

## 三、C++学习笔记—核心编程

本阶段，将对C++面向对象编程技术做详细学习，深入C++中的核心和精髓

### 3.4.6.3 继承中的对象模型

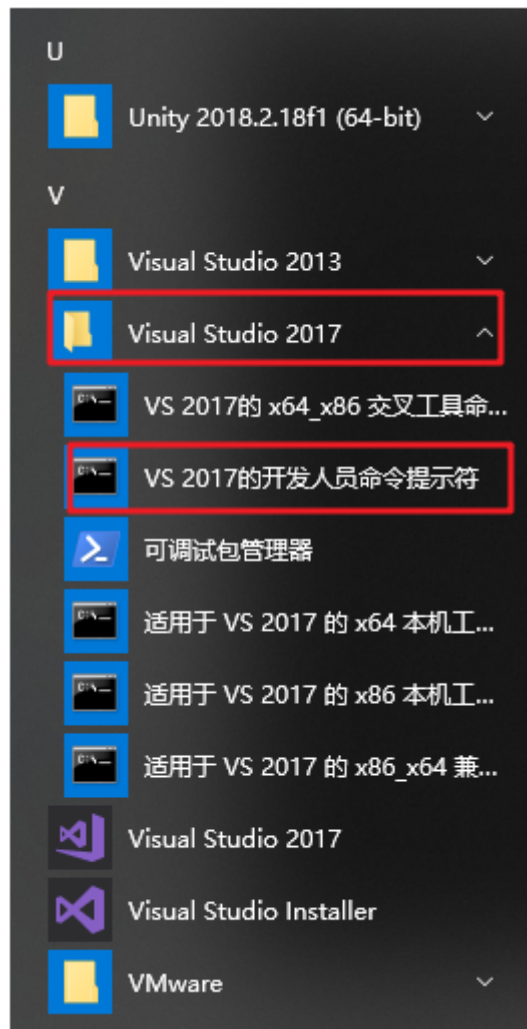
**问题：**从父类继承过来的成员，哪些属于子类对象中？

**示例：**

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //继承中的对象模型
6  class Base
7  {
8  public:
9      int m_A;
10 protected:
11     int m_B;
12 private:
13     int m_C; //私有成员只是被隐藏了，但是还是会继承下去
14
15 };
16
17 //公共继承
18 class Son : public Base
19 {
20
21 public:
22     int m_D;
23
24 private:
25
26 };
27
28 //利用开发人员命令提示工具查看对象模型
29 //在文件目录下：
30 //查看命令：cl /d1 reportSingleClassLayout类名 文件名
31
32
33 void test01()
34 {
35     //在父类中所有非静态成员属性都会被子类继承下去
36     //父类中私有成员属性 是被编译器给隐藏了，因此访问不到，但是确实被继承下去了。
37     cout << "size of Son=" << sizeof(Son) << endl; //16
38 }
```

```
39  
40  
41 int main()  
42 {  
43     test01();  
44     system("pause");  
45     return 0;  
46 }
```

利用工具查看：



打开工具窗口后，定位到当前CPP文件的盘符

然后输入：cl /d1 reportSingleClassLayout查看的类名 所属文件名

效果如下图：

```
F:\VS2017\VS2017_workstation\C++核心编程\C++核心编程>cl /d1 reportSingleClassLayoutSon "C++核心编程.cpp"
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.16.27027.1 版
版权所有 (C) Microsoft Corporation。保留所有权利。

C++核心编程.cpp
F:\VS2017\install\VC\Tools\MSVC\14.16.27023\include\xlocale(319): warning C4530: 使用了 C++ 异常处理程序, 但未启用展开语
义。请指定 /EHsc

class Son      size(16):
0      +--- (base class Base)
0      |   m_A
4      |   m_B
8      |   m_C
12     |   +---
12     |   m_D
12     +---

Microsoft (R) Incremental Linker Version 14.16.27027.1
Copyright (C) Microsoft Corporation. All rights reserved.

