

数据类型_类与对象

一、封装

就是信息隐藏，没什么好说的。我们这里所讨论的主要是类的几种操作：继承、派生和复合。

二、概念

我们有一个已有的类，比如是人。然后我们想要定义学生，显然地，学生是人，所以我们就可以在人的基础上定义学生。人就是基类（父类），学生就是对应的派生类（子类）。这是一种 (is a kind of / is a) 的关系，这里面还有一种特殊的基类：虚基类。虚基类的作用比较特殊，主要是提供接口，下面我们会提到，

但是另一个，就是复合，复合其实更好理解，是 (part of) 的关于。我们有一个类点，然后有了一个新的类线段，显然地，线段由起点和终点定义。所以在线段类中我们会把点类作为其一个数据对象。这就是复合。

二、继承与复合的语法

2.1.继承

2.1.1.基础语法

```
1  class 派生类类名B: <Access> 基类类名A //access表示访问权限:
   public/private/protected
2  {
3      private:
4          ...
5      public:
6          ...
7      protected:
8          ...
9  }
```

基类 A 派生出派生类 B。的部分，系统有默认值：class就是private，struct就是public。

关于继承最重要的一点就是：子类继承了父类的什么:数据和函数。我们通常情况下会将之设计为public，这样，子类中我们就可以调用父类的public成员。这也是我们最常用的做法。

无论是私有 (private) 继承还是公有(public)继承，我们需要注意的是。都没有办法访问基类的私有成员。区别在于继承之后：公有继承下，子类继承了父类的 public 和 protected 成员，并在子类中作为对应的 public 和 protected 成员被访问，显然地，也可以被下一级所继承。而私有继承之下，所有被继承的成员都相当于一个 private 成员，显然不可以被下一级继承所访问。这就是两者的区别。注意：protected成员在public继承中，是可以一直向下传递继承的。而保护继承的含义就是父类的数据成员全部作为 protected成员被继承。

综上，所谓的区别，只需要把握以下几点：

- 继承之后，基类的 private 成员依然不可以被子类访问
- 继承的含义，就是父类的数据成员在子类中承担什么角色 (public、private、还是protected)

- public继承下，基类的public->派生类的public，基类的protected->派生类的protected

示例：考虑下段代码：

```

1  class A
2  {
3      public:
4          void fuc(){printf("%d %d\n",x,y);};
5          A(int a = 0,int b = 0)
6              :x(a),y(b),z(a)
7          { };
8      private:
9          int x,y;
10     protected:
11         int z;
12 };
13 class B:private A
14 {
15     public:
16         void Print(){fuc();};
17         void Print2(){printf("%d\n",z);};
18 };
19 class C:public B
20 {
21     public:
22         void Print3(){printf("%d\n",z);}; //报错，z->A中的private，C没有办法访问
23 };

```

2.1.2.继承的构造函数

首先子类继承父类的数据成员和成员函数，但是并不包括构造函数和析构函数。为了在子类中实现对于父类的初始化，我们需要在子类的构造函数中调用父类的构造函数。进而完成对于父类的初始化。如下代码所示：

```

1  class X{
2      private:
3          double a;
4      public:
5          X(double x = 0) :a(x) { }; //构造函数
6          double getX() const {return a;};
7  };
8  class Y
9  {
10     private:
11         double b;
12     public:
13         Y(double x = 0) :b(x) { }; //构造函数
14         double getY() const {return b;};
15 };
16 class point:public X,public Y
17 {
18     public:
19         point(double x = 0,double y = 0)
20             :X(x),Y(y) //注意，调用基类X和Y中的构造函数实现对于基类的初始化
21         { };

```

```

22         void fuc(){
23             printf("%f,%f",getX(),getY());//我们想直接输出x和y坐标，但是我们不
           能够访问基类的Private成员，只能通过public函数
           //这样的一种间接的方式
24         };
25     };

```

这也可以显而易见的得到构造函数的调用顺序：.构造函数调用顺序：先基类，后派生类。

2.1.3.多继承

没什么好说的，后面多跟几个类就好了。在2.1.2.已经给出了例子。没有什么其他特别的地方。但是值得注意的一点是，多继承时候的命名冲突，尽量把它弄成不一样的名字，不然是会出问题的。另外一个，基类也可以自己单独使用。

