

知识点1【虚析构】

1、知识点的引入

2、解决上面的问题 虚析构 （虚函数）

虚析构作用：通过基类指针、引用 释放 子类的所有空间。

虚析构：在虚析构函数前 加virtual修饰

知识点2【纯虚函数 和抽象类】

如果一个类中拥有 纯虚函数 那么这个类 就是抽象类

1、抽象类 派生出 子类，那么在子类必须实现所有的纯虚函数

案例：饮料制作

知识点3【纯虚析构】

知识点4【虚函数 纯虚函数 虚析构 纯虚析构】（重要）

1、虚函数：只是virtual修饰有函数体 （作用于成员函数）

2、纯虚函数：virtual修饰 加=0 没有函数 所在的类为抽象类

3、虚析构：virtual修饰 类中的析构函数

4、纯虚析构：virtual修饰 加=0 必须实现析构的函数体

知识点5【重写 重载 重定义】（了解）

1、重载

2、重定义（隐藏）

3、重写（覆盖）

知识点6【函数模板】 2-1

c++特点：封装、继承、多态

c++特点：面向对象编程、泛型编程

2、用户指定T的类型

3、函数模板 和 普通函数的区别

案例2：函数模板的调用时机

知识点7【函数模板的课堂练习】

知识点8【函数模板具体化】

知识点9【类模板】

知识点10【类模板作为函数的参数】（了解）

知识点11【类模板 派生 普通类】

知识点1【虚析构】

1、知识点的引入

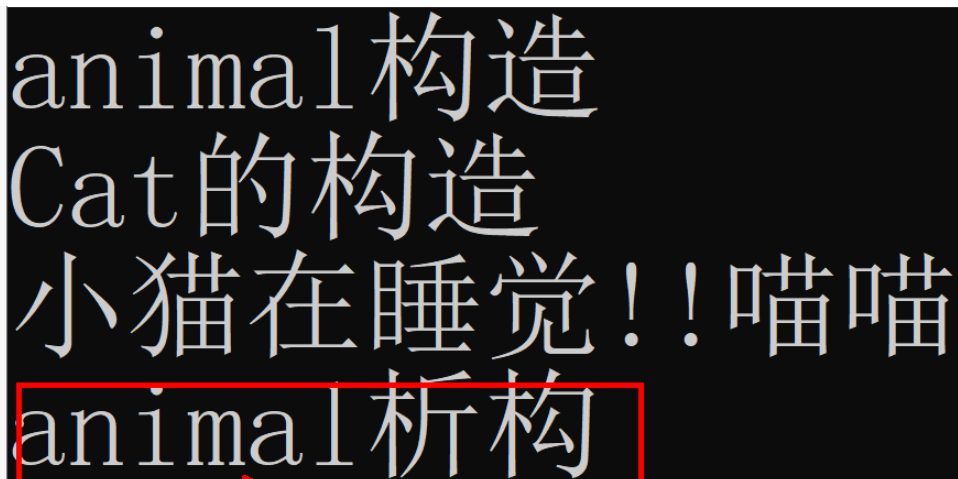
```
1  class Animal{
2  public:
3      //虚函数 本质函数指针 不涉及继承时 指向自身函数（sleep）
4      virtual void sleep(void)
5      {
6          cout<<"动物在睡觉"<<endl;
7      }
8      Animal()
9      {
10         cout<<"animal构造"<<endl;
11     }
12     ~Animal()
13     {
14         cout<<"animal析构"<<endl;
15     }
16 };
17
18 class Cat:public Animal{
19 public:
20     //虚函数 设计到继承 指针子类的sleep
21     virtual void sleep(void)
22     {
```

```

23  cout<<"小猫在睡觉!!喵喵"<<endl;
24  }
25  Cat()
26  {
27  cout<<"Cat的构造"<<endl;
28  }
29  ~Cat()
30  {
31  cout<<"Cat的析构"<<endl;
32  }
33  };
34  void test01()
35  {
36  //通过基类 指针、引用 访问子类的成员函数
37  Animal *p = new Cat;
38  p->sleep();//调用的子类的sleep
39
40  //出现的问题：只能释放 父类析构
41  delete p;
42  }