/out:C++核心编程.exe
C++核心编程.obj
F:\VS2017\VS2017_workstation\C++核心编程\C++核心编程\
```

结论：父类中私有成员也是被子类继承下去了，只是由编译器给隐藏后访问不到

### 3.4.6.4 继承中构造和析构顺序

子类继承父类后，当创建子类对象，也会调用父类的构造函数

问题：父类和子类的构造和析构顺序是谁先谁后？

示例：

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //继承中的构造和析构顺序
6  class Base
7  {
8  public:
9      Base()
10     {
11         cout << "Base构造函数" << endl;
12     }
13
14     ~Base()
15     {
16         cout << "Base析构函数" << endl;
17     }
18 };
19
20 //子类（派生类）
21 class Son : public Base
22 {
23 public:
24     Son()
25     {
26         cout << "Son构造函数" << endl;
27     }
```

```

28     }
29     ~Son()
30     {
31         cout << "Son析构函数" << endl;
32     }
33 };
34
35 void test01()
36 {
37     //Base b;
38     //继承中的构造和析构顺序如下:
39     //先构造父类, 再构造子类, 析构的顺序与构造的顺序相反
40     Son s;
41 }
42
43
44 int main()
45 {
46     test01();
47     system("pause");
48     return 0;
49 }

```

```

F:\VS2017\VS2017_workstation\C++核心编程\C++核心编程>a.exe
Base构造函数
Son构造函数
Son析构函数
Base析构函数
请按任意键继续. . .

```

总结: 继承中 先调用父类构造函数, 再调用子类构造函数, 析构顺序与构造相反

### 3.4.6.5 继承同名成员处理方式

**问题:** 当子类与父类出现同名的成员, 如何通过子类对象, 访问到子类或父类中同名的数据呢?

- 访问子类同名成员 直接访问即可
- 访问父类同名成员 需要加作用域

**示例:**

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //继承中同名成员处理
6  class Base
7  {
8  public:
9      Base()
10     {

```

```

11         m_A = 100;
12     }
13
14     void func()
15     {
16         cout << "Base - func () 调用" << endl;
17     }
18
19     void func(int a)
20     {
21         cout << "Base - func (int a) 调用" << endl;
22     }
23
24
25     int m_A;
26
27 };
28
29
30 //派生类
31 class Son : public Base
32 {
33
34 public:
35     Son()
36     {
37         m_A = 200;
38     }
39
40     void func()
41     {
42         cout << "Son - func () 调用" << endl;
43     }
44
45     int m_A;
46 };
47
48
49 //同名成员属性处理方式
50 void test01()
51 {
52     Son s;
53     cout << "Son 下 m_A=" << s.m_A << endl;
54     //如果通过子类对象 访问到父类中同名成员，需要加作用域
55     cout << "Base 下 m_A=" << s.Base::m_A << endl;
56
57 }
58 //同名成员函数处理
59 void test02()
60 {
61     Son s; //子类对象
62     s.func(); //直接调用 调用的是子类中的同名成员
63     s.Base::func(); //调用父类中的同名成员，需要加作用域

```

```

64
65     //如果子类中出现了和父类同名的成员函数，子类的同名成员会隐藏掉父类中所有同名成员函数
66     //如果想访问到父类中被隐藏的同名成员函数，需要加作用域
67     s.Base::func(300);
68
69 }
70
71 int main()
72 {
73     //test01();
74     test02();
75     system("pause");
76     return 0;
77 }

```

#### 总结：

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

### 3.4.6.6 继承同名静态成员处理方式

问题：继承中同名的静态成员在子类对象上如何进行访问？

静态成员和非静态成员出现同名，处理方式一致

- 访问子类同名成员 直接访问即可
- 访问父类同名成员 需要加作用域

#### 示例：

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //继承中的同名静态成员处理方式
6  class Base
7  {
8  public:
9
10     static int m_A;
11
12     static void func()
13     {
14         cout << "Base - static void func()" << endl;
15     }
16
17 };
18
19 int Base::m_A = 100; //类外初始化
20
21 class Son : public Base