2.1.4.继承的析构函数

首先，有一点需要明白的是，析构函数这种东西，我们其实用的不算太多。只有出现了指针我们需要回收空间时，才会使用它。但是析构函数涉及到对于指针的处理，很容易出现野指针（指向了一块随机位置地址，不可控）和悬空指针（指向的空间已经被回收了，但指针没有被NULL）的情况。这很不好。

我们这里只给出实例：，看一看在继承的关系中，到底谁的析构函数首先被调用。

```

1  class base1
2  {
3      public:
4          ~base1(){printf("The destruction of base1.\n");};
5  };
6
7  class base2
8  {
9      public:
10         ~base2(){printf("The destruction of base2.\n");};
11     };
12     class X:public base2,public base1
13     {
14         public:
15             ~X(){printf("The destruction of X.\n");};
16     };
17     int main(){
18         X* x = new X;
19         delete x;
20         return 0;
21     }

```

输出结果：

```

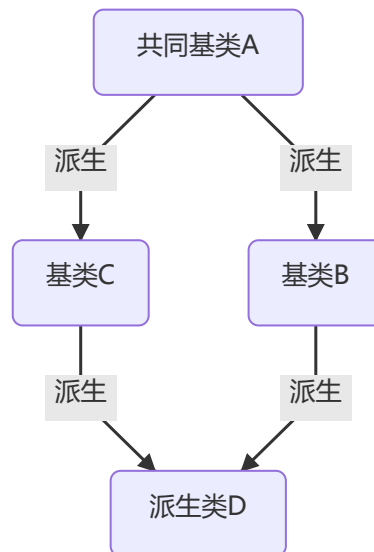
The destruction of X.
The destruction of base1.
The destruction of base2.
请按任意键继续. . .

```

结论，析构函数中，我们首先调用派生类（子类的析构函数），然后在调用基类的。假如是对于字符串，我们想要调用析构函数的话，还要麻烦的多，毕竟涉及到对于指针的操作。字符串的指针操作，还需要简单看一下。

2.1.5.虚基类

关于上面的的多继承的基类的问题，很容易出现菱形继承的情况。如下图所示：



这种继承方式很容易导致二义性（一般这种图里面凡是出现环的，都会出问题）。我们通过派生类D创建出对象d，那么，显然地，我们应该可以通过对象d访问共同基类A里面的数据成员。但是，这种访问却有两条路径：通过B和通过C。B类和C类都是继承的类A，所以在B和C类中应该都有一份关于A的数据类型。于是，在派生类D的对象d中，就储存了两份基类A的成员。一方面造成了冗余，另一方面，也为访问带来了二义性。

为了解决这个问题，我们提供了虚基类的办法。虚基类的解决主要是作于内部实现上，如上所示，当A派生出B和C时，我们在继承的前面加上 `virtual` 的关键字，这样B类和C类所继承的关于公共基类A的数据成员就不再是一份copy，而是两个指针，这两个指针同时指向A中的数据成员，其实质是一样的，就可以避免二义性问题了。对比代码和语法如下：

```
1 class A{
2     public:
3         void fucA(){printf("I'm A\n");};
4 };
5 class B: public A { };
6 class C: public A { };
7 class D:public B,public C { };
8 void test_for_Virtual(){
9     D d;
10    d.fucA();
11 }
12 //存在二义性，会报错。出现了菱形继承。
```

```
D:\_Downloads\Software\Notepad++\npp.7.8.9.bin.x64\new 1.cpp: In function 'void test_for_Virtual()':
D:\_Downloads\Software\Notepad++\npp.7.8.9.bin.x64\new 1.cpp:47:4: error: request for member 'fucA' is ambiguous
d.fucA();
D:\_Downloads\Software\Notepad++\npp.7.8.9.bin.x64\new 1.cpp:26:8: note: candidates are: void A::fucA()
void fucA() {printf("I'm A\n");};
D:\_Downloads\Software\Notepad++\npp.7.8.9.bin.x64\new 1.cpp:26:8: note: void A::fucA()
请按任意键继续. . .
```

