

知识点1【强化训练字符串类String】

知识点2【继承和派生的概述】（了解） 1-2

继承的优点：减少代码的冗余 提高代码的重用性

知识点3【继承的格式】

继承方式分类：

父类个数分类：

注意：

案例1：公有继承 public

总结：

案例2：保护继承protected

总结： protected继承

案例3：私有继承 private

总结： private私有继承

总结：

知识点4【继承的内层结构】（了解）（vs studio）

知识点5【继承中的构造和析构的顺序】

总结：

知识点6【子类中 有父类、对象成员 构造和析构的顺序】

总结：（重要）

知识点7【详解 子类中的构造】

1、子类会默认调用 父类的 无参构造

2、子类 必须显示 使用初始化列表 调用 父类的有参构造

知识点8【父类和子类的同名 成员变量 处理】

- 1、当 父类和子类 成员变量同名时 在子类就近原则 选择本作用域的子类成员
- 2、如果在子类中 必须使用父类中的同名成员 必须加上父类的作用域。
- 3、子类可以借助 父类的公有方法 间接的操作 父类的私有数据（不可见的的数据）

知识点9【父类和子类的同名 成员函数 处理】

案例：1子类继承父类所有成员函数 和成员变量

案例2：子类和父类 同名成员函数

知识点1【强化训练字符串类String】

mystring.h

```
1  #ifndef MYSTRING_H
2  #define MYSTRING_H
3  #include<iostream>
4  using namespace std;
5  class MyString
6  {
7  friend ostream& operator<<(ostream &out, MyString &ob);
8  friend istream& operator>>(istream &in, MyString &ob);
9  private:
10     char *str;
11     int size;
12 public:
13     MyString();
14     MyString(const char *str);
15     MyString(const MyString &ob);
16     ~MyString();
17     int Size(void);
18
19     //重载[]
20     char& operator[](int index);
21     //重载= 参数是对象
22     MyString& operator=(const MyString &ob);
23     //重载= 参数是字符串常量 const char *
24     MyString& operator=(const char *str);
```

```

25
26 //重载+运算符
27 MyString& operator+(const MyString &ob);
28 MyString& operator+(const char *str);
29
30 //重载==运算符
31 bool operator==(const MyString &ob);
32 bool operator==(const char *str);
33
34 };
35
36 #endif // MYSTRING_H
37

```

mystring.cpp

```

1 #include "mystring.h"
2 #include<string.h>
3 #include<iostream>
4 using namespace std;
5 MyString::MyString()
6 {
7     this->str = NULL;
8     this->size = 0;
9     cout<<"无参构造"<<endl;
10 }
11
12 MyString::MyString(const char *str)
13 {
14     cout<<"char *构造函数"<<endl;
15     //申请空间
16     this->str = new char[strlen(str)+1];
17     //拷贝字符串
18     strcpy(this->str, str);
19
20     //更新size
21     this->size = strlen(str);
22 }
23
24 MyString::MyString(const MyString &ob)
25 {
26     cout<<"拷贝构造函数"<<endl;

```

```
27 //申请空间
28 this->str = new char[strlen(ob.str)+1];
29 //拷贝字符串
30 strcpy(this->str, ob.str);
31
32 //更新size
33 this->size = ob.size;
34 }
35
36 MyString::~MyString()
37 {
38     cout<<"析构函数"<<endl;
39     if(this->str != NULL)
40     {
41         delete [] this->str;
42         this->str = NULL;
43     }
44 }
45
46 int MyString::Size()
47 {
48     return this->size;
49 }
50
51 char& MyString::operator[](int index)//index表示数组的下标
52 {
53     //判断下标是否合法
54     if(index >=0 && index < this->size)
55     {
56         return this->str[index];
57     }
58     else
59     {
60         cout<<"index无效"<<endl;
61     }
62 }
63
64 MyString &MyString::operator=(const MyString &ob)
65 {
66     //将ob.str拷贝到 this->str里面
```

```

67 //1、将this->str指向的旧空间 释放掉
68 if(this->str != NULL)
69 {
70 delete [] this->str;
71 this->str = NULL;
72 }
73
74 //根据ob.str的大小申请空间
75 this->str = new char[ob.size+1];
76 strcpy(this->str, ob.str);
77
78 this->size = ob.size;
79
80 return *this;
81 }
82
83 MyString &MyString::operator=(const char *str)
84 {
85 //1、将this->str指向的旧空间 释放掉
86 if(this->str != NULL)
87 {
88 delete [] this->str;
89 this->str = NULL;
90 }
91
92 this->str = new char[strlen(str)+1];
93 strcpy(this->str, str);
94 this->size = strlen(str);
95 return *this;
96 }
97
98 MyString& MyString::operator+(const MyString &ob)
99 {
100 //this 指向的是str5 ob是str6的别名
101 //计算将来两个字符串拼接后的长度
102 int newSize = this->size + ob.size + 1;
103 char *tmp_str = new char[newSize];
104 //清空tmp_str所指向的空间
105 memset(tmp_str, 0, newSize);
106
107 //先将this->str拷贝到 tmp_str中 然后将ob.str追加到tmp_str中

```

```

108 strcpy(tmp_str, this->str);
109 strcat(tmp_str, ob.str);
110
111 static MyString newString(tmp_str);
112 //释放tmp_str指向的临时空间
113 if(tmp_str != NULL)
114 {
115 delete [] tmp_str;
116 tmp_str = NULL;
117 }
118
119 return newString;
120 }
121
122 MyString &MyString::operator+(const char *str)
123 {
124 //计算将来两个字符串拼接后的长度
125 int newSize = this->size + strlen(str) + 1;
126 char *tmp_str = new char[newSize];
127 //清空tmp_str所指向的空间
128 memset(tmp_str, 0, newSize);
129
130 //先将this->str拷贝到 tmp_str中 然后将str指向的字符串 追加到tmp_str中
131 strcpy(tmp_str, this->str);
132 strcat(tmp_str, str);
133
134 static MyString newString(tmp_str);
135 //释放tmp_str指向的临时空间
136 if(tmp_str != NULL)
137 {
138 delete [] tmp_str;
139 tmp_str = NULL;
140 }
141
142 return newString;
143 }
144
145 bool MyString::operator==(const MyString &ob)
146 {
147 if( (strcmp(this->str, ob.str) == 0) && (this->size == ob.size))