```

```

22 {
23 public:
24     static int m_A;
25
26     static void func()
27     {
28         cout << "Son - static void func()" << endl;
29     }
30
31 };
32
33 int Son::m_A=200;
34
35
36 //同名静态成员属性
37 void test01()
38 {
39     //1、通过对象访问
40     cout << "通过对象访问: " << endl;
41     Son s;
42     cout << "SON m_A =" << s.m_A << endl;
43     cout << "Base m_A =" << s.Base::m_A << endl;
44
45     //2、通过类名访问
46     cout << "通过类名访问: " << endl;
47     cout << "Son 下m_A:" << Son::m_A << endl;
48     //第一个::通过类名访问方式 第二个::代表访问父类作用域下
49     cout << "Base 下m_A:" << Son::Base::m_A << endl;
50
51 }
52
53
54 //同名静态成员函数
55 void test02()
56 {
57     //1、通过对象访问
58     cout << "通过对象访问" << endl;
59     Son s;
60     s.func();
61     s.Base::func();
62
63     //2、通过类名访问
64     cout << "通过类名访问" << endl;
65     Son::func();
66
67     //子类出现和父类同名静态成员函数，也会隐藏父类中所有同名成员函数
68     //如果想访问父类中被隐藏同名成员，需要加作用域
69     Son::Base::func();
70 }
71
72 int main()
73 {
74     //test01();

```

```

75     test02();
76     system("pause");
77     return 0;
78 }

```

总结：同名静态成员处理方式和非静态处理方式一样，只不过有两种访问的方式（通过对象 和 通过类名）

### 3.4.6.7 多继承语法

C++允许一个类继承多个类

语法：class 子类：继承方式 父类1，继承方式 父类2...

多继承可能会引发父类中有同名成员出现，需要加作用域区分

**C++实际开发中不建议用多继承**

示例：

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //多继承语法
6  class Base1
7  {
8  public:
9      Base1()
10     {
11         m_A = 100;
12     }
13     int m_A;
14 };
15
16 class Base2
17 {
18 public:
19     Base2()
20     {
21         m_A = 200;
22     }
23     int m_A;
24
25 };
26
27 //子类 需要继承Base1 和 Base2
28 //语法: class子类: 继承方式 父类1, 继承方式 父类2...
29 class Son:public Base1,public Base2
30 {
31 public:
32     Son()
33     {
34         m_C = 300;

```



```

35         m_D = 400;
36     }
37
38     int m_C;
39     int m_D;
40 };
41
42 void test01()
43 {
44     Son s;
45     cout << "sizeof SON=" << sizeof(s) << endl;
46
47     //当父类中出现同名的成员，需要加作用域区分
48     cout << "Base1=m_A=" << s.Base1::m_A << endl;
49     cout << "Base2=m_A=" << s.Base2::m_A << endl;
50
51 }
52
53
54 int main()
55 {
56     test01();
57     //test02();
58     system("pause");
59     return 0;
60 }

```

总结：多继承中如果父类中出现了同名情况，子类使用时候要加作用域

在实际开发过程中，不建议使用多继承

### 3.4.6.8 菱形继承

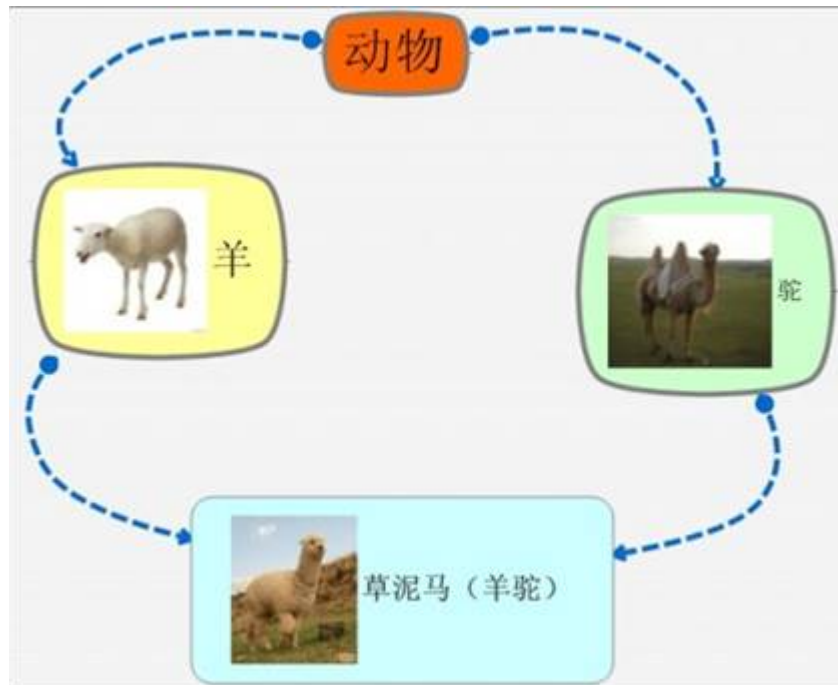
**菱形继承概念：**

两个派生类继承同一个基类

又有某个类同时继承者两个派生类

这种继承被称为菱形继承，或者钻石继承

**典型的菱形继承案例：**



#### 菱形继承问题：

1. 羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性。
2. 草泥马继承动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

#### 示例：

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //菱形继承
6  //动物类
7  class Animal
8  {
9  public:
10     int m_Age;
11 };
12
13 //利用虚继承，解决菱形继承的问题
14 //继承之前，加上关键字virtual 变为虚继承
15 //Animal类称为 虚基类
16
17
18 //羊类
19 class Sheep :virtual public Animal {};
20 //驼类
21 class Tuo : virtual public Animal {};
22 //羊驼类
23 class SheepTuo : public Sheep, public Tuo {};
24
25
26 void test01()
27 {
```

```

28     SheepTuo st;
29     st.Sheep::m_Age = 18;
30     st.Tuo::m_Age = 28;
31
32     //当菱形继承，两个父类拥有相同数据，需要加以作用域区分
33     cout << "st.Sheep::m_Age=" << st.Sheep::m_Age << endl;
34     cout << "st.Tuo::m_Age=" << st.Tuo::m_Age << endl;
35     cout << "st.m_A=" << st.m_Age << endl;
36
37     //这份数据我们知道，只要有一份就可以，菱形继承导致数据有两份，资源浪费
38
39 }
40 int main()
41 {
42     test01();
43     //test02();
44     system("pause");
45     return 0;
46 }