针对上述代码的优化，只要在公共基类继承的前面加上一个 `virtual` 关键字即可。

```
1  class A{
2      public:
3          void fucA(){printf("I'm A\n");};
4  };
5  class B:virtual public A { };
6  class C:virtual public A { };
7  class D:public B,public C { };
8  void test_for_virtual(){
9      D d;
10     d.fucA();
11 }
```

结果输出：一切正常



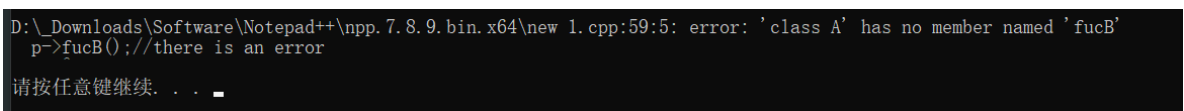
注意：虽然虚函数、纯虚函数、虚基类都有一个虚字，但是他们的目的是不一样的，前两者的作用是为了实现多态，虚基类则是为了解决菱形继承的问题而给出的解决方案。当然，他们之前也有着共同发挥作用的时候：纯虚函数+虚基类提供接口，就是永远嗨神。

2.1.6.指针访问派生类、基类的区别

基类指针可以指向其派生类，这是C++的一个特点，但是这样的指针没有办法访问不存在于基类只存在于派生类的元素。而且派生类类型的指针没有办法指向自己的基类，编译器会报错的。

理解这个，最好从C++哲学层面理解：学者（派生类）一定是人（基类），但是人不一定是学者。所以，一个指向基类的指针，可以访问自己的派生类，因为派生类一定是自己的一种。反过来，就不行了。

```
1  class A{
2      public:
3          void fucA(){printf("I'm A\n");};
4  };
5  class D:public A
6  {
7      public:
8          void fucB(){printf("I'm D\n");};
9  };
10 void test_for_ptr(){
11     A*p = NULL;
12     D d;
13     p = &d;
14     p->fucA(); //但是这里依然可以执行
15     p->fucB(); //这里会报错
16 }
```



```
D:\_Downloads\Software\Notepad++\npp.7.8.9.bin.x64\new 1.cpp:59:5: error: 'class A' has no member named 'fucB'
p->fucB()); //there is an error
```

请按任意键继续. . .

一句很骚的代码

```
1 A* p = new D; // 一个基类A型指针指向其派生类D的实例
2 // 等价于
3 A*p = NULL;
4 D d;
5 p = &d;
```

[补充材料](#)

三、复合

没什么好说的，和继承相对，但是比继承简单多了。复合是 part of 的关系。就是把其他的类作为自己的子对象。可能子对象的话，结构体会多一点。毕竟方便。给一个例子。没什么特殊语法，看看就好。唯一需要注意的就是，显然子对象的private类型数据不能被访问到。

```
1 void test_for_compound(){
2     class name{
3         private: string n;
4         public:
5             string getN() const{return n;};
6             name(string s = ""):n(s){};
7
8     };
9     class weight{
10        private: string w;
11        public:
12            string getW() const{return w;}; // 额外一个函数用来获取private类型
13            weight(string s = ""):w(s){};
14    };
15    class person{
16        private:
17            name na;
18            weight we;
19        public:
20            person(string s1,string s2)
21                :na(s1),we(s2)
22                { };
23        void output()
24        {printf("%s\n%s\n",na.getN().c_str(),we.getW().c_str());}; // 注意c_str()函数用
25        // 来转换成%s
26    };
27    person p("htx","70kg");
28    p.output();
29 }
```

四、多态

在虚函数我们已经简单讲了讲，这里就不再多说了。额外提出一点：模板也是一种实现多态的手段。另外，多态性的定义：

所谓多态性是指发出同样的消息被不同类型的对象接收时导致完全不同的行为。这里所说的消息主要是指对类的成员函数的调用，而不同的行为是指不同的实现。利用多态性，用户只需发送一般形式的消息，而将所有的实现留给接收消息的对象。对象根据所接收到的消息执行相应的动作（即操作）。多态性的实现包括两种，其一是通过函数以及运算符重载，另一则是通过虚函数结合基类指针或者引用实现。