```

```

148 {
149     return true;
150 }
151
152     return false;
153 }
154
155 bool MyString::operator==(const char *str)
156 {
157     if( (strcmp(this->str, str) == 0) && (this->size == strlen(str)))
158     {
159         return true;
160     }
161
162     return false;
163 }
164 ostream& operator<<(ostream &out, MyString &ob)
165 {
166     out<<ob.str;//访问了ob中的私有数据 必须设置成友元
167     return out;
168 }
169
170 istream& operator>>(istream &in, MyString &ob)
171 {
172     //记得将原有的数据清楚
173     if(ob.str != NULL)
174     {
175         delete [] ob.str;
176         ob.str =NULL;
177     }
178
179     //获取键盘输入的字符串
180     char buf[1024]="";//临时buf
181     in >> buf;//先得到键盘输入的数据 然后根据buf的实际大小 开辟空间
182
183     ob.str = new char[strlen(buf)+1];
184     strcpy(ob.str, buf);
185     ob.size = strlen(buf);
186
187     return in;
188 }

```

main.cpp

```
1  #include <iostream>
2  #include "mystring.h"
3  using namespace std;
4
5  int main(int argc, char *argv[])
6  {
7      MyString str1("hehe");
8      //自定义对象 必须重载<< (普通全局友元函数实现)
9      cout<<str1<<endl;
10     cout<<"size = "<<str1.Size()<<endl;
11
12     //自定义对象 必须重载>> (普通全局友元函数实现)
13     cin>>str1;
14     cout<<str1<<endl;
15     cout<<"size = "<<str1.Size()<<endl;
16
17     MyString str2("hello class");
18     //重载[]运算符
19     cout<<str2[1]<<endl;
20
21     //重载[]运算符 返回值必须是左值 才能写操作
22     //重载[]运算符 的返回值必须是引用
23     str2[1] = 'E';
24     cout<<str2<<endl;
25
26     MyString str3("hello str3");
27     cout<<"str3:"<<str3<<endl;
28
29     //将对象str2 赋值 给str3
30     //(默认赋值语句 浅拷贝)
31     //必须重载=运算符 (成员函数完成)
32     str3 = str2;
33     cout<<"str3:"<<str3<<endl;
34
35     MyString str4("hello str4");
36     cout<<"str4:"<<str4<< ", size = "<<str4.Size()<<endl;
37     //必须重载=运算符 (成员函数完成)
38     str4="hello string";
```



```
39  cout<<"str4:"<<str4<<" , size = "<<str4.Size()<<endl;
40
41  //重载+运算符
42  MyString str5("我爱大家");
43  MyString str6("我爱千锋");
44  cout<<str5+str6<<endl;
45
46  MyString str7("大家爱我");
47  cout<< str7+"千锋爱我"<<endl;
48
49  //重载==运算符
50  MyString str8("hehe");
51  MyString str9("haha");
52  if(str8 == str9)
53  {
54      cout<<"相等"<<endl;
55  }
56  else
57  {
58      cout<<"不相等"<<endl;
59  }
60
61  if(str8 == "hehe")
62  {
63      cout<<"相等"<<endl;
64  }
65  else
66  {
67      cout<<"不相等"<<endl;
68  }
69
70
71  return 0;
72 }
73
```

运行结果：

char *构造函数

hehe

size = 4

hehe

hehe

size = 4

char *构造函数

e

hEllo class

char *构造函数

str3:hello str3

str3:hEllo class

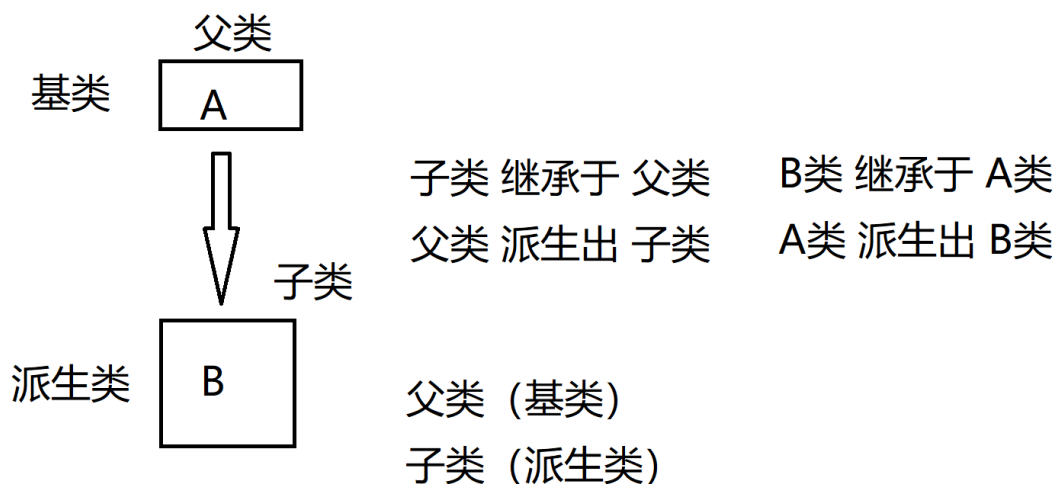
char *构造函数

str4:hello str4, size = 10

str4:hello string, size = 12

char *构造函数
char *构造函数
char *构造函数
我爱大家我爱千锋
char *构造函数
char *构造函数
大家爱我千锋爱我
char *构造函数
char *构造函数
不相等
相等
析构造函数
析构造函数
析构造函数
析构造函数