```

运行结果：



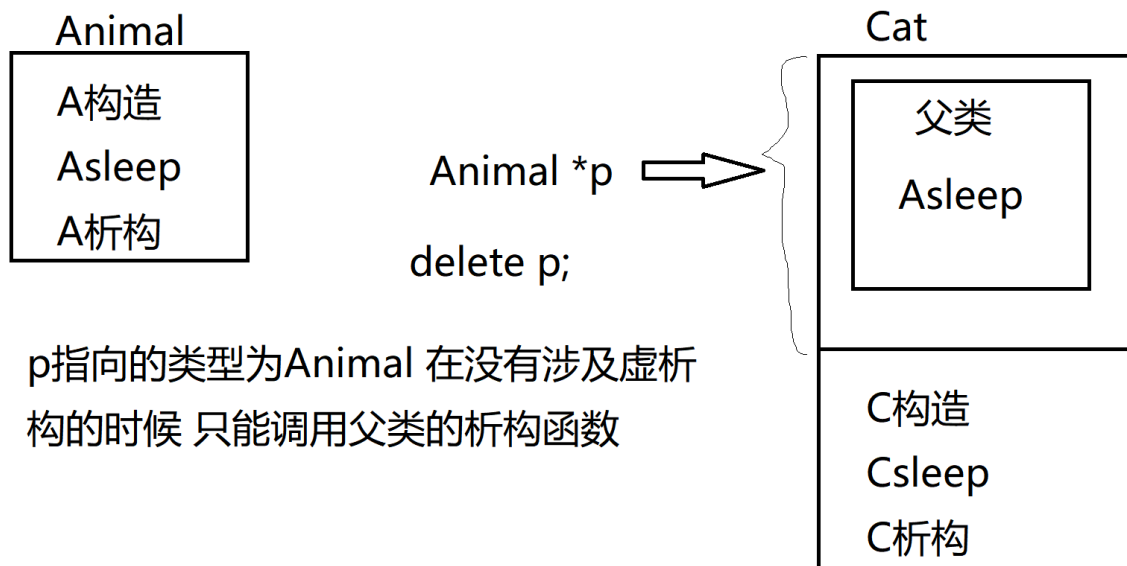
```

animal构造
Cat的构造
小猫在睡觉!!喵喵
animal析构

```

只能释放父类析构

原因分析：



2、解决上面的问题 虚析构（虚函数）

虚析构作用：通过基类指针、引用 释放 子类的所有空间。

虚析构：在虚析构函数前 加virtual修饰

```
1 class Animal{
2 public:
3 //虚函数 本质函数指针 不涉及继承时 指向自身函数（sleep）
4 virtual void sleep(void)
5 {
6 cout<<"动物在睡觉"<<endl;
7 }
8 Animal()
9 {
10 cout<<"animal构造"<<endl;
11 }
12
13 //虚析构
14 virtual ~Animal()
15 {
16 cout<<"animal析构"<<endl;
17 }
18 };
19
20 class Cat:public Animal{
21 public:
22 //虚函数 设计到继承 指针子类的sleep
```

