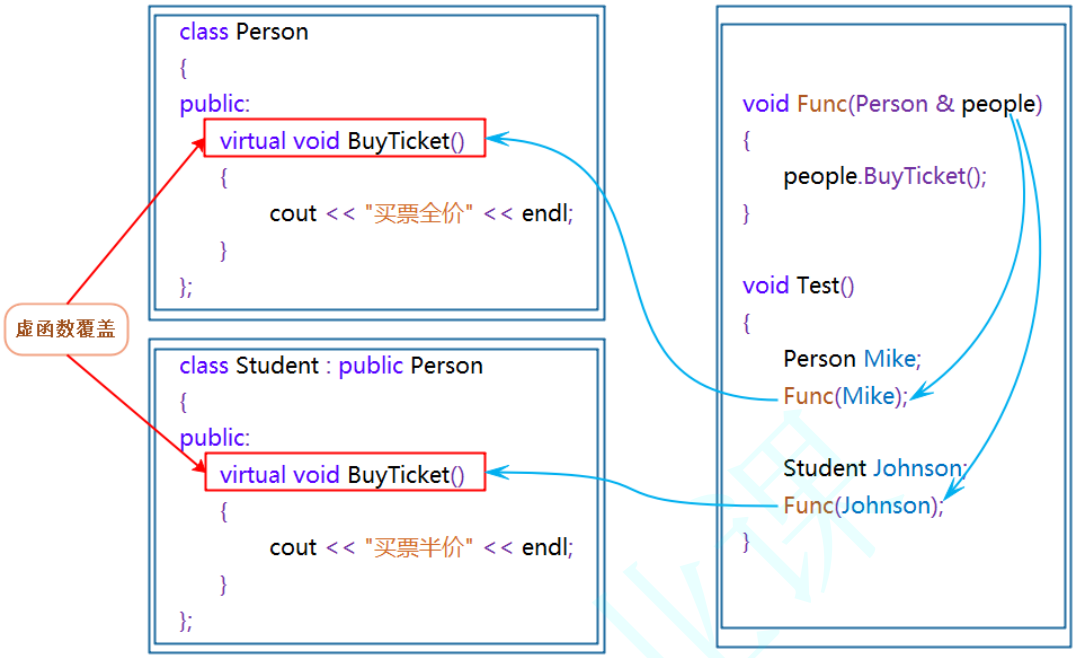
2. 多态的定义及实现

2.1多态的构成条件

多态是在不同继承关系的类对象，去调用同一函数，产生了不同的行为。比如Student继承了Person。Person对象买票全价，Student对象买票半价。

那么在继承中要**构成多态还有两个条件**：

1. 必须通过基类的指针或者引用调用虚函数
2. 被调用的函数必须是虚函数，且派生类必须对基类的虚函数进行重写



2.2 虚函数

虚函数：即被virtual修饰的类成员函数称为虚函数。

```
class Person {
public:
    virtual void BuyTicket() { cout << "买票-全价" << endl;}
};
```

2.3虚函数的重写

虚函数的重写(覆盖)：**派生类中有一个跟基类完全相同的虚函数(即派生类虚函数与基类虚函数的返回值类型、函数名字、参数列表完全相同)**，称子类的虚函数重写了基类的虚函数。

```
class Person {
public:
    virtual void BuyTicket() { cout << "买票-全价" << endl; }
};

class Student : public Person {
public:
    virtual void BuyTicket() { cout << "买票-半价" << endl; }

 /*注意：在重写基类虚函数时，派生类的虚函数在不加virtual关键字时，虽然也可以构成重写(因为继承后基类的虚函数被继承下来了在派生类依旧保持虚函数属性),但是该种写法不是很规范，不建议这样使用*/
 /*void BuyTicket() { cout << "买票-半价" << endl; }*/
};

void Func(Person& p)
```

```
{ p.BuyTicket(); }

int main()
{
 Person ps;
 Student st;

 Func(ps);
 Func(st);

    return 0;
}
```

虚函数重写的两个例外：

1. **协变(基类与派生类虚函数返回值类型不同)**

   派生类重写基类虚函数时，与基类虚函数返回值类型不同。即基类虚函数返回基类对象的指针或者引用，派生类虚函数返回派生类对象的指针或者引用时，称为协变。（了解）

```
class A{};
class B : public A {};

class Person {
public:
    virtual A* f() {return new A;}
};

class Student : public Person {
public:
    virtual B* f() {return new B;}
};
```