知识点2 【继承和派生的概述】 （了解） 1-2



继承的优点：减少代码的冗余 提高代码的重用性

知识点3 【继承的格式】

```
1 派生类定义格式：  
2    Class 派生类名 : 继承方式 基类名{  
3        //派生类新增的数据成员和成员函数  
4    };  
5    class 子类: 继承方式 父类名{  
6        //子类新增的数据成员和成员函数  
7    };
```

继承方式分类：

public : 公有继承 （重要）

private : 私有继承

protected : 保护继承

父类个数分类：

单继承：指每个派生类只直接继承了一个基类的特征 （一个父类 派生出 一个子类）

多继承：指多个基类派生出一个派生类的继承关系,多继承的派生类直接继承了不止一个基类的特征（多个父类 派生出 一个子类）

注意：

子类继承父类，子类拥有父类中全部成员变量和成员方法（除了构造和析构之外的成员方法），但是在子类中，继承的成员并不一定能直接访问，不同的继承方式会导致不同的访问权限。

案例1：公有继承 public

```
1 //设置一个父类
```

```

2  class Base
3  {
4  public:
5      int a;
6  private:
7      int b;
8  protected:
9      int c;
10 };
11
12 //设置一个子类
13 class Son:public Base
14 {
15 public:
16     //父类中的public数据 在子类中 也是public
17     //父类中的private数据 在子类中 是不可见的
18     //父类中的protected数据 在子类中 是protected的
19     //子类的内部
20     void showSon()
21     {
22         //b = 200;//不能直接访问
23         c = 300;//在子类 内部是可以访问的
24     }
25 };
26
27 void test01()
28 {
29     //子类的外部
30     Son ob;
31     ob.a = 100;
32     cout<<"父类中的public数据a = "<<ob.a<<endl;
33
34     //ob.b = 200;//在子类外 访问不了
35     //ob.c = 200;//在子类外 访问不了
36
37 }

```

总结:

父类中的public数据 在子类中 也是public

父类中的private数据 在子类中 是不可见的

父类中的protected数据 在子类中 是protected的

(public 继承 父类中的私有数据 在子类 不可见 其他保持原样)

案例2：保护继承protected

```
1 //保护继承
2 class Son1:protected Base
3 {
4     private:
5
6     public:
7     //父类中的public数据 在子类中 也是protected
8     //父类中的private数据 在子类中 是不可见的
9     //父类中的protected数据 在子类中 是protected的
10    //子类的内部
11    void showbase(){
12        a = 100;//子类内部可访问
13        //b = 200;//不能直接访问
14        c = 300;//子类内部可访问
15    }
16 };
17 void test02()
18 {
19     Son1 ob;
20     //ob.a;//子类外不可访问
21     //ob.b;//子类外不可访问
22     //ob.c;//子类外不可访问
23 }
```

总结： protected继承

父类中的public数据 在子类中 也是protected

父类中的private数据 在子类中 是不可见的

父类中的protected数据 在子类中 是protected的

(保护继承 父类的私有数据 在子类中 不可见 其他数据 都变保护)

案例3：私有继承 private

```
1 //保护继承
2 class Son2:private Base
3 {
4     private:
5
```

```
6 public:
7 //父类中的public数据 在子类中 也是private
8 //父类中的private数据 在子类中 是不可见的
9 //父类中的protected数据 在子类中 是private的
10 //子类的内部
11 void showbase(){
12 a = 100; //子类内部可访问
13 //b = 200; //不能直接访问
14 c = 300; //子类内部可访问
15 }
16 };
17 void test03()
18 {
19 Son2 ob;
20 //ob.a; //子类外不可访问
21 //ob.b; //子类外不可访问
22 //ob.c; //子类外不可访问
23 }
```

