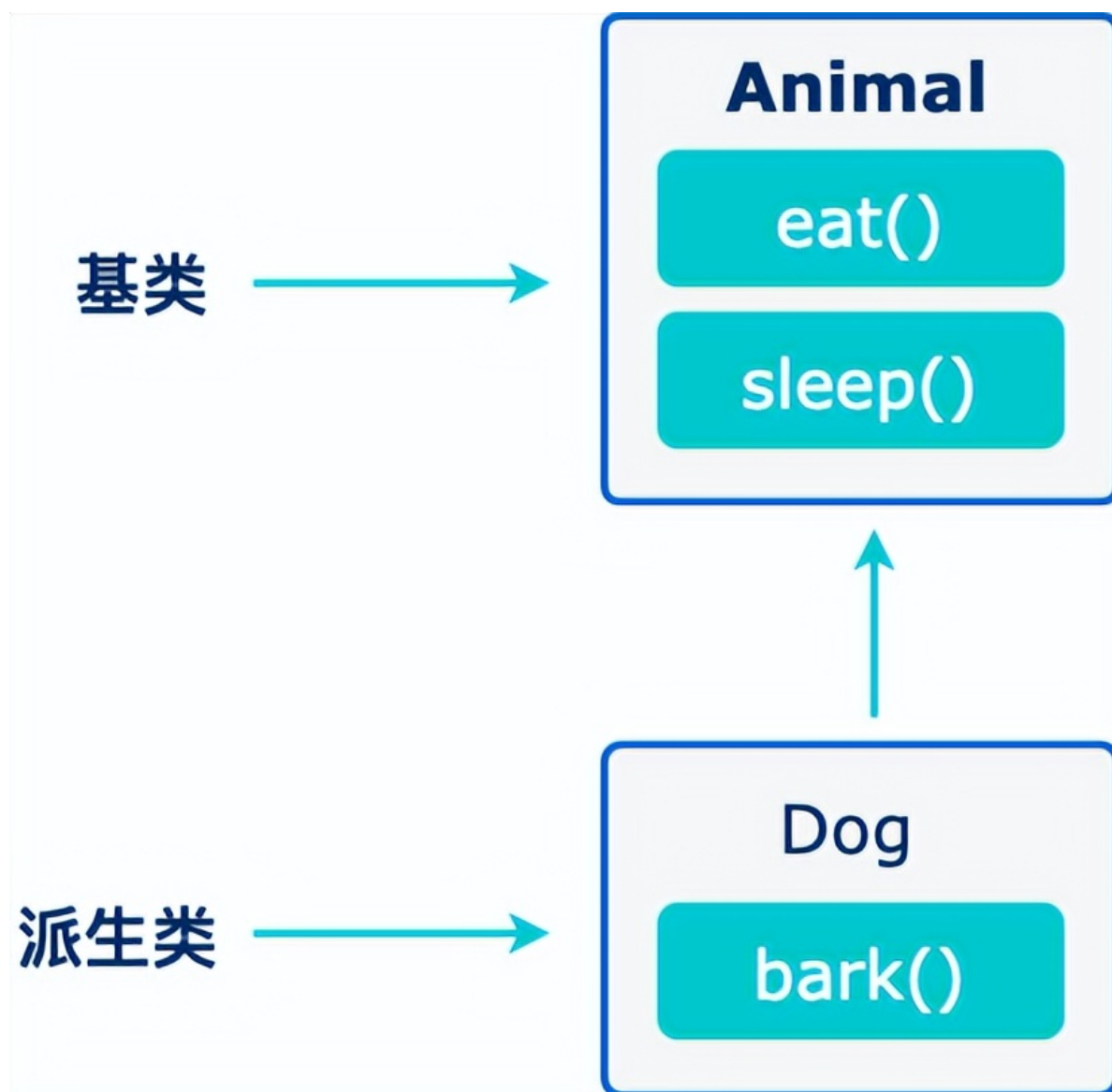


什么是继承？

在C++的类中，可以封装属性和方法，对于不同的对象，我们会设置不同的类进行描述。然而，很多时候类之间有一定关系。比如，动物类都包括eat()和sleep()方法，而狗属于动物，并且还有bark()方法。



如果狗类在动物类的基础上，直接复制过来再添加bark()方法，就会造成代码重复。因此，继承可以解决这类问题。

```
1 class Animal {
2 public:
3     void eat() {
4         cout << "吃饭" << endl;
5     }
6
7     void sleep() {
8         cout << "睡觉" << endl;
9     }
10 };
11
12 class Dog : public Animal {
13 public:
14     void bark() {
```

```
15     cout << "叫" << endl;
16 }
17 };
```

Dog类继承Animal类，Dog类中并没有eat()和sleep()方法，但是dog对象依然可以调用这两个方法。

依此类推，如果还有Cat类等，也可以继承Animal类。

访问控制

类当中的访问控制包括public、private、protected，被public修饰的属性或方法可以在类的外部直接访问，被private、protected修饰的属性或方法可以在类的外部不可以直接访问。一般成员变量都设置为private，操作接口设置为public暴露出来，供类的外部操作。