2. **析构函数的重写(基类与派生类析构函数的名字不同)**

   如果基类的析构函数为虚函数，此时派生类析构函数只要定义，无论是否加virtual关键字，都与基类的析构函数构成重写，虽然基类与派生类析构函数名字不同。虽然函数名不相同，看起来违背了重写的规则，其实不然，这里可以理解为编译器对析构函数的名称做了特殊处理，编译后析构函数的名称统一处理成destructor。

```
class Person {
public:
    virtual ~Person() {cout << "~Person()" << endl;}
};

class Student : public Person {
public:
    virtual ~Student() { cout << "~Student()" << endl; }
};

// 只有派生类Student的析构函数重写了Person的析构函数，下面的delete对象调用析构函数，才能构成多态，才能保证p1和p2指向的对象正确的调用析构函数。
int main()
{
    Person* p1 = new Person;
    Person* p2 = new Student;
```

```
    delete p1;
    delete p2;

    return 0;
}
```

2.4 C++11 override 和 final

从上面可以看出，C++对函数重写的要求比较严格，但是有些情况下由于疏忽，可能会导致函数名字母次序写反而无法构成重载，而这种错误在编译期间是不会报出的，只有在程序运行时没有得到预期结果才来debug会得不偿失，因此：C++11提供了override和final两个关键字，可以帮助用户检测是否重写。

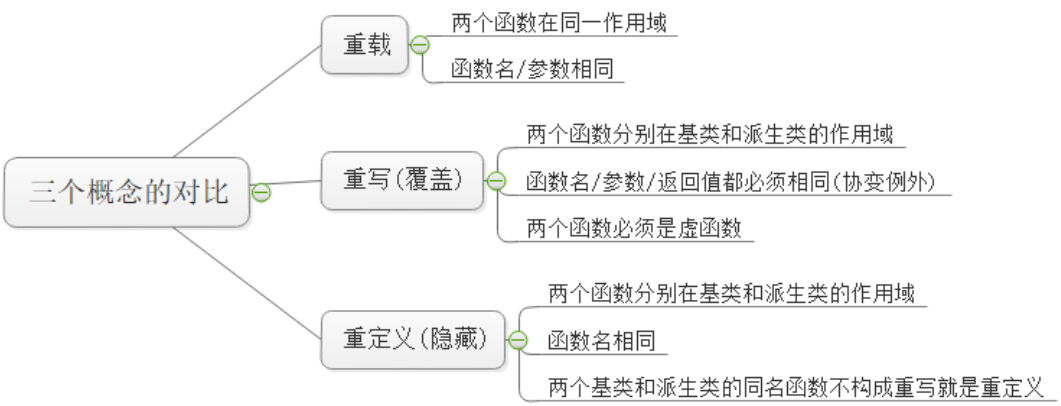
1. **final：修饰虚函数，表示该虚函数不能再被重写**

```
class Car
{
public:
    virtual void Drive() final {}
};

class Benz :public Car
{
public:
    virtual void Drive() {cout << "Benz-舒适" << endl;}
};
```

2. **override: 检查派生类虚函数是否重写了基类某个虚函数，如果没有重写编译报错。**

```
class Car{
public:
    virtual void Drive(){}
};

class Benz :public Car {
public:
    virtual void Drive() override {cout << "Benz-舒适" << endl;}
};
```

2.5 重载、覆盖(重写)、隐藏(重定义)的对比



3. 抽象类

3.1 概念

> 在虚函数的后面写上 =0 ，则这个函数为纯虚函数。**包含纯虚函数的类叫做抽象类（也叫接口类），抽象类不能实例化出对象。**派生类继承后也不能实例化出对象，只有重写纯虚函数，派生类才能实例化出对象。纯虚函数规范了派生类必须重写，另外纯虚函数更体现出了接口继承。

```
class Car
{
public:
 virtual void Drive() = 0;
};

class Benz :public Car
{
public:
    virtual void Drive()
 {
    cout << "Benz-舒适" << endl;
 }
};

class BMW :public Car
{
public:
    virtual void Drive()
 {
    cout << "BMW-操控" << endl;
 }
};

void Test()
{
 Car* pBenz = new Benz;
    pBenz->Drive();

    Car* pBMW = new BMW;
    pBMW->Drive();
}
```

3.2 接口继承和实现继承

> 普通函数的继承是一种实现继承，派生类继承了基类函数，可以使用函数，继承的是函数的实现。虚函数的继承是一种接口继承，派生类继承的是基类虚函数的接口，目的是为了重写，达成多态，继承的是接口。所以如果不实现多态，不要把函数定义成虚函数。

4.多态的原理

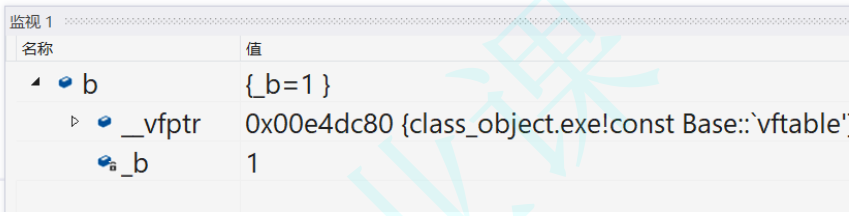
4.1虚函数表

比特就业课

```cpp
// 这里常考一道笔试题：sizeof(Base)是多少？
class Base
{
public:
    virtual void Func1()
    {
        cout << "Func1()" << endl;
    }

private:
    int _b = 1;
};
```