总结: private私有继承

父类中的public数据 在子类中 也是private

父类中的private数据 在子类中 是不可见的

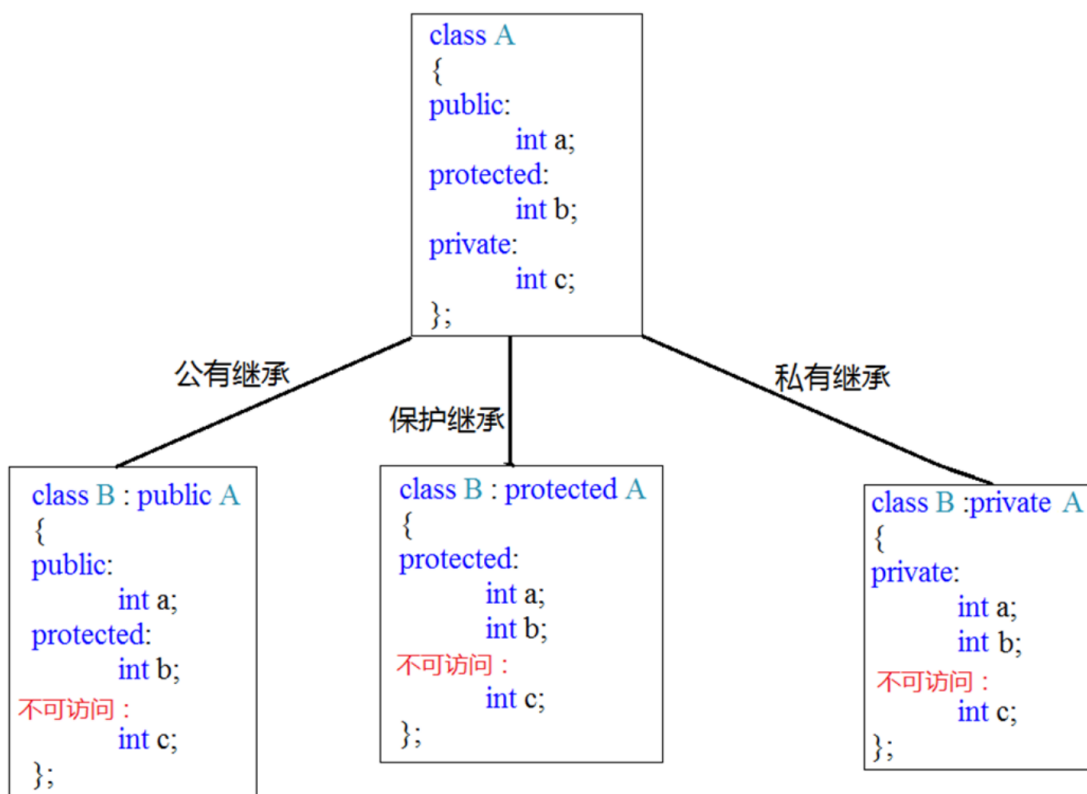
父类中的protected数据 在子类中 是private的

(私有继承 父类中的私有数据在子类中 不可见 其他变成私有)

总结:

公有派生		私有派生		保护派生	
基类属性	派生类权限	基类属性	派生类权限	基类属性	派生类权限
私有	不能访问	私有	不能访问	私有	不能访问
保护	保护	保护	私有	保护	保护
公有	公有	公有	私有	公有	保护