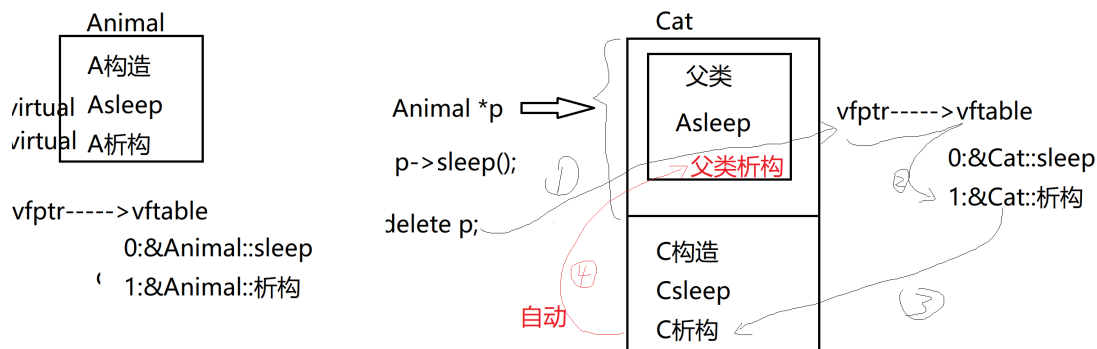
```
23 virtual void sleep(void)
24 {
25     cout<<"小猫在睡觉!!喵喵"<<endl;
26 }
27 Cat()
28 {
29     cout<<"Cat的构造"<<endl;
30 }
31 virtual ~Cat()
32 {
33     cout<<"Cat的析构"<<endl;
34 }
35 };
36 void test01()
37 {
38     //通过基类 指针、引用 访问子类的成员函数
39     Animal *p = new Cat;
40     p->sleep();//调用的子类的sleep
41
42     //如果设置成了 虚析构 就可以释放 子类以及父类的构造函数
43     delete p;
44 }
```

运行结果：

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

```
animal构造
Cat的构造
小猫在睡觉!!喵喵
Cat的析构
animal析构
```

原理分析：



delete p调用的是子类的析构函数，子类析构调用完，系统会自动调用 父类析构。

知识点2 【纯虚函数 和抽象类】

1 纯虚函数格式: `virtual void sleep(void) = 0;`

如果一个类中拥有 纯虚函数 那么这个类 就是抽象类

抽象类 不能实例化对象。

```
1 void test02()
2 {
3     //Animal 抽象类 不能实例化 一个对象
4     //Animal ob;//err
5
6 }
```

1、抽象类 派生出 子类，那么在子类必须实现所有的纯虚函数

如果 漏掉一个 那个子类也是抽象

```
1 class Animal{
2 public:
3     //纯虚函数
4     //如果一个类中拥有 纯虚函数 那么这个类 就是抽象类
5     //抽象类 不能实例化对象
6     virtual void sleep(void) = 0;
7
8     Animal()
9     {
10         cout<<"animal构造"<<endl;
11     }
12
13     //虚析构
```

```

14  virtual ~Animal()
15  {
16  cout<<"animal析构"<<endl;
17  }
18  };
19
20  class Cat:public Animal{
21  public:
22  #if 1
23  //在子类中 必须实现 基类的纯虚函数
24  virtual void sleep(void)
25  {
26  cout<<"小猫在睡觉!!喵喵"<<endl;
27  }
28  #endif
29  Cat()
30  {
31  cout<<"Cat的构造"<<endl;
32  }
33  virtual~Cat()
34  {
35  cout<<"Cat的析构"<<endl;
36  }
37  };
38
39  void test01()
40  {
41  Animal *p = new Cat;
42  p->sleep();
43  delete p;
44  }