继承时一般采用public方式，子类可以直接继承父类的控制权限。如果采用protected继承会将父类的public也变为protected，如果采用private继承会将父类所有权限都变为private。

protected可以被子类继承，private不可以被子类继承。

如果子类的方法和父类方法相同，会重定义父类方法，子类对象调用时采用就近原则，只有子类没有该方法，才回到父类查找并调用。

```
1  class Animal {
2  private:
3
4  protected:
5      string m_name;
6  public:
7
8      Animal() {
9          m_name = "动物类";
10         cout << "父类构造函数" << endl;
11     }
12
13     ~Animal() {
14         cout << "父类析构函数" << endl;
15     }
16     void eat() {
17         cout << "吃饭" << endl;
18     }
19
20     void sleep() {
21         cout << "睡觉" << endl;
22     }
23
24     void output() {
25         cout << "父类: " << m_name << endl;
26     }
27 };
28
29 class Dog : public Animal {
30 public:
31
32     Dog() {
33         cout << "子类构造函数" << endl;
34     }
35
36     ~Dog() {
```

```

37     cout << "子类析构函数" << endl;
38 }
39
40 void bark() {
41     cout << "叫" << endl;
42 }
43
44 /*void output() {
45     cout << "子类: " << m_name << endl;
46 }*/
47 };
48
49 int main() {
50
51     {
52         Dog dog;
53         dog.eat();
54         dog.sleep();
55         dog.bark();
56         dog.output();
57     }
58
59     return 0;
60 }

```

父类构造函数
 子类构造函数
 吃饭
 睡觉
 叫
 父类：动物类
 子类析构函数
 父类析构函数

先调用父类构造函数，先析构子类析构函数。

子类调用父类构造函数

子类的成员变量大多数和父类相同，可以直接继承，然而，子类中可能还会添加新的成员变量，那么在进行构造函数初始化的时候，就要将父类的初始化过程再写一遍，这是很不方便的，因此，可以将父类构造函数直接在子类中引用，重复部分就不需要再实现一次。

公众号：黑猫编程

网址：<https://noi.hiqier.co>

```

1  class Animal {
2  private:
3
4  protected:
5      string m_name;
6      int m_age;
7  public:
8      Animal(string name, int age) {
9          m_name = name;
10         m_age = age;
11     }
12 };
13
14 class Dog : public Animal {
15 private:
16     char m_gender;
17 public:
18
19     Dog() {
20         cout << "子类构造函数" << endl;
21     }
22
23     Dog(string name, int age, char gender): Animal(name, age) {
24         m_gender = gender;
25     }
26
27     void output() {
28         cout << m_name << " " << m_age << " " << m_gender << endl;
29     }
30 };
31
32 Dog dog = Dog("旺财", 6, 'f');

```

多态

C++在默认情况下，编译器会根据指针类型调用对应函数，不存在多态。如果要实现多态，需要使用virtual关键字。

多态就是同一操作作用于不同对象，会产生不同结果，比如创建父类指针，new不同子类：Animal* dog = new Dog();根据子类不同，父类指针调用同一个方法结果也不同。

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Animal {
6
7  public:
8
9      void eat() {
10         cout << "动物吃饭" << endl;
11     }
12
13     void sleep() {
14         cout << "动物睡觉" << endl;
15     }
16 };

```

```

17
18 class Dog : public Animal {
19 public:
20
21     void eat() {
22         cout << "狗吃饭" << endl;
23     }
24
25     void sleep() {
26         cout << "狗睡觉" << endl;
27     }
28 };
29
30 int main() {
31
32     Animal* dog = new Dog();
33
34     dog->eat();
35     dog->sleep();
36
37     return 0;
38 }

```

```

dog->eat();
➡ 00007FF77B1425DC  mov     rcx,qword ptr [dog]
00007FF77B1425E0  call    Animal::eat (07FF77B141514h)
dog->sleep();
00007FF77B1425E5  mov     rcx,qword ptr [dog]
00007FF77B1425E9  call    Animal::sleep (07FF77B14150Fh)

```

正常情况下，结果还是调用父类方法，编译器根据父类指针判断对象，而不是new后面的子类。反汇编分析发现call调用的Animal里面的方法。

下面，给父类加上virtual，子类也可以加上virtual，也可以省略。**子类不需要强制重写父类方法。**

```

1 virtual void eat() {
2     cout << "动物吃饭" << endl;
3 }
4
5 virtual void sleep() {
6     cout << "动物睡觉" << endl;
7 }

```