不管啥继承方式：父类中的私有数据在 子类中不可见



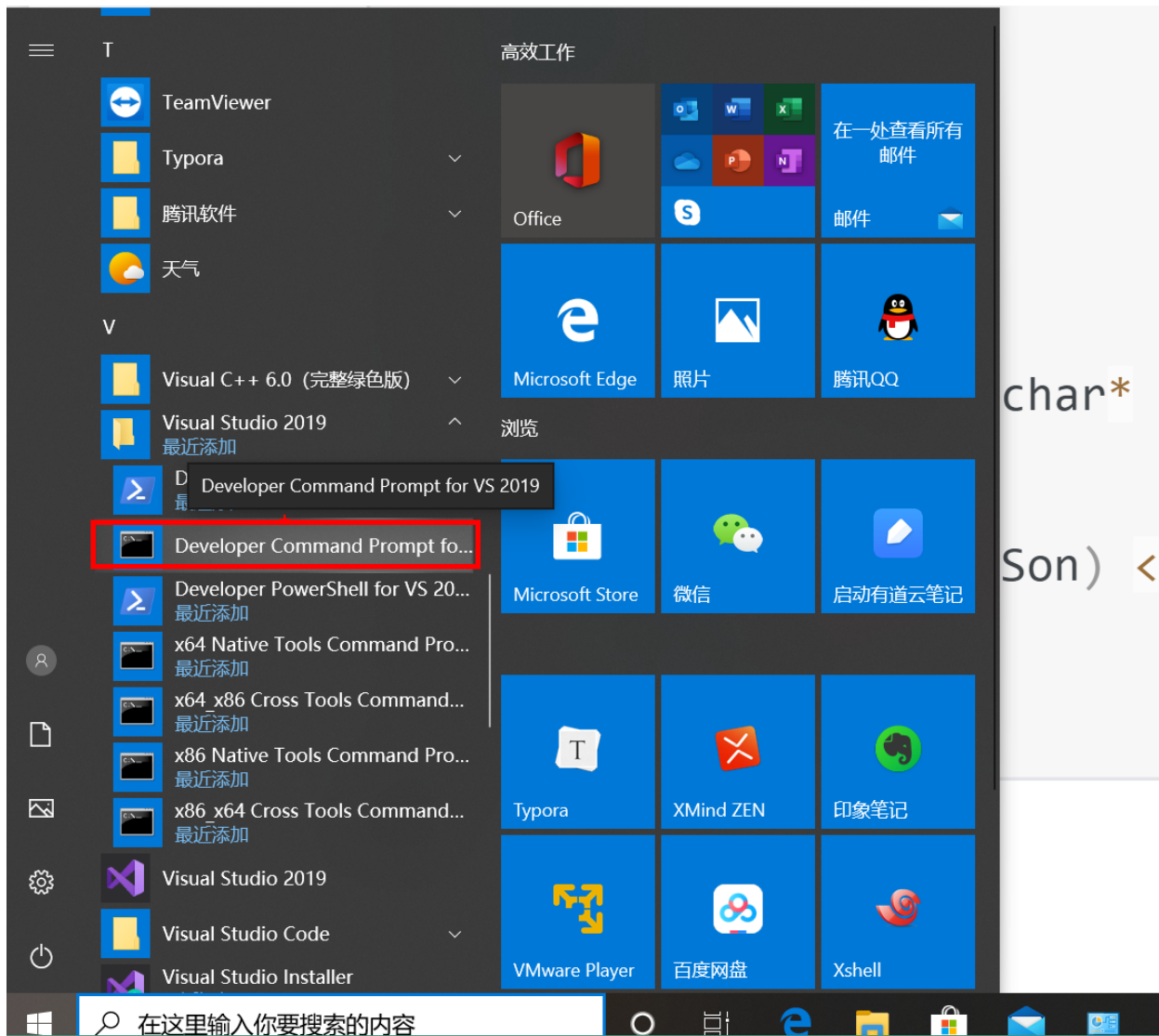
知识点4 【继承的内层结构】（了解）（vs studio）

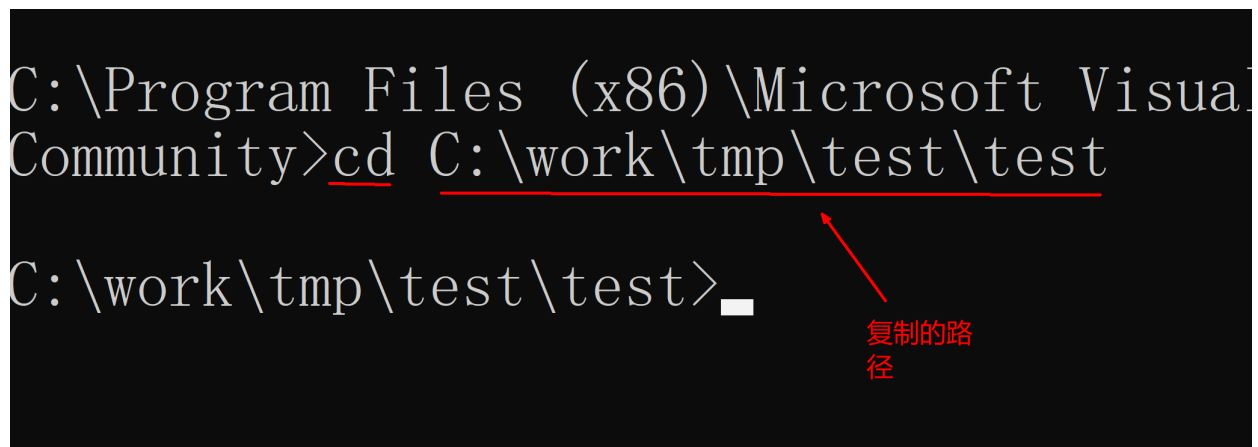
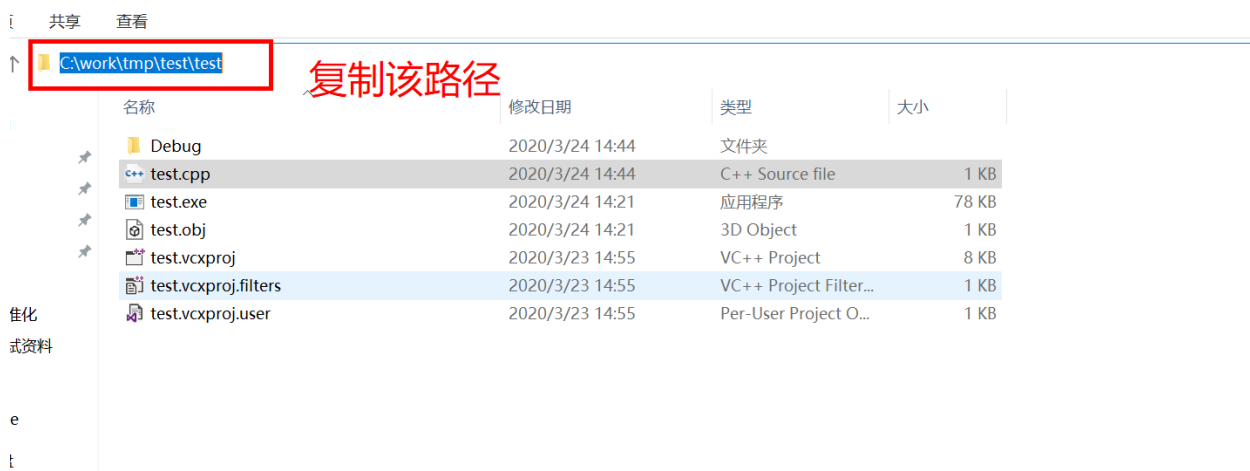
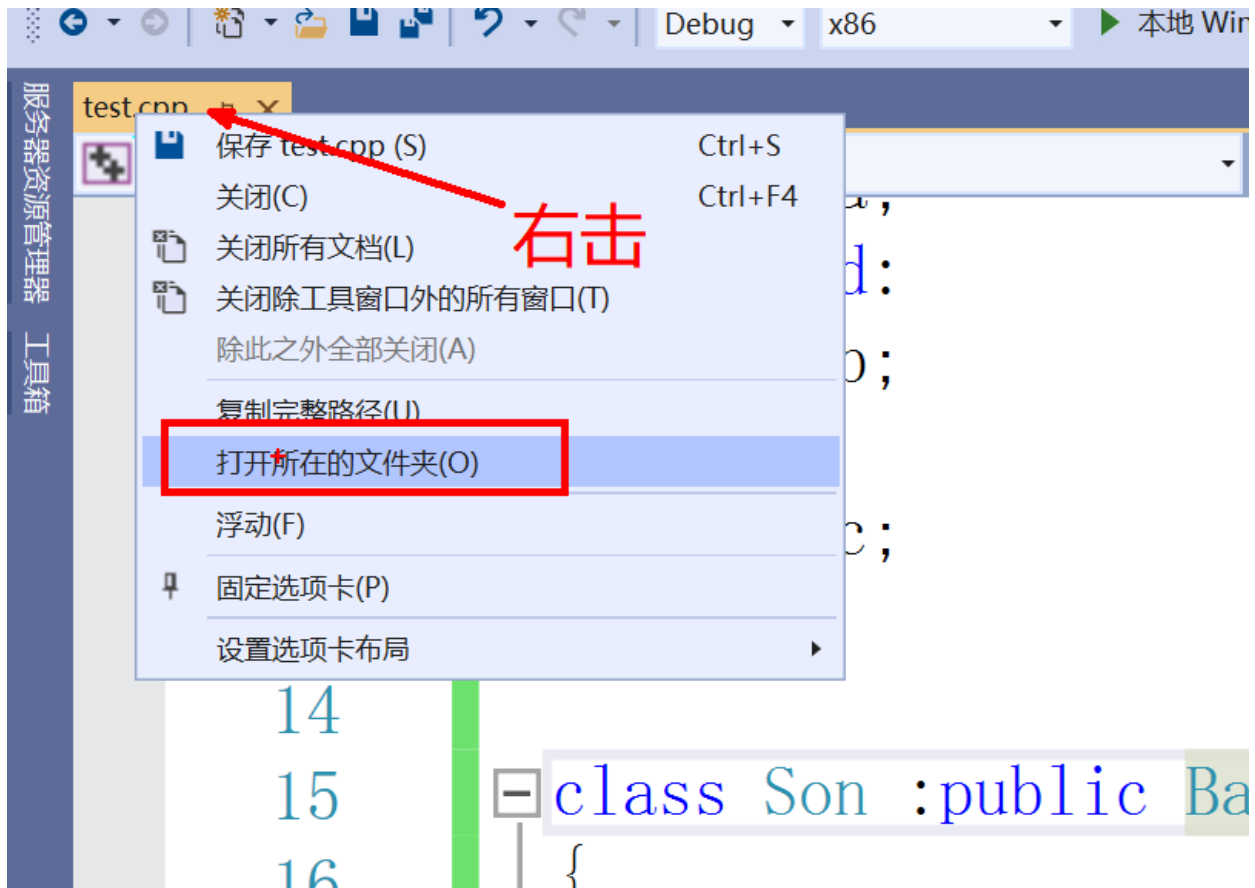
```
1 class Base
```



```
2 {
3 public:
4     int a;
5 protected:
6     int b;
7 private:
8     int c;
9 };
10
11 class Son :public Base
12 {
13 public:
14     int d;
15     int e;
16 };
17 int main(int argc, char* argv[])
18 {
19     cout << sizeof(Son) << endl;
20     return 0;
21 }
```