```

#### 总结：

- 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

## 3.4.7 多态

### 3.4.7.1 多态的基本概念

多态是C++面向对象三大特性之一

多态分为两类：

- 静态多态: 函数重载 和 运算符重载属于静态多态，复用函数名
- 动态多态: 派生类和虚函数实现运行时多态

静态多态和动态多态区别：

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

下面通过案例进行讲解多态

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //多态
6  //动物类
7  class Animal
8  {
9  public:
10     //虚函数

```

```

11     virtual void speak()
12     {
13         cout << "动物在说话" << endl;
14     }
15 };
16
17 //派生类--猫类
18 class Cat :public Animal
19 {
20 public:
21     //重写 函数返回类型 函数名 参数列表 完全相同
22     void speak()
23     {
24         cout << "小猫在说话" << endl;
25     }
26 };
27 //派生类--狗类
28 class Dog : public Animal
29 {
30 public:
31     void speak()
32     {
33         cout << "小狗在说话" << endl;
34     }
35 };
36
37
38 //执行说话的函数
39 //地址早绑定 在编译阶段就确定了函数的地址
40 //如果想执行让猫说话，那么这个函数地址就不能提前绑定，需要在运行阶段进行绑定，地址晚绑定
41
42 //动态多态满足条件
43 //1、有继承关系
44 //2、子类重写父类的虚函数
45
46 //动态多态使用
47 //1、父类的指针或者引用 指向子类对象
48
49 void doSpeak(Animal &animal) //Animal & animal = cat;
50 {
51     animal.speak();
52 }
53
54 //测试函数
55 void test01()
56 {
57     Cat cat;
58     doSpeak(cat);
59
60     Dog dog;
61     doSpeak(dog);
62
63 }

```

```

64
65
66 int main()
67 {
68     test01();
69     //test02();
70     system("pause");
71     return 0;
72 }

```

**总结：**

**多态满足条件**

- 有继承关系
- 子类重写父类中的虚函数

**多态使用条件**

- 父类指针或引用指向子类对象

**重写：** 函数返回值类型 函数名 参数列表 完全一致称为重写

```

class Animal
{
public:
    //虚函数
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