```

    dog->eat();
    00007FF6B37020F1  mov             rax,qword ptr [dog]
    00007FF6B37020F5  mov             rax,qword ptr [rax]
    00007FF6B37020F8  mov             rcx,qword ptr [dog]
    00007FF6B37020FC  call            qword ptr [rax]
    dog->sleep();
    00007FF6B37020FE  mov             rax,qword ptr [dog]
    00007FF6B3702102  mov             rax,qword ptr [rax]
    00007FF6B3702105  mov             rcx,qword ptr [dog]
    00007FF6B3702109  call            qword ptr [rax+8]

```

反汇编分析，call后面的地址是寄存器中的值，是一个可变的值。

虚表原理

虚函数的实现原理是虚表，只要加上virtual关键字，就会有虚表，存储着最终需要调用的虚函数地址。

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Animal {
6
7  public:
8      int m_age1;
9      virtual void eat() {
10         cout << "动物吃饭" << endl;
11     }
12
13     virtual void sleep() {
14         cout << "动物睡觉" << endl;
15     }
16 };
17
18 class Dog : public Animal {
19 public:
20     int m_age2;
21     void eat() {
22         cout << "狗吃饭" << endl;
23     }
24
25     void sleep() {
26         cout << "狗睡觉" << endl;
27     }
28 };
29
30 int main() {
31
32     Animal* dog = new Dog();
33     dog->m_age1 = 0x1234abcd;
34     dog->eat();
35     dog->sleep();

```

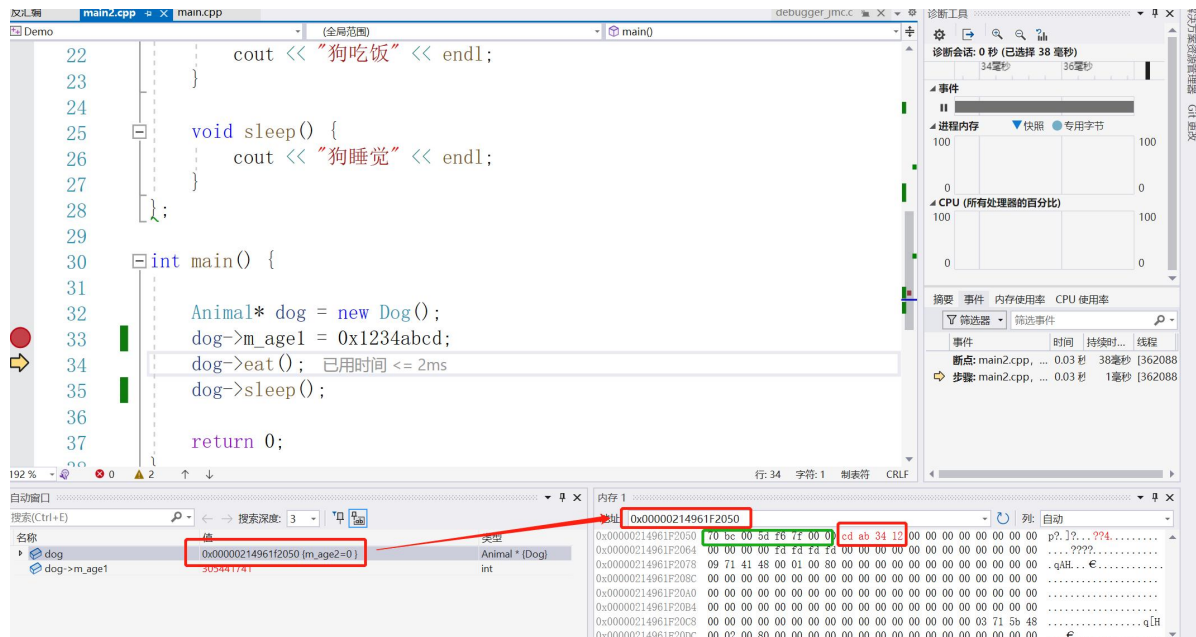


```

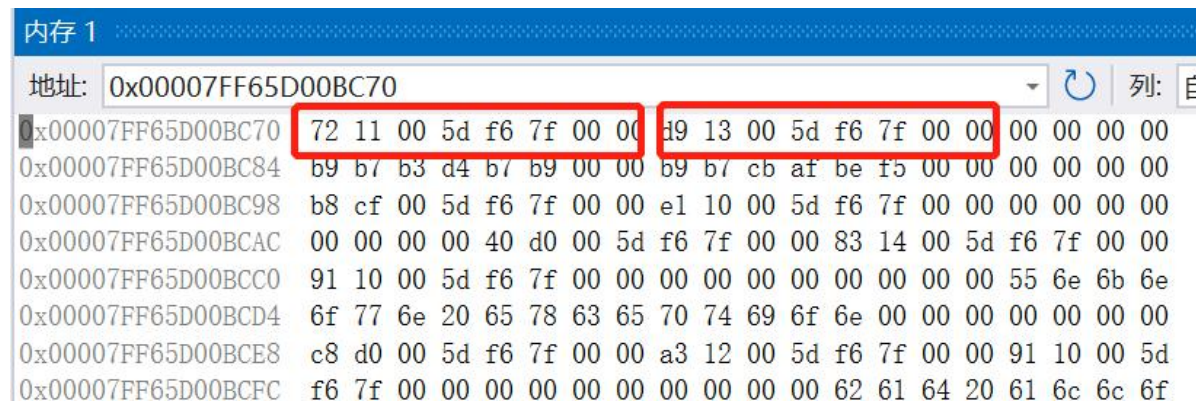
36
37     return 0;
38 }