步骤:





cl /d1 reportSingleClassLayoutSon test.cpp

```
C:\work\tmp\test\test>cl /d1 reportSingleClassLayout
Son test.cpp
```

Son类的布局:

```
class Son          size(20):
    +---+
0    | +---+ (base class Base)
0    | |
4    | | a
8    | | b
    | | c
    | +---+
12   | d
16   | e
    | +---+
```

知识点5 【继承中的构造和析构的顺序】

```
1  class Base
2  {
3  public:
4      Base()
5      {
6          cout<<"父类的无参构造函数"<<endl;
7      }
8      ~Base()
9      {
10         cout<<"父类中的析构函数"<<endl;
11     }
12 };
13 class Son:public Base
14 {
15 public:
16     Son()
17     {
```

```

18  cout<<"子类的无参构造"<<endl;
19  }
20  ~Son()
21  {
22  cout<<"子类中的析构函数"<<endl;
23  }
24  };
25  void test01()
26  {
27  Son ob1;
28  }

```

运行结果：

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

```

父类的无参构造函数
子类的无参构造
子类中的析构函数
父类中的析构函数

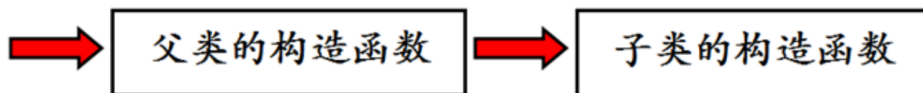
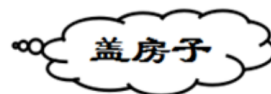
```

总结：

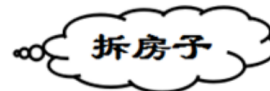
构造顺序：父类（基类）构造 -----> 子类（派生类）构造

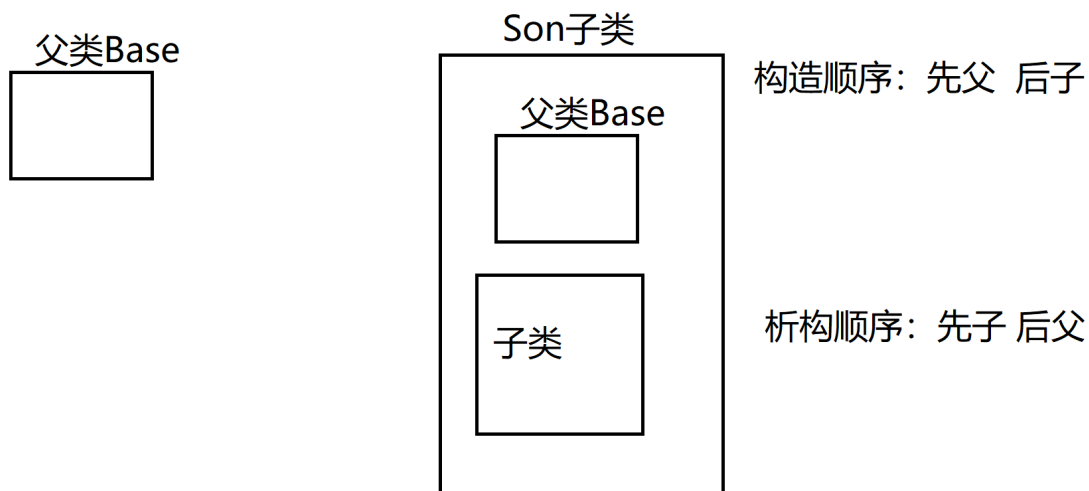
析构顺序：子类（派生类）析构-----> 父类（基类）析构

子类构造函数的执行顺序：



子类析构函数的执行顺序：



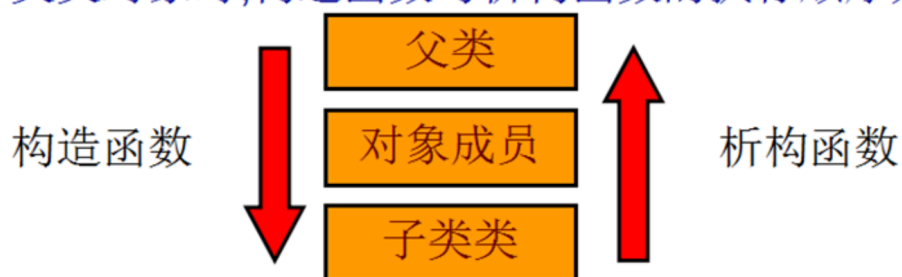


知识点6 【子类中 有父类、对象成员 构造和析构的顺序】

父类的构造和析构 对象成员的构造和析构 子类自身的构造和析构

总结: (重要)