通过观察测试我们发现b对象是8bytes，**除了_b成员，还多一个__vfptr放在对象的前面(注意有些平台可能会放到对象的最后面，这个跟平台有关)，对象中的这个指针我们叫做虚函数表指针(v代表virtual，f代表function)**。一个含有虚函数的类中都至少都有一个虚函数表指针，因为虚函数的地址要被放到虚函数表中，虚函数表也简称虚表，。那么派生类中这个表放了些什么呢？我们接着往下分析

```cpp
int main()
{
    Base b;

    return 0;
}
```

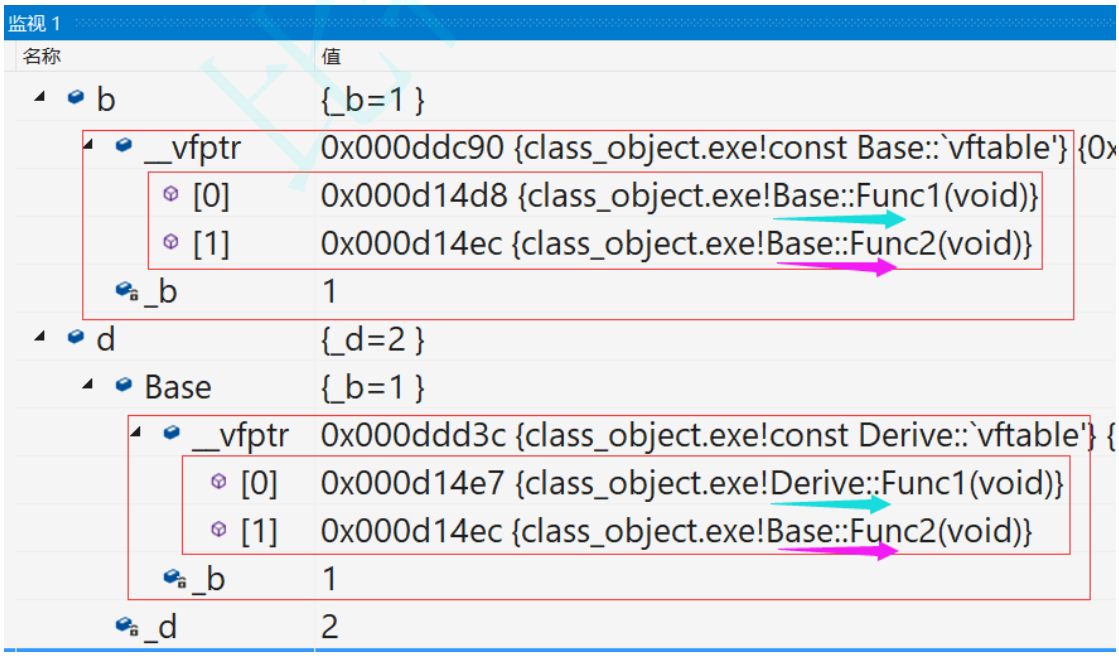



```cpp
// 针对上面的代码我们做出以下改造
// 1.我们增加一个派生类Derive去继承Base
// 2.Derive中重写Func1
// 3.Base再增加一个虚函数Func2和一个普通函数Func3
class Base
{
public:
    virtual void Func1()
    {
        cout << "Base::Func1()" << endl;
    }

    virtual void Func2()
    {
        cout << "Base::Func2()" << endl;
    }

    void Func3()
    {
        cout << "Base::Func3()" << endl;
    }

private:
    int _b = 1;
};

class Derive : public Base
{
public:
    virtual void Func1()
    {
```

```
        cout << "Derive::Func1()" << endl;
    }
private:
    int _d = 2;
};

int main()
{
    Base b;
    Derive d;

    return 0;
}
```