```

案例：饮料制作



1. 煮水
2. 冲泡咖啡
3. 倒入杯中
4. 加糖和牛奶

冲咖啡



1. 煮水
2. 冲泡茶叶
3. 倒入杯中
4. 加柠檬

冲茶叶

```
1 //抽象类 提供一个固定的流程 接口
2 class AbstractDrinking{
3 public:
4 //烧水
5 virtual void Boil() = 0;
6 //冲泡
7 virtual void Brew() = 0;
8 //倒入杯中
9 virtual void PourInCup() = 0;
10 //加入辅料
11 virtual void PutSomething() = 0;
12 //规定流程
13 void MakeDrink(){
14 Boil();
15 Brew();
16 PourInCup();
17 PutSomething();
18 }
19 };
20 //制作咖啡
21 class Coffee : public AbstractDrinking{
22 public:
23 //烧水
24 virtual void Boil(){
```



```
25  cout << "煮农夫山泉!" << endl;
26  }
27  //冲泡
28  virtual void Brew(){
29  cout << "冲泡咖啡!" << endl;
30  }
31  //倒入杯中
32  virtual void PourInCup(){
33  cout << "将咖啡倒入杯中!" << endl;
34  }
35  //加入辅料
36  virtual void PutSomething(){
37  cout << "加入牛奶!" << endl;
38  }
39  };
40
41  //制作茶水
42  class Tea : public AbstractDrinking{
43  public:
44  //烧水
45  virtual void Boil(){
46  cout << "煮自来水!" << endl;
47  }
48  //冲泡
49  virtual void Brew(){
50  cout << "冲泡茶叶!" << endl;
51  }
52  //倒入杯中
53  virtual void PourInCup(){
54  cout << "将茶水倒入杯中!" << endl;
55  }
56  //加入辅料
57  virtual void PutSomething(){
58  cout << "加入食盐!" << endl;
59  }
60  };
61  //业务函数
62  void DoBusiness(AbstractDrinking* drink){
63  drink->MakeDrink();
64  delete drink;
```

```
65 }  
66  
67 void test01()  
68 {  
69     //制作 咖啡  
70     DoBussiness(new Coffee);  
71  
72     //制作 茶水  
73     DoBussiness(new Tea);  
74 }
```

运行结果：

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

煮农夫山泉！
冲泡咖啡！
将咖啡倒入杯中！
加入牛奶！
煮自来水！
冲泡茶叶！
将茶水倒入杯中！
加入食盐！

知识点3 【纯虚析构】

纯虚函数：不需要实现函数体

```
1 //纯虚析构函数  
2 class B{  
3 public:
```

```

4  1、virtual修饰 加上=0
5  virtual ~B() = 0;
6  };
7  //2、必须实现 析构函数的函数体
8  B::~~B(){}
9
10 原因：通过基类指针 释放子类对象时 先调用子类析构 再父类析构
11 （如果父类的析构不实现，无法实现调用）

```

```

1  class Base
2  {
3  public:
4      //纯虚析构函数
5      virtual ~Base()=0;
6  };
7  Base::~~Base()
8  {
9
10 }
11 int main(int argc, char *argv[])
12 {
13     //Base ob;//不能实例化对象
14     return 0;
15 }

```

知识点4【虚函数 纯虚函数 虚析构 纯虚析构】 (重要)

1、虚函数：只是virtual修饰有函数体 （作用于成员函数）

目的：通过基类指针或引用 操作 子类的方法

```

1  class Base
2  {
3  public:
4      virtual my_fun(void)
5      {
6          //有函数体;
7      }
8  }

```

2、纯虚函数：virtual修饰 加=0 没有函数 所在的类为抽象类

目的：为子类提供固定的流程和接口

```
1 class Base
2 {
3 public:
4     virtual my_fun(void)=0;
5 }
```

3、虚析构：virtual修饰 类中的析构函数

目的：为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象

```
1 class Base
2 {
3 public:
4     virtual ~Base()
5     {}
6 }
```

4、纯虚析构：virtual修饰 加=0 必须实现析构的函数体

目的：用基类的指针删除派生类对象、同时提供固定接口

```
1 class Base
2 {
3 public:
4     virtual ~Base()=0;
5 }
6 Base::~~Base()
7 {函数体}
```

知识点5【重写 重载 重定义】（了解）

1、重载

同一作用域的同名函数、参数个数，参数顺序，参数类型不同

和函数返回值，没有关系

const也可以作为重载条件 //do(const Teacher& t){} do(Teacher& t)

```
1 int fun(int a){}
2 int fun(int b,int c){}
3 int fun(char b,int c){}
```

2、重定义（隐藏）

有继承

子类（派生类）重新定义父类（基类）的同名成员（非virtual函数）

```
1 class Base{
```

```

2 public:
3     void fun(int){}
4     void fun(int,int){}
5 }
6 class Son:public Base{
7 public:
8     void fun(参数可以不同){} // 重定义
9 }