在子类类对象时,构造函数与析构函数的执行顺序为:



```

1  class Other
2  {
3  public:
4      Other()
5      {
6          cout<<"对象成员的构造函数"<<endl;
7      }
8      ~Other()
9      {
10         cout<<"对象成员的析构函数"<<endl;
11     }
12 };
13 class Base

```

```
14 {
15 public:
16     Base()
17     {
18         cout<<"父类的无参构造函数"<<endl;
19     }
20     ~Base()
21     {
22         cout<<"父类中的析构函数"<<endl;
23     }
24 };
25 class Son:public Base
26 {
27 public:
28     Son()
29     {
30         cout<<"子类的无参构造"<<endl;
31     }
32     ~Son()
33     {
34         cout<<"子类中的析构函数"<<endl;
35     }
36
37     Other ob;//对象成员
38
39 };
40 void test01()
41 {
42     Son ob1;
43 }
```

运行结果：

父类的无参构造函数
对象成员的构造函数
子类的无参构造
子类中的析构函数
对象成员的析构函数
父类中的析构函数

知识点7【详解 子类中的构造】

1、子类会默认调用 父类的 无参构造

2、子类 必须显示 使用初始化列表 调用 父类的有参构造

调用形式：父类名称。

```
1 Son(int a,int b):Base(a),b(b)
2 {
3     //this->b = b;
4 }
```

```
1 class Base
2 {
3     private:
4         int a;
5     public:
6
7         Base()
8         {
9             cout<<"父类的无参构造函数"<<endl;
```



```
10  }
11  Base(int a)
12  {
13      this->a = a;
14      cout<<"父类的有参构造函数"<<endl;
15  }
16  ~Base()
17  {
18      cout<<"父类中的析构函数"<<endl;
19  }
20  };
21  class Son:public Base
22  {
23  private:
24      int b;
25  public:
26      Son()
27      {
28          cout<<"子类的无参构造"<<endl;
29      }
30      Son(int b)
31      {
32          this->b = b;
33          cout<<"子类的有参构造函数int"<<endl;
34      }
35
36      //子类必须用 初始化列表 显示的调用父类的有参构造
37      //父类名称(参数)
38      Son(int a,int b):Base(a)//显示的调用父类的有参构造
39      {
40          this->b = b;
41          cout<<"子类的有参构造函数 int int"<<endl;
42      }
43      ~Son()
44      {
45          cout<<"子类中的析构函数"<<endl;
46      }
47  };
48  void test01()
49  {
```

```

50 //子类 默认 会调用 父类的无参构造
51 //Son ob1(10);
52
53 //子类必须用 初始化列表 显示的调用父类的有参构造
54 //父类名称+()
55 Son ob2(10,20);
56
57 }

```

运行结果：

```

父类的有参构造函数
子类的有参构造函数 int int
子类中的析构函数
父类中的析构函数

```

案例提高：

如果父类有参构造：

```

1 Base(int a, int data)
2 {
3     this->a = a;
4     this->data = data;
5     cout<<"父类的有参构造函数"<<endl;
6 }

```

子类想调用 父类有参构造：

```

1 //子类必须用 初始化列表 显示的调用父类的有参构造
2 //父类名称(参数)
3 Son(int a,int b, int c):Base(a,c),b(b)//显示的调用父类的有参构造
4 {
5     //this->b = b;
6     cout<<"子类的有参构造函数 int int"<<endl;
7 }