```

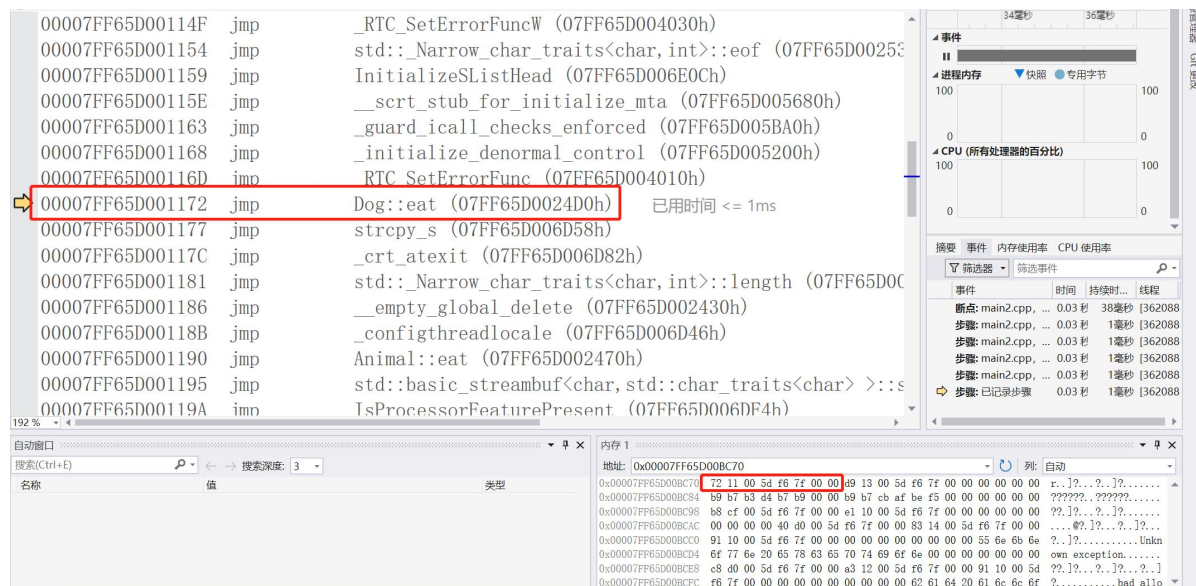
通过dog地址，在内存中查找，找到m_age1的位置，前面的8个字节就是指向虚表的地址，由于本项目是64位，所以指针占8个字节。



在内存中继续查找虚表地址内容，记录下如下两个地址，即我们最终的虚函数地址：



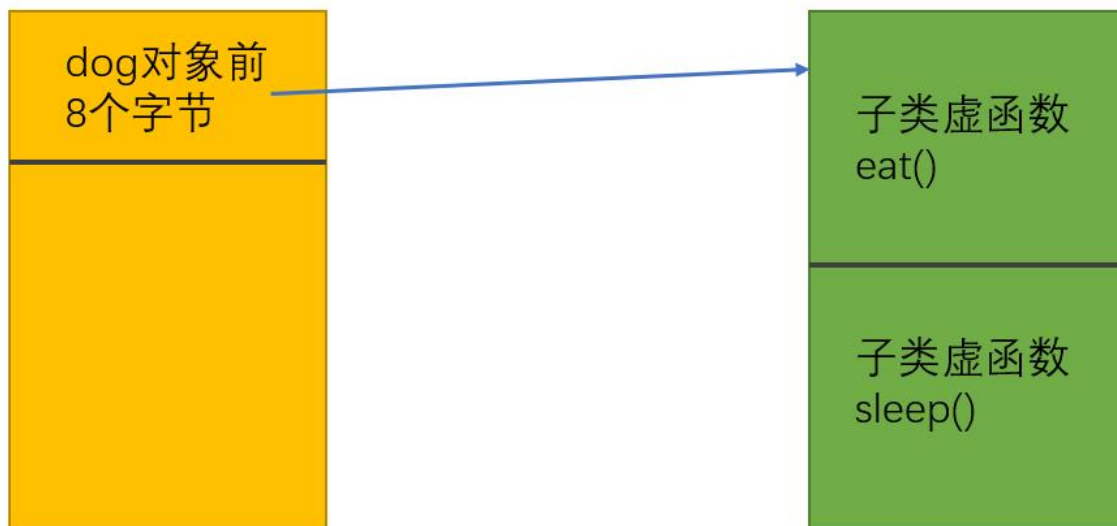
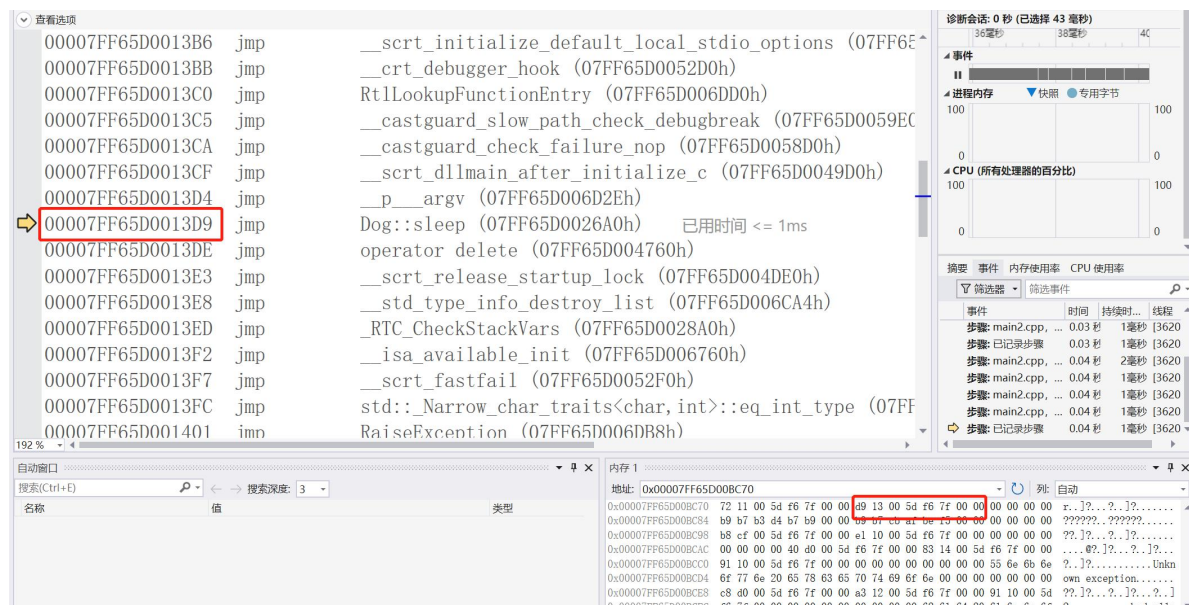
F11向下执行过程中，执行过call语句：