```

3、重写（覆盖）

有继承

子类（派生类）重写父类（基类）的 **virtual 函数**

函数返回值，函数名字，函数参数，必须和基类中的 **虚函数一致**

```

1 class Base{
2 public:
3     virtual void fun(int){}
4
5 }
6 class Son:public Base{
7 public:
8     virtual void fun(int){} // 重写
9 }

```

知识点6 【函数模板】 2-1

c++ 特点：封装、继承、多态

c++ 特点：面向对象编程、泛型编程

案例：

```

1 /函数模板 template 是关键字
2 //class 和 typename 一样的
3 //T 系统自动推导 或 用户指定
4 template<typename T>
5 void mySwap(T &a, T &b)
6 {
7     T tmp;
8     tmp = a;
9     a = b;

```

```

10  b = tmp;
11  }
12  void test02()
13  {
14      int data1 = 10,data2=20;
15      cout<<"data1 = "<<data1<<"", data2 = "<<data2<<endl;
16      mySwap(data1,data2);//自动推导出T为int
17      cout<<"data1 = "<<data1<<"", data2 = "<<data2<<endl;
18
19
20      char data3 = 'a',data4='b';
21      cout<<"data3 = "<<data3<<"", data4 = "<<data4<<endl;
22      mySwap(data3,data4);//自动推导出T为char
23      cout<<"data3 = "<<data3<<"", data4 = "<<data4<<endl;
24  }
25  int main(int argc, char *argv[])
26  {
27      test02();
28      return 0;
29  }

```

运行结果：

C:\Qt\Qt5.9.0\tools\qtcreator\bin\qtcreator_process_sub.exe

```

data1 = 10, data2 = 20
data1 = 20, data2 = 10
data3 = a, data4 = b
data3 = b, data4 = a

```

2、用户指定T的类型

```

1  void test03()
2  {
3      int data1 = 10,data2=20;
4      cout<<"data1 = "<<data1<<"", data2 = "<<data2<<endl;
5      //用户显示指定 T为int
6      mySwap<int>(data1,data2);
7      cout<<"data1 = "<<data1<<"", data2 = "<<data2<<endl;

```

```
8 }
```

3、函数模板 和 普通函数的区别

```
1 void mySwap(int &a,int &b)
2 {
3     cout<<"普通函数"<<endl;
4     int tmp;
5     tmp = a;
6     a = b;
7     b = tmp;
8 }
9
10 template<typename T>
11 void mySwap(T &a,T &b)
12 {
13     cout<<"函数模板"<<endl;
14     T tmp;
15     tmp = a;
16     a = b;
17     b = tmp;
18 }
19 void test01()
20 {
21     int data1 = 10,data2 = 20;
22     //函数模板和普通函数 都识别 优先选择普通函数
23     mySwap(data1,data2);
24
25     //函数模板和普通函数 都识别 选择函数模板 加<>
26     mySwap<>(data1,data2);
27 }
```

运行结果：

普通函数 函数模板

案例2：函数模板的调用时机


函数模板可以重载

```
1  #include <iostream>
2
3  using namespace std;
4
5  void mySwap(int a,int b)
6  {
7      cout<<"普通函数"<<endl;
8  }
9
10 template<typename T>
11 void mySwap(T a,T b)
12 {
13     cout<<"函数模板"<<endl;
14 }
15 void test01()
16 {
17     int a = 10;
18     char b = 'b';
19
20     //默认优先选择普通函数
21     mySwap(a,a);
22
23     //选择函数模板
24     mySwap<>(a,a);
25
```



```
26 //函数模板 的参数类型 不能自动类型转换
27 //普通函数 的参数类型 能自动类型转换
28 mySwap(a,b);//选择普通函数
29
30
31 //用户 指定T的类型
32 mySwap<int>(a,b);//选择函数模板
33 }
34 int main(int argc, char *argv[])
35 {
36     test01();
37     return 0;
38 }
39
```