```

知识点8 【父类和子类的同名 成员变量 处理】

1、当 父类和子类 成员变量同名时 在子类就近原则 选择本作用域的子类成员

2、如果在子类中 必须使用父类中的同名成员 必须加上父类的作用域。

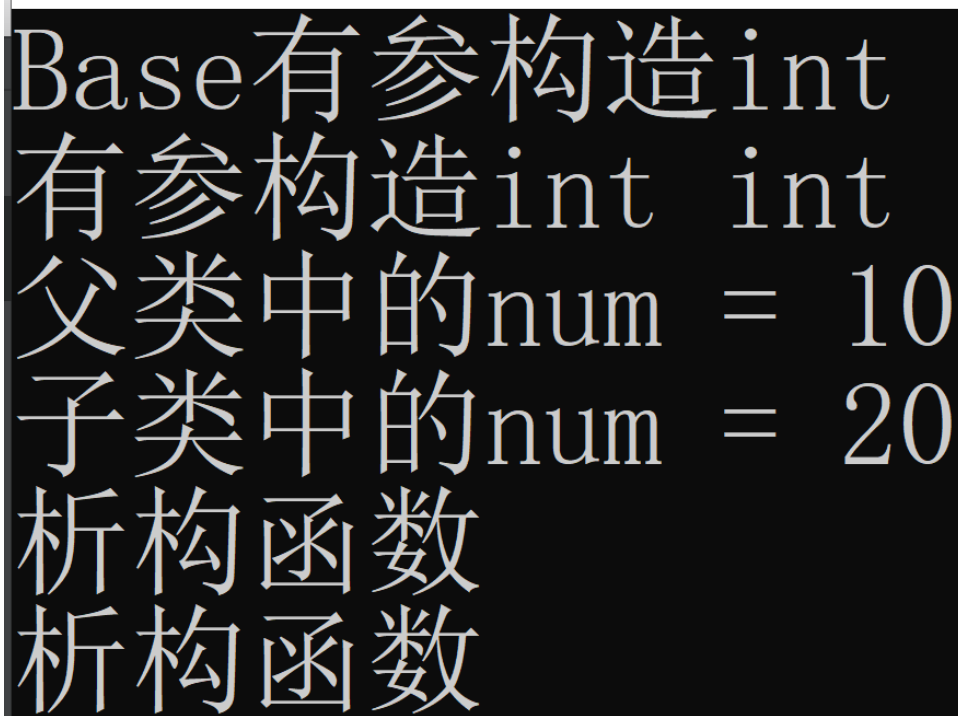
```
1  class Base
2  {
3      //父类的私有数据 一旦涉及继承 在子类中不可见
4  public:
5      int num;
6  public:
7      Base(int num)
8      {
9          this->num = num;
10         cout<<"Base有参构造int"<<endl;
11     }
12     ~Base()
13     {
14         cout<<"析构函数"<<endl;
15     }
16 };
17
18 class Son:public Base
19 {
20 private:
21     int num;
22 public:
23     Son(int num1,int num2):Base(num1)
24     {
25         this->num = num2;
26         cout<<"有参构造int int"<<endl;
27     }
28
29     ~Son()
30     {
31         cout<<"析构函数"<<endl;
32     }
33     void showNum(void)
34     {
35         //如果在子类中 必须使用父类中的同名成员 必须加上父类的作用域
36         cout<<"父类中的num = "<<Base::num<<endl;
```

```

37
38 //当 父类和子类 成员变量同名时 在子类就近原则 选择本作用域的子类成员
39 cout<<"子类中的num = "<<num<<endl;
40 }
41 };
42
43 void test01()
44 {
45     Son ob1(10,20);
46     ob1.showNum();
47 }

```

运行结果：



```

Base有参构造int
有参构造int int
父类中的num = 10
子类中的num = 20
析构函数
析构函数

```

3、子类可以借助 **父类的公有方法** 间接的操作 父类的私有数据（不可见的的数据）

```

1 class Base
2 {
3
4 private:
5     int num;//父类的私有数据 一旦涉及继承 在子类中不可见

```

```
6 public:
7     Base(int num)
8     {
9         this->num = num;
10        cout<<"Base有参构造int"<<endl;
11    }
12    ~Base()
13    {
14        cout<<"析构函数"<<endl;
15    }
16    int getNum(void)
17    {
18        return num;
19    }
20 };
21
22 class Son:public Base
23 {
24 private:
25     int num;
26 public:
27     Son(int num1,int num2):Base(num1)
28     {
29         this->num = num2;
30         cout<<"有参构造int int"<<endl;
31     }
32
33     ~Son()
34     {
35         cout<<"析构函数"<<endl;
36     }
37     void showNum(void)
38     {
39         //如果在子类中 必须使用父类中的同名成员 必须加上父类的作用域
40         cout<<"父类中的num = "<<getNum()<<endl;
41
42         //当 父类和子类 成员变量同名时 在子类就近原则 选择本作用域的子类成员
43         cout<<"子类中的num = "<<num<<endl;
44     }
45 };
```

```

46
47 void test01()
48 {
49     Son ob1(10,20);
50     ob1.showNum();
51 }

```

运行结果：

```

Base有参构造int
有参构造int int
父类中的num = 10
子类中的num = 20
析构函数
析构函数

```

知识点9 【父类和子类的同名 成员函数 处理】

案例：1子类继承父类所有成员函数 和成员变量

```

1 class Base
2 {
3 public:
4     void func(void)
5     {
6         cout<<"父类中的void func"<<endl;
7     }
8     void func(int a)
9     {
10        cout<<"父类中的int func a = "<<a<<endl;
11    }
12 };

```

```

13
14 class Son:public Base
15 {
16 public:
17
18 };
19
20 void test01()
21 {
22     //为啥构造和析构除外?父类的构造和析构 只有父类自己知道该怎么做（构造和析构 系
    统自动调用）
23     //子类会继承父类所有成员函数(构造和析构函数除外) 和成员变量
24     Son ob1;
25     ob1.func();//访问的是父类的void func(void)
26     ob1.func(10);//访问的是父类的func(int a)
27 }

```

案例2：子类和父类 同名成员函数

```

1 class Base
2 {
3 public:
4     void func(void)
5     {
6         cout<<"父类中的void func"<<endl;
7     }
8     void func(int a)
9     {
10         cout<<"父类中的int func a = "<<a<<endl;
11     }
12 };
13
14 class Son:public Base
15 {
16 public:
17     //一旦子类 实现了 父类的同名成员函数 将屏蔽所有父类同名成员函数
18     void func(void)
19     {
20         cout<<"子类中voidfunc"<<endl;
21     }
22 };
23

```

```

24 void test01()
25 {
26     //为啥构造和析构除外?父类的构造和析构 只有父类自己知道该怎么做（构造和析构 系
    统自动调用）
27     //子类会继承父类所有成员函数(构造和析构函数除外) 和成员变量
28     Son ob1;
29     ob1.func();
30     //ob1.func(10);//err //一旦子类 实现了 父类的同名成员函数 将屏蔽所有父类同名
    成员函数
31
32     //如果用户 必须要调用父类 的同名成员函数 必须加作用域
33     ob1.Base::func();//调用父类的void func
34     ob1.Base::func(10);//调用父类的int func
35 }
36 int main(int argc, char *argv[])
37 {
38     test01();
39     return 0;
40 }

```

运行结果：

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_process_stub.exe

子类中voidfunc
父类中的void func
父类中的int func a = 10