即第一个地址已经对应，大家自己查找下一个地址是否匹配。

跳出后，继续F11：

```
00007FF65D002785  mov          dword ptr [rax+8], 1234ABCDh
                  dog->eat();
00007FF65D00278C  mov          rax, qword ptr [dog]
00007FF65D002790  mov          rax, qword ptr [rax]
00007FF65D002793  mov          rcx, qword ptr [dog]
00007FF65D002797  call         qword ptr [rax]
                  dog->sleep();
00007FF65D002799  mov          rax, qword ptr [dog]
00007FF65D00279D  mov          rax, qword ptr [rax]
00007FF65D0027A0  mov          rcx, qword ptr [dog]
00007FF65D0027A4  call         qword ptr [rax+8]  已用时间 <= 1ms
```



虚析构造函数

如果存在父类指针指向子类对象情况，一般将析构造函数声明为虚析构造函数，使子类析构造函数也被调用，保证析构完整性。


```

5      class Animal {
6
7      public:
8          int m_age1;
9          virtual void eat() {
10             cout << "动物吃饭" << endl;
11         }
12
13         virtual void sleep() {
14             cout << "动物睡觉" << endl;
15         }
16
17         virtual ~Animal() {
18             cout << "父类析构" << endl;
19         }
20     };
21
22     class Dog : public Animal {
23     public:
24         int m_age2;
25         void eat() {
26             cout << "狗吃饭" << endl;
27         }
28
29         void sleep() {
30             cout << "狗睡觉" << endl;
31         }
32
33         ~Dog() {
34             cout << "子类析构" << endl;
35         }
36     };

```

纯虚函数：父类没有函数体且初始化为0的虚函数，一般用作定义接口规范，子类必须重写父类虚函数。

如果没有完全重写，子类也是抽象类，不可以实例化对象。

```
5      class Animal {
6      |
7      |     public:
8      |         virtual void eat() = 0;
9      |         virtual void sleep() = 0;
10     |     };
    
```