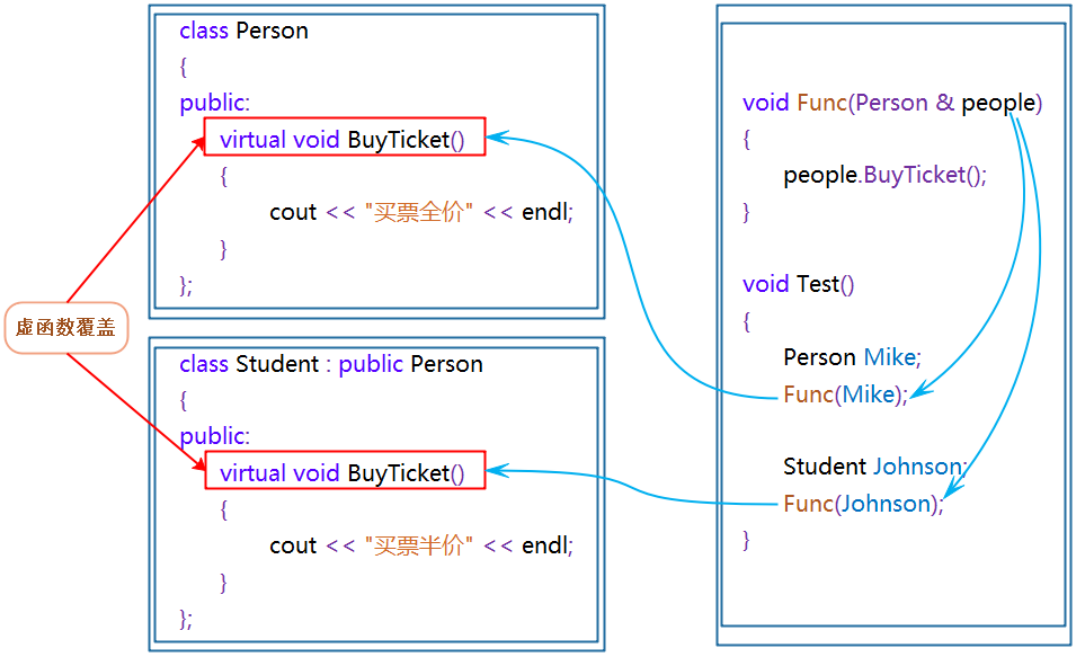
通过观察和测试，我们发现了以下几点问题：

1. 派生类对象d中也有一个虚表指针，d对象由两部分构成，一部分是父类继承下来的成员，虚表指针也就是存在部分的另一部分是自己的成员。
2. 基类b对象和派生类d对象虚表是不一样的，这里我们发现**Func1完成了重写，所以d的虚表中存的是重写的Derive::Func1，所以虚函数的重写也叫作覆盖**，覆盖就是指虚表中虚函数的覆盖。重写是语法的叫法，覆盖是原理层的叫法。
3. 另外Func2继承下来后是虚函数，所以放进了虚表，Func3也继承下来了，但是不是虚函数，所以不会放进虚表。
4. 虚函数表本质是一个存虚函数指针的指针数组，一般情况这个数组最后面放了一个nullptr。
5. 总结一下派生类的虚表生成：a.先将基类中的虚表内容拷贝一份到派生类虚表中 b.如果派生类重写了基类中某个虚函数，用派生类自己的虚函数覆盖虚表中基类的虚函数 c.派生类自己新增加的虚函数按其在派生类中的声明次序增加到派生类虚表的最后。
6. 这里还有一个童鞋们很容易混淆的问题：**虚函数存在哪的？虚表存在哪的？ 答：虚函数存在虚表，虚表存在对象中。注意上面的回答的错的。**但是很多童鞋都是这样深以为然的。注意**虚表存的是虚函数指针，不是虚函数**，虚函数和普通函数一样的，都是存在代码段的，只是他的指针又存到了虚表中。另外对象中存的不是虚表，存的是虚表指针。那么**虚表存在哪的呢？实际我们去验证一下会发现vs下是存在代码段的，Linux g++下大家自己去验证？**



4.2多态的原理

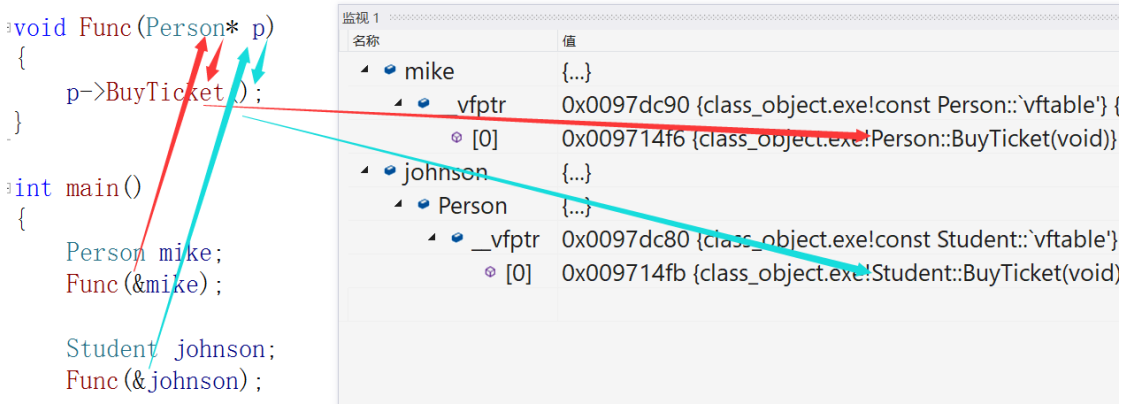
上面分析了这个半天了那么多态的原理到底是什么？还记得这里Func函数传Person调用的Person::BuyTicket，传Student调用的是Student::BuyTicket




```
class Person {
public:
    virtual void BuyTicket() { cout << "买票-全价" << endl; }
};

class Student : public Person {
public:
    virtual void BuyTicket() { cout << "买票-半价" << endl; }
};

void Func(Person& p)
{
 p.BuyTicket();
}

int main()
{
 Person Mike;
    Func(Mike);

 Student Johnson;
 Func(Johnson);

    return 0;
}
```