2.1.5.虚函数

虚函数是为了实现C++的多态而实现的概念。分为纯虚函数和普通虚函数。考虑下段代码：

```
1  class base
2  {
3      public:
4          void fuc1(){printf("I'm base::fuc1!\n");};
5          virtual void fuc2 (){ printf("I'm base::virtual fuc2!\n");};
6  };
7  class derive:public base
8  {
9      public:
10         void fuc1(){printf("I'm derive::fuc1!\n");};
11         void fuc2(){printf("I'm derive::fuc2!\n");};
12 };
13 base a;
14 derive b;
15 base*p = &a;
16 p->fuc1();
17 p->fuc2();
18 p = &b;
19 p->fuc1();
20 p->fuc2();
```

输出结果如下：

The image shows a terminal window with a black background and white text. The output of the program is displayed as follows:
I'm base::fuc1!
I'm base::virtual fuc2!
I'm base::fuc1!
I'm derive::fuc2!
请按任意键继续. . .

为了实现多态性。我们提供了多态性，意义就是说我们希望通过一个东西，可控的访问多个东西。基类指针就是这么个玩意儿，这个东西指向了之类，却可以实现对于派生类数据的访问。这就是一种多态，但是这种多态却有局限性，在上述代码中，可以注意到，关于函数 `fuc1`。派生类中对其进行了重载，然后，通过基类指针，我们就访问不到它了。这显然不可以。所以就有了虚函数（加在基类上）。语法很简单，如上，就是在返回值类型前面加上一个关键字 `virtual`。这样，我们在访问具有相同名字的函数的时候，这个函数就彻底地交由指针控制了：指向基类，基类的函数；指向派生类，派生类的函数。

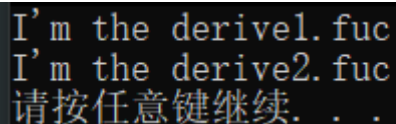
另外，注意一个专有名字：静态联编和动态联编指的就是虚函数的这块内容。

2.1.6.纯虚函数

加上虚基类，就是一个天生的接口，可以作为一个抽象类：比如 `shape` 这种。创建一个对象不太合适，但是作为基类又很合适。纯虚函数的语法很简单，在函数原型的后面加一个 `=0`。一个基类如果包含一个或一个以上的纯虚函数，他就是抽象基类，抽象基类不能也没必要定义自己的对象。如果在基类声明了虚函数，则在派生类中凡是与该函数有**相同的函数名、函数类型、参数个数和类型的函数，均为虚函数**(不论在派生类中是否用 `virtual` 声明)。这意味着我们的基类指针在动态联编就可以访问一切派生类了。另外纯虚函数在自己的基类中并不会被定义，他的具体定义在自己的派生类之中。考虑下段代码：

```
1  class base
2  {
3      public:
4          virtual void fuc()=0;
5  };
6  class derive1:public base
7  {
8      public:
9          void fuc(){printf("I'm the derive1.fuc\n");}; //override
10 };
11 class derive2:public base
12 {
13     public:
14         void fuc(){printf("I'm the derive2.fuc\n");}; //override
15 };
16 derive1 d1;
17 derive2 d2;
18 base*p;
19 p = &d1;
20 p->fuc();
21 p = &d2;
22 p->fuc();
```

结果：



```
I'm the derive1.fuc
I'm the derive2.fuc
请按任意键继续. . .
```

应用：利用多态性取代选择结构

工厂模式(也就是策略模式)