多继承

多继承是一个子类同时继承多个父类，具有多个父类的属性和方法。然而多继承会导致函数变量重名等问题，导致类的设计非常复杂，谨慎使用。

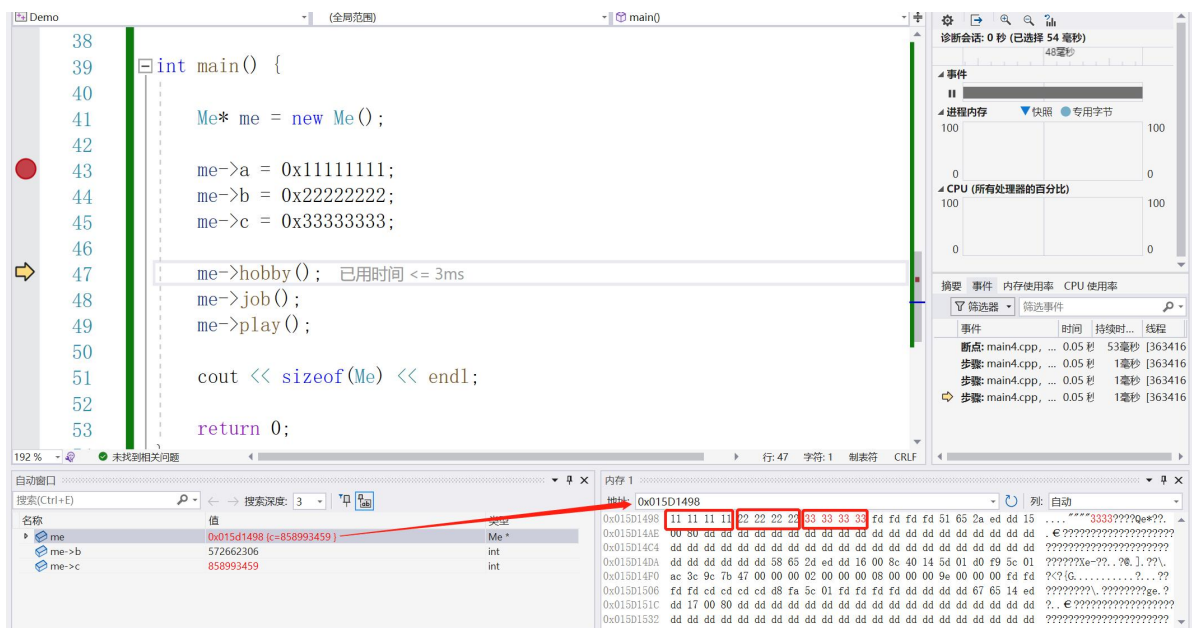
```
1  class Father{
2  public:
3      int a;
4
5      virtual void hobby() {
6          cout << "Father()::hobby()" << endl;
7      }
8  };
9
10 class Mother {
11 public:
12     int b;
13
14     virtual void job() {
15         cout << "Mother()::job()" << endl;
16     }
17 };
18
19 class Me : public Father, public Mother {
20 public:
21     virtual void hobby() {
22         cout << "Me()::hobby()" << endl;
23     }
24
25     virtual void job() {
26         cout << "Me()::job()" << endl;
27     }
28
29     virtual void play() {
30         cout << "Me()::play()" << endl;
31     }
32 };
33
34 int main() {
    
```

```

35
36     Me* me = new Me();
37
38     me->a = 0x11111111;
39     me->b = 0x22222222;
40
41     me->hobby();
42     me->job();
43     me->play();
44
45     cout << sizeof(Me) << endl;
46
47     return 0;
48 }

```

Me类继承了Father类和Mother类，那么在Me类对象中，就包括三个成员变量，按照继承的父类顺序分别为a, b, 最后是Me类中的c。



```

004B2628  mov          eax,dword ptr [me]
004B262B  mov          dword ptr [eax+4],2222222h
           me->c = 0x33333333;
004B2632  mov          eax,dword ptr [me]
004B2635  mov          dword ptr [eax+8],33333333h

           me->hobby();
➡ 004B263C  mov          ecx,dword ptr [me]
           004B263F  call         Father::hobby (04B1505h)
           me->job();
           004B2644  mov          ecx,dword ptr [me]
           004B2647  add          ecx,4
           004B264A  call         Mother::job (04B14FBh)
           me->play();
           004B264F  mov          ecx,dword ptr [me]
           004B2652  call         Me::play (04B1500h)
           cout << sizeof(Me) << endl;

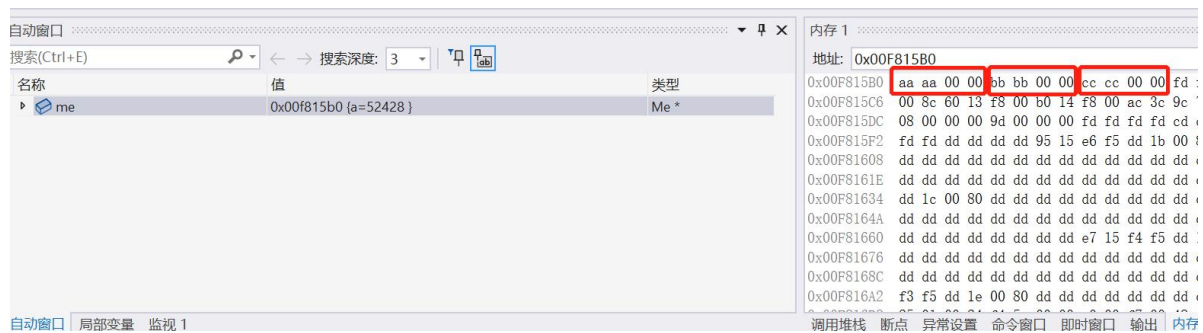
```

同名成员变量

```

1  class Father {
2  public:
3      int a = 10;
4  };
5
6  class Mother {
7  public:
8      int a = 30;
9  };
10
11 class Me : public Father, public Mother {
12 public:
13     int a = 30;
14 };
15
16 int main() {
17
18     Me* me = new Me();
19
20     cout << me->Father::a << endl;
21     cout << me->Mother::a << endl;
22     cout << me->Me::a << endl;
23     cout << me->a << endl;
24
25     return 0;
26 }