1. 观察下图的红色箭头我们看到，p是指向mike对象时，p->BuyTicket在mike的虚表中找到虚函数是Person::BuyTicket。
2. 观察下图的蓝色箭头我们看到，p是指向johnson对象时，p->BuyTicket在johson的虚表中找到虚函数是Student::BuyTicket。
3. 这样就实现出了不同对象去完成同一行为时，展现出不同的形态。
4. 反过来思考我们要达到多态，有两个条件，一个是虚函数覆盖，一个是对象的指针或引用调用虚函数。反思一下为什么？
5. 再通过下面的汇编代码分析，**看出满足多态以后的函数调用，不是在编译时确定的，是运行起来以后到对象的中取找的。不满足多态的函数调用时编译时确认好的。**

比特就业课




```
void Func(Person* p)
{
    p->BuyTicket();
}

int main()
{
    Person mike;
    Func(&mike);
    mike.BuyTicket();

    return 0;
}

// 以下汇编代码中跟你这个问题不相关的都被去掉了
void Func(Person* p)
{
...
    p->BuyTicket();
// p中存的是mike对象的指针，将p移动到eax中
001940DE  mov         eax,dword ptr [p]
// [eax]就是取eax值指向的内容，这里相当于把mike对象头4个字节(虚表指针)移动到了edx
001940E1  mov         edx,dword ptr [eax]
// [edx]就是取edx值指向的内容，这里相当于把虚表中的头4字节存的虚函数指针移动到了eax
00B823EE  mov         eax,dword ptr [edx]
// call eax中存虚函数的指针。这里可以看出满足多态的调用，不是在编译时确定的，是运行起来以后到对象的中取找的。
001940EA  call        eax
00头1940EC  cmp         esi,esp
}

int main()
{
...
// 首先BuyTicket虽然是虚函数，但是mike是对象，不满足多态的条件，所以这里是普通函数的调用转换成地址时，是在编译时已经从符号表确认了函数的地址，直接call 地址
    mike.BuyTicket();
00195182  lea         ecx,[mike]
00195185  call        Person::BuyTicket (01914F6h)
...
}
```

4.3 动态绑定与静态绑定

1. 静态绑定又称为前期绑定(早绑定)，**在程序编译期间确定了程序的行为，也称为静态多态**，比如：函数重载
2. 动态绑定又称后期绑定(晚绑定)，是在程序运行期间，根据具体拿到的类型确定程序的具体行为，调用具体的函数，**也称为动态多态。**
3. 本小节之前(5.2小节)买票的汇编代码很好的解释了什么是静态(编译器)绑定和动态(运行时)绑定。