运行结果：

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

普通函数
函数模板
普通函数
函数模板

知识点7 【函数模板的课堂练习】

```
1  template<typename T>
2  void myPrintArrayTemplate(T *arr, int len)
3  {
4      int i=0;
5      for(i=0;i<len;i++)
6          cout<<arr[i]<<" ";
7      cout<<endl;
8  }
9
10 template<typename T>
11 void mySortArrayTemplate(T *arr, int len)
12 {
13     int i=0,j=0;
14     for(i=0;i<len-1; i++)
15     {
16         int min = i;
17         for(j=min+1; j<len;j++)
18         {
19             if(arr[min] > arr[j])
20                 min = j;
21         }
22
23         if(min != i)
24         {
25             T tmp = arr[min];
26             arr[min] = arr[i];
27             arr[i] = tmp;
28         }
29     }
30     return;
31 }
32 int main(int argc, char *argv[])
33 {
34     char str[]="hello template";
35     int str_len = strlen(str);
36     int arr[]={5,3,4,7,8,9,1,6,10};
37     int arr_len = sizeof(arr)/sizeof(arr[0]);
38 }
```

```

39 //用函数模板遍历数组
40 myPintArrayTemplate(str, str_len);
41 myPintArrayTemplate(arr, arr_len);
42
43 //用函数模板对数组排序
44 mySortArrayTemplate(str, str_len);
45 mySortArrayTemplate(arr, arr_len);
46
47 //用函数模板遍历数组
48 myPintArrayTemplate(str, str_len);
49 myPintArrayTemplate(arr, arr_len);
50
51 return 0;
52 }

```

运行结果：

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

```

h e l l o   t e m p l a t e
5 3 4 7 8 9 1 6 10
  a e e e h l l l m o p t t
1 3 4 5 6 7 8 9 10

```

知识点8 【函数模板具体化】

```

1 #include <iostream>
2
3 using namespace std;
4
5 class Person
6 {
7     friend ostream& operator<<(ostream &out, Person &ob);
8 public:
9     int a;
10    int b;
11 public:
12    Person(int a,int b)

```

```

13  {
14  this->a = a;
15  this->b = b;
16  cout<<"构造函数"<<endl;
17  }
18  ~Person()
19  {
20  cout<<"析构函数"<<endl;
21  }
22
23  //方法2: 重载>运算符 推荐
24  bool operator>(const Person &ob)
25  {
26  return (this->a > ob.a);
27  }
28  };
29
30  ostream& operator<<(ostream &out, Person &ob)
31  {
32  out<<"a = "<<ob.a<<" , b = "<<ob.b<<endl;
33  return out;
34  }
35  template<typename T>
36  T& myMax(T &a, T &b)
37  {
38  return a>b ? a:b;
39
40  }
41  //方法1:提供函数模板 具体化
42  #if 0
43  template<> Person& myMax<Person>(Person &ob1, Person &ob2)
44  {
45  return ob1.a>ob2.b ? ob1:ob2;
46  }
47  #endif
48
49  void test01()
50  {
51  int data1=10,data2=20;
52

```

```
53  cout<<myMax(data1,data2)<<endl;
54
55  Person  ob1(10,20);
56  Person  ob2(100,200);
57
58  cout<<myMax(ob1,ob2)<<endl;
59  }
60  int  main(int argc, char *argv[])
61  {
62      test01();
63      return 0;
64  }
65
```