```



同名成员变量调用，可以指定类名作为作用域，对同名变量进行区分。

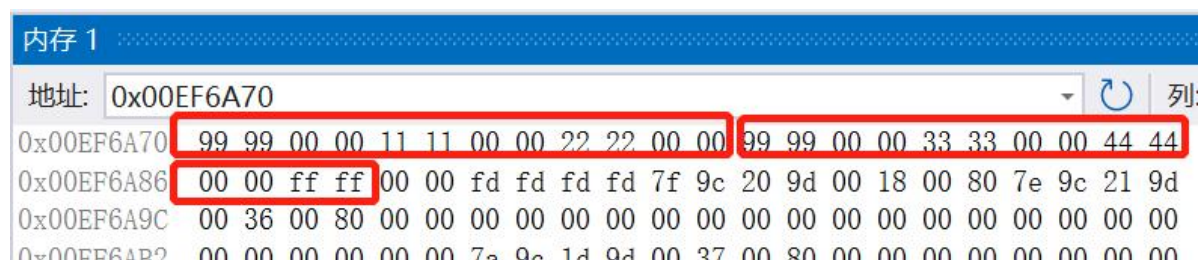
菱形继承

两个派生类继承同一个基类，而又有某个类同时继承这两个派生类，这种继承被叫做菱形继承，是多继承当中更加复杂的一种情况。

```

1  class People {
2  public:
3      int x = 0x9999;
4  };
5
6  class Father: public People {
7  public:
8      int a = 0x1111;
9      int c = 0x2222;
10 };
11
12 class Mother : public People {
13 public:
14     int a = 0x3333;
15     int d = 0x4444;
16 };
17
18 class Me : public Father, public Mother {
19 public:
20     int e = 0xffff;
21 };
22
23 int main() {
24
25     Me* me = new Me();
26
27     cout << sizeof(Me) << endl;
28
29     return 0;
30 }

```



根据内存分布可知，Father和Mother同时继承了People，都具有x属性，Me继承Father和Mother，就会有二个x属性，如果People中属性足够多，那么Me中的属性就会出现大量冗余。

虚继承和虚表指针

在继承父类前加上virtual关键字，C++编译器就会将父类中重复部分设置为一个共享空间，内存中只有一份数据，但是要多出两个虚表指针。

```
9      class Father : virtual public People {
10      public:
11          int a = 0x1111;
12          int c = 0x2222;
13      };
14
15      class Mother : virtual public People {
16      public:
17          int a = 0x3333;
18          int d = 0x4444;
19      };
```

```
1  class People {
2  public:
3      int x = 0x9999;
4  };
5
6  class Father : virtual public People {
7  public:
8      int a = 0x1111;
9  };
10
11 class Mother : virtual public People {
12 public:
13     int b = 0x2222;
14 };
15
16 class Me : public Father, public Mother {
17 public:
18     int c = 0xefef;
19 };
20
21 int main() {
22
23     Me* me = new Me();
24
25     cout << sizeof(Me) << endl;
26
27     return 0;
28 }
```

内存 1																
地址: 0x008969E0																
0x008969E0	48	7b	a6	00	11	11	00	00	54	7b	a6	00	22	22	00	00
0x008969F6	00	00	fd	fd	fd	fd	00	31	00	80	2d	b1	b4	49	00	16
0x00896A0C	00	00	00	00	00	00	00	00	00	00	00	00	2e	b1	b3	49
0x00896A22	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00896A38	2a	b1	af	49	00	33	00	80	00	00	00	00	00	00	24	b1
0x00896A4E	00	80	00	00	00	00	00	00	00	00	26	b1	ab	49	00	34
0x00896A64	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00896A7A	a7	49	00	35	00	80	00	00	00	00	00	00	00	00	00	00
0x00896A90	3f	b1	a2	49	00	18	00	8d	80	ab	89	00	90	ed	89	00
0x00896AA6	00	00	02	00	00	00	17	00	00	00	4b	00	00	fd	fd	fd
0x00896ABC	55	45	44	49	54	49	4f	4e	3d	43	6f	6d	6d	75	6e	69
0x00896AD2	fd	00	00	00	00	00	36	b1	9b	49	00	19	00	80	00	00