//猫类
class Cat :public Animal
{
public:
    //重写 函数返回值类型 函数名 参数列表 完全相同
    virtual void speak()
    {
        cout << "小猫在说话" << endl;
    }
};

```

当子类重写父类的虚函数

子类中的虚函数表 内部 会替换成 子类的虚函数地址



vfptr - 虚函数（表）指针  
v - virtual  
f - function  
ptr - pointer

vftable - 虚函数表  
v - virtual  
f - function  
table - table

当父类的指针或者引用指向子类对象时候，发生多态

```

Animal & animal = cat;
animal.speak();

```

### 3.4.7.2 多态案例一-计算器类

**案例描述：**

分别利用普通写法和多态技术，设计实现两个操作数进行运算的计算器类

**多态的优点：**

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

写之前，先想想，看看能不能用多态

## 示例:

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //分别利用普通写法和多态技术实现计算器
6
7  //普通写法
8  class Calculator {
9  public:
10     int getResult(string oper)
11     {
12         if (oper == "+") {
13             return m_Num1 + m_Num2;
14         }
15         else if (oper == "-") {
16             return m_Num1 - m_Num2;
17         }
18         else if (oper == "*") {
19             return m_Num1 * m_Num2;
20         }
21
22         //如果要提供新的运算，需要修改源码
23         //在真实开发中，提倡 开闭原则
24         //开闭原则：对的扩展进行开发，对修改进行关闭
25
26     }
27
28     public:
29         int m_Num1;
30         int m_Num2;
31 };
32
33
34
35 void test01()
36 {
37     //普通实现测试
38     Calculator c;
39     c.m_Num1 = 10;
40     c.m_Num2 = 10;
41     cout << c.m_Num1 << " + " << c.m_Num2 << " = " << c.getResult("+") << endl;
42
43     cout << c.m_Num1 << " - " << c.m_Num2 << " = " << c.getResult("-") << endl;
44
45     cout << c.m_Num1 << " * " << c.m_Num2 << " = " << c.getResult("*") << endl;
46 }
47
48 //利用多态实现计算器
49 //实现计算器的抽象类
50 //多态好处：
51 //1、组织结构清洗
```

```
52 //2、可读性强
53 //3、对于前期和后期扩展以为维护性高
54
55 class AbstractCalculator
56 {
57 public:
58     //虚函数
59     virtual int getResult()
60     {
61         return 0;
62     }
63
64     int m_Num1;
65     int m_Num2;
66 };
67
68 //加法计算器类
69 class AddCalculator : public AbstractCalculator
70 {
71     int getResult()
72     {
73         return m_Num1 + m_Num2;
74     }
75 };
76 //减法运算器类
77 class SubCalculator : public AbstractCalculator
78 {
79     int getResult()
80     {
81         return m_Num1 - m_Num2;
82     }
83 };
84 //乘法计算器类
85 class MulCalculator : public AbstractCalculator
86 {
87     int getResult()
88     {
89         return m_Num1 * m_Num2;
90     }
91 };
92
93 //测试函数
94 void test02()
95 {
96     //多态使用条件
97     //父类指针或者引用执行子类对象
98
99     //加法
100     AbstractCalculator * abc = new AddCalculator;
101     abc->m_Num1 = 10;
102     abc->m_Num2 = 10;
103     cout << abc->m_Num1<<"+" << abc->m_Num2 << "=" << abc->getResult() << endl;
104     //用完后记得销毁
```

```

105     delete abc;
106
107     //减法
108     abc = new SubCalculator;
109     abc->m_Num1 = 100;
110     abc->m_Num2 = 100;
111     cout << abc->m_Num1 << "-" << abc->m_Num2 << "=" << abc->getResult() << endl;
112     //用完后记得销毁
113     delete abc;
114
115     //乘法
116     abc = new MulCalculator;
117     abc->m_Num1 = 100;
118     abc->m_Num2 = 100;
119     cout << abc->m_Num1 << "*" << abc->m_Num2 << "=" << abc->getResult() << endl;
120     //用完后记得销毁
121     delete abc;
122
123 }
124
125
126 int main()
127 {
128     //test01();
129     test02();
130     system("pause");
131     return 0;
132 }

```

总结：C++开发提倡利用多态设计程序架构，因为多态优点很多