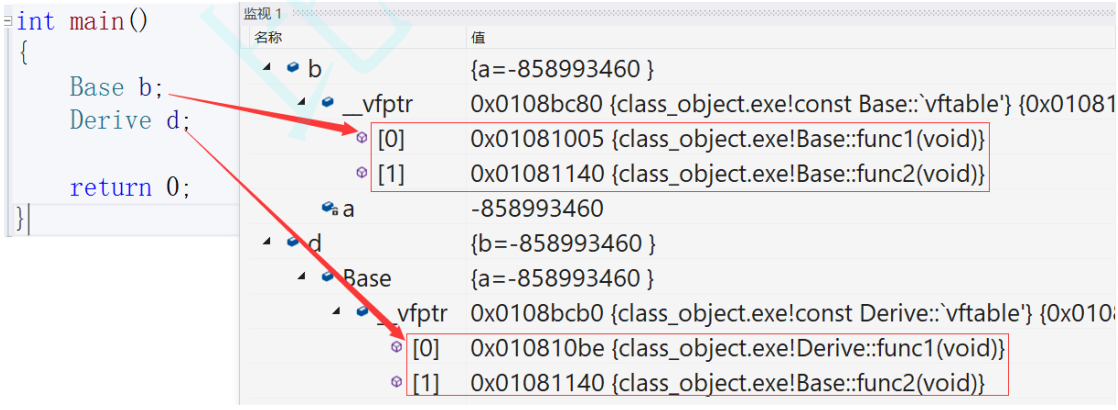
5.单继承和多继承关系的虚函数表

需要注意的是在单继承和多继承关系中，下面我们去关注的是派生类对象的虚表模型，因为基类的虚表模型前面我们已经看过了，没什么需要特别研究的

5.1 单继承中的虚函数表

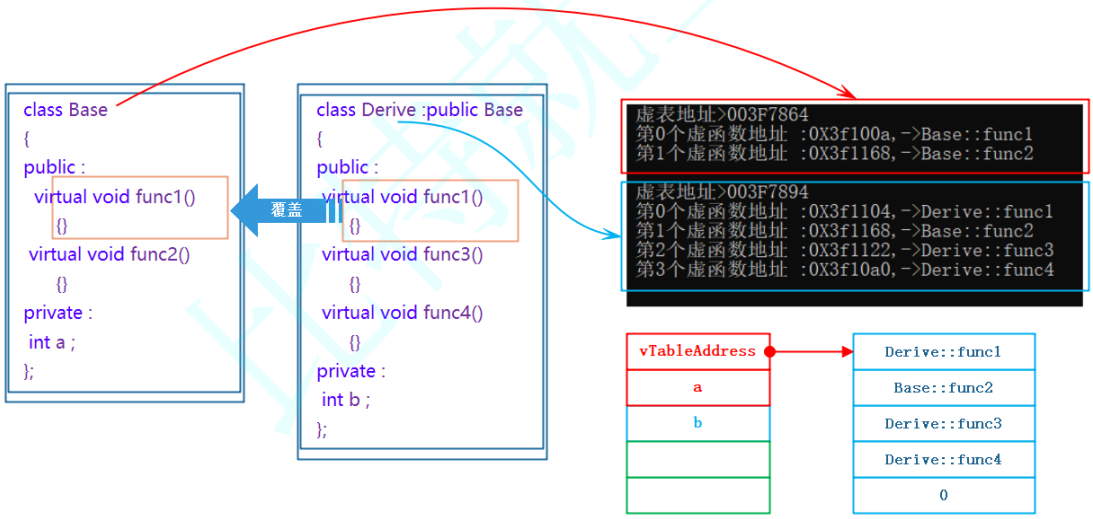
```
class Base {
public :
    virtual void func1() { cout<<"Base::func1" <<endl;}
    virtual void func2() {cout<<"Base::func2" <<endl;}
private :
    int a;
};

class Derive :public Base {
public :
    virtual void func1() {cout<<"Derive::func1" <<endl;}
    virtual void func3() {cout<<"Derive::func3" <<endl;}
    virtual void func4() {cout<<"Derive::func4" <<endl;}
private :
    int b;
};
```

观察下图中的监视窗口中我们发现看不见func3和func4。这里是编译器的监视窗口故意隐藏了这两个函数，也可以认为是他的一个小bug。那么我们如何查看d的虚表呢？下面我们使用代码打印出虚表中的函数。




```
typedef void(*VFPTR) ();
void PrintVTable(VFPTR vTable[])
{
 // 依次取虚表中的虚函数指针打印并调用。调用就可以看出存的是哪个函数
    cout << " 虚表地址>" << vTable << endl;
    for (int i = 0; vTable[i] != nullptr; ++i)
    {
        printf(" 第%d个虚函数地址 :0X%x,->", i, vTable[i]);
        VFPTR f = vTable[i];
        f();
```

```
    }
    cout << endl;
}

int main()
{
    Base b;
    Derive d;

 // 思路：取出b、d对象的头4bytes，就是虚表的指针，前面我们说了虚函数表本质是一个存虚函数指针的指针数组，这个数组最后面放了一个nullptr
 // 1.先取b的地址，强转成一个int*的指针
 // 2.再解引用取值，就取到了b对象头4bytes的值，这个值就是指向虚表的指针
 // 3.再强转成VFPTR*，因为虚表就是一个存VFPTR类型(虚函数指针类型)的数组。
 // 4.虚表指针传递给PrintVTable进行打印虚表
 // 5.需要说明的是这个打印虚表的代码经常会崩溃，因为编译器有时对虚表的处理不干净，虚表最后面没有放nullptr，导致越界，这是编译器的问题。我们只需要点目录栏的-生成-清理解决方案，再编译就好了。
    VFPTR* vTableb = (VFPTR*)(*(int*)&b);
    PrintVTable(vTableb);

    VFPTR* vTabled = (VFPTR*)(*(int*)&d);
    PrintVTable(vTabled);

    return 0;
}
```