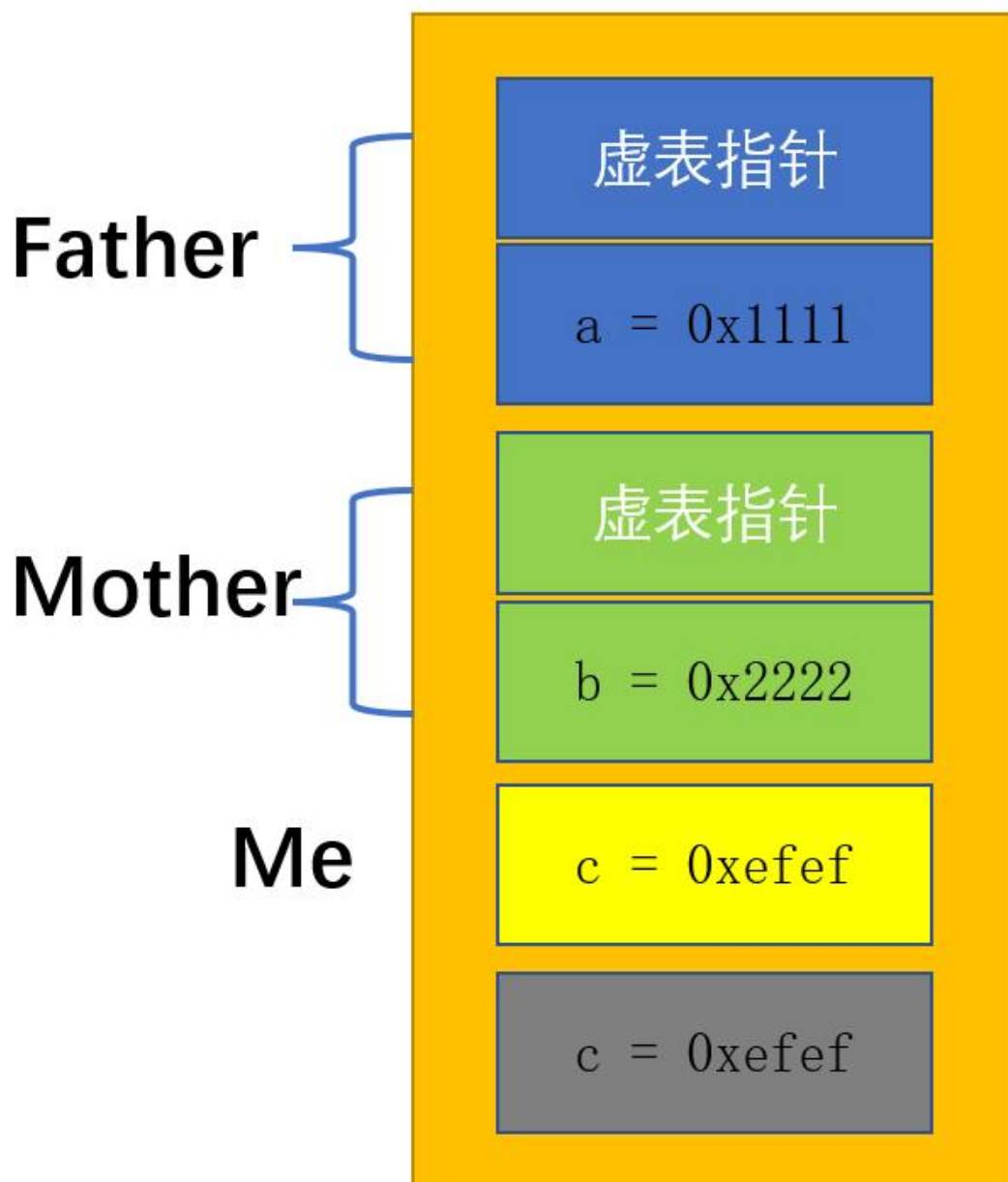
地址: 0x00A67B48																
0x00A67B48	00	00	00	00	14	00	00	00	00	00	00	00	00	00	00	00

虚表指针保存两个值，第一个是虚表指针与当前类的偏移量（一般为0），第二个是虚基类第一个成员变量与本类的起始偏移量。

如图，第一个虚表指针偏移为20（16进制表示为14）。

内存 1																
地址: 0x00A67B54																
0x00A67B54	00	00	00	00	0c	00	00	00	00	00	00	cc	8a	a6	00	ca
0x00A67B6A	a6	00	00	00	00	00	55	6e	6b	6e	6f	77	6e	20	65	78
0x00A67B80	6e	00	00	00	00	00	00	00	24	8b	a6	00	ff	10	a6	00
0x00A67B96	00	00	62	61	64	20	61	6c	6c	6f	63	61	74	69	6f	6e
0x00A67BAC	80	8b	a6	00	a5	10	a6	00	27	11	a6	00	00	00	00	00
0x00A67BC2	72	61	79	20	6e	65	77	20	6c	65	6e	67	74	68	00	00
0x00A67BD8	98	7c	a6	00	a8	7d	a6	00	00	7f	a6	00	24	7f	a6	00
0x00A67BEE	a6	00	01	00	00	00	00	00	00	00	01	00	00	00	01	00
0x00A67C04	01	00	00	00	53	74	61	63	6b	20	61	72	6f	75	6e	64
0x00A67C1A	61	72	69	61	62	6c	65	20	27	00	27	20	77	61	73	20
0x00A67C30	74	65	64	2e	00	00	00	00	54	68	65	20	76	61	72	69
0x00A67C46	00	00	27	20	69	73	20	62	65	69	6e	67	20	75	73	65

第二个虚表指针偏移为12（16进制表示为0c）。



但是，要注意这里的虚表指针是派生类继承基类时的指针，如果在派生类内部继续添加虚函数，还会有一个虚表指针。

当前Me类大小为24Byte，如果再加一个虚函数会变成28Byte，然而继续加虚函数虚表指针只有一个，不会增加。