运行结果：

20
构造函数
构造函数
a = 100, b = 200
析构函数
析构函数

知识点9 【类模板】

```
1  #include <iostream>
2  #include<string>
3  using namespace std;
```

```
4 //类模板
5 template<class T1, class T2>
6 class Data
7 {
8 private:
9     T1 name;
10    T2 num;
11 public:
12    Data(T1 name, T2 num)
13    {
14        this->name = name;
15        this->num = num;
16        cout<<"有参构造"<<endl;
17    }
18    ~Data()
19    {
20        cout<<"析构函数"<<endl;
21    }
22    void showData(void)
23    {
24        cout<<"name="<<this->name<<", num="<<this->num<<endl;
25    }
26 };
27
28 void test01()
29 {
30     //Data ob1("德玛西亚",100);//err 类模板不允许 自动推导
31     Data<string,int> ob1("德玛西亚",100);
32     ob1.showData();
33
34     Data<int,string> ob2(200, "提莫");
35     ob2.showData();
36
37     Data<int,int> ob3(100,200);
38     ob3.showData();
39
40     Data<string,string> ob4("小炮","德玛");
41     ob4.showData();
42 }
43 int main(int argc, char *argv[])
```

```
44 {  
45     test01();  
46     return 0;  
47 }  
48
```

运行结果：

```
有参构造  
name=德玛西亚, num=100  
有参构造  
name=200, num=提莫  
有参构造  
name=100, num=200  
有参构造  
name=小炮, num=德玛  
析构函数  
析构函数  
析构函数  
析构函数
```

知识点10 【类模板作为函数的参数】（了解）

```
1 #include <iostream>  
2 #include<string>  
3 using namespace std;  
4 //类模板  
5 template<class T1, class T2>  
6 class Data
```

```
7 {
8     friend void addData(Data<string,int> &ob);
9 private:
10     T1 name;
11     T2 num;
12 public:
13     Data(T1 name, T2 num)
14     {
15         this->name = name;
16         this->num = num;
17         cout<<"有参构造"<<endl;
18     }
19     ~Data()
20     {
21         cout<<"析构函数"<<endl;
22     }
23     void showData(void)
24     {
25         cout<<"name="<<this->name<<", num="<<this->num<<endl;
26     }
27 };
28
29 void test01()
30 {
31     //Data ob1("德玛西亚",100);//err 类模板不允许 类型推导
32     Data<string,int> ob1("德玛西亚",100);
33     ob1.showData();
34
35     Data<int,string> ob2(200, "提莫");
36     ob2.showData();
37
38     Data<int,int> ob3(100,200);
39     ob3.showData();
40
41     Data<string,string> ob4("小炮","德玛");
42     ob4.showData();
43 }
44
45 void addData(Data<string,int> &ob)
46 {cout<<"-普通函数---"<<endl;
```



```

47  ob.name += "_vip";
48  ob.num += 2000;
49  return;
50 }
51
52 void test02()
53 {
54  Data<string,int> ob("德玛西亚",18);
55  addData(ob);
56  ob.showData();
57 }
58 int main(int argc, char *argv[])
59 {
60  test02();
61  return 0;
62 }
63

```

运行结果：

有参构造
 ---普通函数---
 name=德玛西亚_vip, num=2018
 析构函数

知识点11 【类模板 派生 普通类】

```

1  #include <iostream>
2  #include<string>
3  using namespace std;
4  //类模板
5  template<class T>
6  class Base{
7  private:
8      T num;
9  public:
10     Base(T num)

```

```
11 {
12     cout<<"有参构造"<<endl;
13     this->num = num;
14 }
15 ~Base()
16 {
17     cout<<"析构函数"<<endl;
18 }
19 void showNum(void)
20 {
21     cout<<num<<endl;
22 }
23 };
24
25 //类模板 派生 普通类 必须给基类 指定T类型
26 class Son1:public Base<int>{
27 public:
28     Son1(int a):Base<int>(a)
29     {
30         cout<<"Son1的构造函数"<<endl;
31     }
32 };
33
34 class Son2:public Base<string>
35 {
36 public:
37     Son2(string a):Base<string>(a)
38     {
39         cout<<"Son2的构造函数"<<endl;
40     }
41 };
42
43 int main(int argc, char *argv[])
44 {
45     Son1 ob1(100);
46     ob1.showNum();
47
48     Son2 ob2("德玛西亚");
49     ob2.showNum();
50     return 0;
```

```
51 }
```

```
52
```

运行结果：

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

有参构造
Son1的构造函数
100
有参构造
Son2的构造函数
德玛西亚
析构造函数
析构造函数