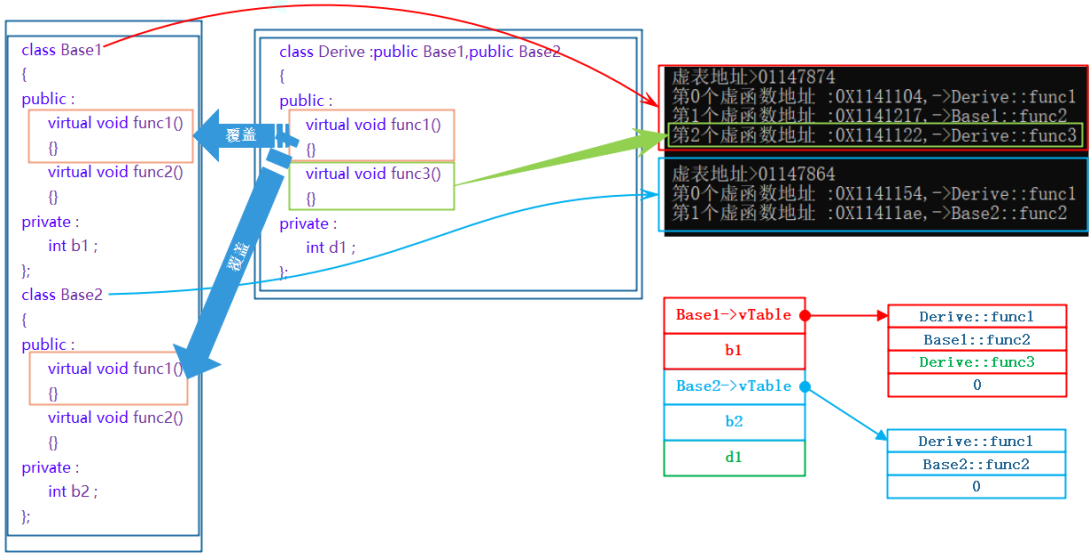


5.2 多继承中的虚函数表

```
class Base1 {
public:
    virtual void func1() {cout << "Base1::func1" << endl;}
    virtual void func2() {cout << "Base1::func2" << endl;}
private:
    int b1;
};

class Base2 {
public:
    virtual void func1() {cout << "Base2::func1" << endl;}
    virtual void func2() {cout << "Base2::func2" << endl;}
private:
```

```
    int b2;
};

class Derive : public Base1, public Base2 {
public:
    virtual void func1() {cout << "Derive::func1" << endl;}
    virtual void func3() {cout << "Derive::func3" << endl;}
private:
    int d1;
};

typedef void(*VFPTR) ();
void PrintVTable(VFPTR vTable[])
{
    cout << " 虚表地址>" << vTable << endl;
    for (int i = 0; vTable[i] != nullptr; ++i)
    {
        printf(" 第%d个虚函数地址 :0X%x,->", i, vTable[i]);
        VFPTR f = vTable[i];
        f();
    }
    cout << endl;
}

int main()
{
    Derive d;

    VFPTR* vTableb1 = (VFPTR*)(*(int*)&d);
    PrintVTable(vTableb1);

    VFPTR* vTableb2 = (VFPTR*)(*(int*)((char*)&d+sizeof(Base1)));
    PrintVTable(vTableb2);

    return 0;
}
```

观察下图可以看出：**多继承派生类的未重写的虚函数放在第一个继承基类部分的虚函数表中**



5.3. 菱形继承、菱形虚拟继承

> 实际中我们不建议设计出菱形继承及菱形虚拟继承，一方面太复杂容易出问题，另一方面这样的模型，访问基类成员有一定得性能损耗。所以菱形继承、菱形虚拟继承我们的虚表我们就不看了，一般我们也不需要研究清楚，因为实际中很少用。如果好奇心比较强的宝宝，可以去看下面的两篇链接文章。
>
> 1. C++ 虚函数表解析
> 2. C++ 对象的内存布局