这是一段非常漂亮的代码，可以多看看，尤其是这种编程思想。基类当作接口使用，里面基本上就只有一个纯虚函数用来供其子类重载。

```
1  // 基类,定义一个公用接口 RoleOperation, 类里有一个纯虚函数 Op, 供派生类(子类)具体实现:
2  class RoleOperation
3  {
4      public:
5          virtual std::string Op() = 0; // 纯虚函数
6          virtual ~RoleOperation() {} // 虚析构函数
7  };
```



```

8
9 //接下来针对不同的角色类，继承基类，并实现 Op 函数：
10 // 系统管理员(有 A 操作权限)
11 class RootAdminRole : public RoleOperation {
12 public:
13     RootAdminRole(const std::string &roleName)
14         : m_RoleName(roleName) {}
15
16     std::string Op() {
17         return m_RoleName + " has A permission";
18     }
19
20 private:
21     std::string m_RoleName;
22 };
23
24 // 订单管理员(有 B 操作权限)
25 class OrderAdminRole : public RoleOperation {
26 public:
27     OrderAdminRole(const std::string &roleName)
28         : m_RoleName(roleName) {}
29
30     std::string Op() {
31         return m_RoleName + " has B permission";
32     }
33
34 private:
35     std::string m_RoleName;
36 };
37
38 // 普通用户(有 C 操作权限)
39 class NormalRole : public RoleOperation {
40 public:
41     NormalRole(const std::string &roleName)
42         : m_RoleName(roleName) {}
43
44     std::string Op() {
45         return m_RoleName + " has C permission";
46     }
47
48 private:
49     std::string m_RoleName;
50 };
51
52 /*
53 接下来在写一个工厂类 RoleFactory，提供两个接口：
54 用以注册角色指针对象到工厂的 RegisterRole 成员函数
55 用以获取对应角色指针对象的 GetRole 成员函数
56 */
57
58 // 角色工厂
59 class RoleFactory {
60 public:
61     // 获取工厂单例，工厂的实例是唯一的
62     static RoleFactory& Instance() {
63         static RoleFactory instance; // C++11 以上线程安全
64         return instance;
65     }

```

```

66     }
67
68     // 把指针对象注册到工厂
69     void RegisterRole(const std::string& name, RoleOperation* registrar) {
70         m_RoleRegistry[name] = registrar;
71     }
72
73     // 根据名字name，获取对应的角色指针对象
74     RoleOperation* GetRole(const std::string& name) {
75
76         std::map<std::string, RoleOperation*>::iterator it;
77
78         // 从map找到已经注册过的角色，并返回角色指针对象
79         it = m_RoleRegistry.find(name);
80         if (it != m_RoleRegistry.end()) {
81             return it->second;
82         }
83
84         return nullptr; // 未注册该角色，则返回空指针
85     }
86
87 private:
88     // 禁止外部构造和虚构
89     RoleFactory() {}
90     ~RoleFactory() {}
91
92     // 禁止外部拷贝和赋值操作
93     RoleFactory(const RoleFactory &);
94     const RoleFactory &operator=(const RoleFactory &);
95
96     // 保存注册过的角色，key:角色名称，value:角色指针对象
97     std::map<std::string, RoleOperation*> m_RoleRegistry;
98 };
99
100 //把所有的角色注册（聚合）到工厂里，并封装成角色初始化函数InitializeRole:
101 void InitializeRole() // 初始化角色到工厂
102 {
103     static bool bInitialized = false;
104
105     if (bInitialized == false) {
106         // 注册系统管理员
107         RoleFactory::Instance().RegisterRole("ROLE_ROOT_ADMIN", new
RootAdminRole("ROLE_ROOT_ADMIN"));
108         // 注册订单管理员
109         RoleFactory::Instance().RegisterRole("ROLE_ORDER_ADMIN", new
OrderAdminRole("ROLE_ORDER_ADMIN"));
110         // 注册普通用户
111         RoleFactory::Instance().RegisterRole("ROLE_NORMAL", new
NormalRole("ROLE_NORMAL"));
112         bInitialized = true;
113     }
114 }

```

五、相同class的各个对象互为友元

考虑下述代码,解释参注释:

```
1  class complex{
2      private:
3          double re,im;
4      public:
5          complex(double r = 0,double i = 0):re(r),im(i) { };
6          int fuc(const complex& parm){ return parm.im + parm.re;};//注意这一步
7  }
8  //调用
9  void main(){
10     complex a(1,2);
11     complex b(3,4);
12     a.fuc(b);
13     //很奇怪，作为一种类的封装
14     //我们却可以在a的public函数中调用 b 的private数据
15     //这就是题目中的解释
16 }
```