6. 继承和多态常见的面试问题

6.1 概念查考

> 1. 下面哪种面向对象的方法可以让你变得富有( )
>
>   A: 继承　B: 封装　C: 多态　D: 抽象
>
> 2. ( )是面向对象程序设计语言中的一种机制。这种机制实现了方法的定义与具体的对象无关，而对方法的调用则可以关联于具体的对象。
>
>   A: 继承　B: 模板　C: 对象的自身引用　D: 动态绑定
>
> 3. 面向对象设计中的继承和组合，下面说法错误的是？（）
>
>   A：继承允许我们覆盖重写父类的实现细节，父类的实现对于子类是可见的，是一种静态复用，也称为白盒复用
>
>   B：组合的对象不需要关心各自的实现细节，之间的关系是在运行时候才确定的，是一种动态复用，也称为黑盒复用
>
>   C：优先使用继承，而不是组合，是面向对象设计的第二原则
>
>   D：继承可以使子类能自动继承父类的接口，但在设计模式中认为这是一种破坏了父类的封装性的表现
>
> 4. 以下关于纯虚函数的说法,正确的是( )
>
>   A：声明纯虚函数的类不能实例化对象　　B：声明纯虚函数的类是虚基类
>
>   C：子类必须实现基类的纯虚函数　　D：纯虚函数必须是空函数
>
> 5. 关于虚函数的描述正确的是( )
>
>   A：派生类的虚函数与基类的虚函数具有不同的参数个数和类型　B：内联函数不能是虚函数
>
>   C：派生类必须重新定义基类的虚函数　　D：虚函数可以是一个static型的函数
>
> 6. 关于虚表说法正确的是（ ）
>
>   A：一个类只能有一张虚表
>
>   B：基类中有虚函数，如果子类中没有重写基类的虚函数，此时子类与基类共用同一张虚表
>
>   C：虚表是在运行期间动态生成的
>
>   D：一个类的不同对象共享该类的虚表
>
> 7. 假设A类中有虚函数，B继承自A，B重写A中的虚函数，也没有定义任何虚函数，则（ ）
>
>   A：A类对象的前4个字节存储虚表地址，B类对象前4个字节不是虚表地址
>
>   B：A类对象和B类对象前4个字节存储的都是虚基表的地址
>
>   C：A类对象和B类对象前4个字节存储的虚表地址相同
>
>   D：A类和B类虚表中虚函数个数相同，但A类和B类使用的不是同一张虚表
>
> 8. 下面程序输出结果是什么？（）
>
>    ```
>    #include<iostream>
>    ```

```
using namespace std;
class A{
public:
    A(char *s) { cout<<s<<endl; }
    ~A(){}
};

class B:virtual public A
{
public:
    B(char *s1,char*s2):A(s1) { cout<<s2<<endl; }
};

class C:virtual public A
{
public:
    C(char *s1,char*s2):A(s1) { cout<<s2<<endl; }
};

class D:public B,public C
{
public:
    D(char *s1,char *s2,char *s3,char *s4):B(s1,s2),C(s1,s3),A(s1)
    { cout<<s4<<endl;}
};

int main() {
    D *p=new D("class A","class B","class C","class D");
    delete p;
    return 0;
}
```

A：class A class B class C class D　　B：class D class B class C class A

C：class D class C class B class A　　D：class A class C class B class D

9. 多继承中指针偏移问题？下面说法正确的是( )

```
class Base1 {  public:  int _b1; };
class Base2 {  public:  int _b2; };
class Derive : public Base1, public Base2 { public: int _d; };

int main(){
    Derive d;
    Base1* p1 = &d;
    Base2* p2 = &d;
    Derive* p3 = &d;
    return 0;
}
```

A：p1 == p2 == p3　　B：p1 < p2 < p3　　C：p1 == p3 != p2　　D：p1 != p2 != p3

10. 以下程序输出结果是什么 ()

```
class A
{
public:
    virtual void func(int val = 1){ std::cout<<"A->"<< val <<std::endl;}
```

```
    virtual void test(){ func();}
};

class B : public A
{
public:
    void func(int val=0){ std::cout<<"B->"<< val <<std::endl; }
};

int main(int argc ,char* argv[])
{
    B*p = new B;
    p->test();
    return 0;
}
```

A: A->0　B: B->1　C: A->1　D: B->0　E: 编译出错　F: 以上都不正确

参考答案：

1. A　2. D　3. C　4. A　5. B
6. D　7. D　8. A　9. C　10. B

6.2 问答题

1. 什么是多态？答：参考本节课件内容
2. 什么是重载、重写(覆盖)、重定义(隐藏)？答：参考本节课件内容
3. 多态的实现原理？答：参考本节课件内容
4. inline函数可以是虚函数吗？答：可以，不过编译器就忽略inline属性，这个函数就不再是inline，因为虚函数要放到虚表中去。
5. 静态成员可以是虚函数吗？答：不能，因为静态成员函数没有this指针，使用类型::成员函数的调用方式无法访问虚函数表，所以静态成员函数无法放进虚函数表。
6. 构造函数可以是虚函数吗？答：不能，因为对象中的虚函数表指针是在构造函数初始化列表阶段才初始化的。
7. 析构函数可以是虚函数吗？什么场景下析构函数是虚函数？答：可以，并且最好把基类的析构函数定义成虚函数。参考本节课件内容
8. 对象访问普通函数快还是虚函数更快？答：首先如果是普通对象，是一样快的。如果是指针对象或者是引用对象，则调用的普通函数快，因为构成多态，运行时调用虚函数需要到虚函数表中去查找。
9. 虚函数表是在什么阶段生成的，存在哪的？答：虚函数表是在编译阶段就生成的，一般情况下存在代码段(常量区)的。
10. C++菱形继承的问题？虚继承的原理？答：参考继承课件。注意这里不要把虚函数表和虚基表搞混了。
11. 什么是抽象类？抽象类的作用？答：参考（3.抽象类）。抽象类强制重写了虚函数，另外抽象类体现出了接口继承关系。