

知识点1【模板的概述】（了解）

知识点2【函数模板】（重要）

知识点3【函数模板的注意点】（重要）

- 1、函数模板 和 普通函数 都识别。（优先选择 普通函数）
- 2、函数模板 和 普通函数 都识别。强制使用函数模板
- 3、函数模板 自动类型推导时 不能对函数的参数 进行 自动类型转换。
- 4、函数模板可以重载
- 5、模板的局限性

解决办法一：运算符重载（推荐）

解决办法二：具体化函数模板

知识点4【类模板】（重要）

- 1、类模板的定义
- 2、函数模板作为类模板的友元
- 3、普通函数作为类模板的友元

知识点5【类模板的继承】（重要）

- 1、类模板 派生处 普通类
- 2、类模板 派生处 类模板

知识点6【类模板头文件 和源文件分离问题】（了解）

知识点7【设计数组类模板】（了解）

知识点1【模板的概述】（了解）

c++面向对象编程思想：封装、继承、多态

c++泛型编程思想：模板

模板的分类：函数模板、类模板

将**功能**相同，类型不同的函数（类）的类型抽象成**虚拟的类型**。当调用函数（类实例化对象）的时候，编译器自动将虚拟的类型 具体化。这个就是函数模板（类模板）。

知识点2 【函数模板】（重要）

关键字template

```
1 //T只能对当前函数有效
2 template<typename T>
3 void swapAll(T &a, T &b)
4 {
5     T tmp = a;
6     a = b;
7     b=tmp;
8     return;
9 }
10
11 void test01()
12 {
13     int a = 10, b = 20;
14     //函数调用时 更具实参的类型 会自动推导T的类型
15     swapAll(a, b);
16     cout<<a<<" "<<b<<endl;
17
18     char a1 = 'a', b1 = 'b';
19     swapAll(a1, b1);
20     cout<<a1<<" "<<b1<<endl;
21 }
```

函数模板 会编译**两次**：

第一次：是对函数模板 本身编译

第二次：函数调用处 将T的类型具体化

函数模板目标：模板是为了实现**泛型**，可以减轻编程的工作量，增强函数的**重用性**。

知识点3 【函数模板的注意事项】（重要）

1、函数模板 和 普通函数 都**识别**。（优先选择 **普通函数**）

```
1 //T只能对当前函数有效 typename可以换成class
2 template<typename T>
3 void swapAll(T &a, T &b)
4 {
```

```

5  T tmp = a;
6  a = b;
7  b=tmp;
8  cout<<"函数模板"<<endl;
9  return;
10 }
11
12 void swapAll(int &a, int &b)
13 {
14     int tmp = a;
15     a = b;
16     b=tmp;
17     cout<<"普通函数"<<endl;
18     return;
19 }
20 void test01()
21 {
22     int a = 10, b = 20;
23     //函数调用时 更具实参的类型 会自动推导T的类型
24     swapAll(a, b);
25     cout<<a<<" "<<b<<endl;
26 }

```

普通函数
20 10

2、函数模板 和 普通函数 都识别。强制使用函数模板

```

1  //强制使用函数模板
2  swapAll<>(a, b);

```

3、函数模板 自动类型推导时 不能对函数的参数 进行 自动类型转换。

```

1  #if 1
2  template<typename T>
3  void myPrintAll(T a, T b)
4  {
5      cout<<a<<" "<<b<<endl;
6      cout<<"函数模板"<<endl;

```

```

7  }
8  #endif
9  void myPrintAll(int a, int b)
10 {
11     cout<<a<<" "<<b<<endl;
12     cout<<"普通函数"<<endl;
13 }
14 void test02()
15 {
16     myPrintAll(10, 20); // 普通函数
17     myPrintAll('a', 'b'); // 函数模板
18     myPrintAll(10, 'b'); // 普通函数
19     // 支持自动类型转换
20     myPrintAll<int>(10, 'b');
21 }

```

```

10 20
普通函数
a b
函数模板
10 98
普通函数
10 98
函数模板

```

如果函数模板显示指明T的具体类型 这时函数模板的参数 可以自动类型转换。

4、函数模板可以重载

```

1  template<typename T>
2  void myPrintAll(T a)
3  {
4      cout<<a<<endl;
5      cout<<"函数模板"<<endl;
6  }
7
8  template<typename T>

```

```
9 void myPrintAll(T a, T b)
10 {
11     cout<<a<<" "<<b<<endl;
12     cout<<"函数模板"<<endl;
13 }
```

5、模板的局限性

当函数模板 推导出 T为数组或其他自定义类型数据 可能导致运算符 不识别。

解决办法一：运算符重载（推荐）

```
1 #include <iostream>
2
3 using namespace std;
4 class Data
5 {
6     friend ostream& operator<<(ostream& out, Data ob);
7 private:
8     int data;
9 public:
10     Data(){}
11     Data(int data)
12     {
13         this->data = data;
14     }
15 };
16 ostream& operator<<(ostream& out, Data ob)
17 {
18     out<<ob.data;
19     return out;
20 }
21
22
23 template<typename T>
24 void myPrintAll(T a)
25 {
26     cout<<a<<endl;
27     cout<<"函数模板"<<endl;
28 }
29 int main(int argc, char *argv[])
30 {
31     myPrintAll(10);
```

```

32  myPrintAll('a');
33
34  Data ob1(100);
35  myPrintAll(ob1);
36
37  return 0;
38  }

```

10
函数模板
a
函数模板
100
函数模板

解决办法二：具体化函数模板

```

1  template<typename T>
2  void myPrintAll(T a)
3  {
4      cout<<a<<endl;
5      cout<<"函数模板"<<endl;
6  }
7
8  class Data
9  {
10     friend void myPrintAll<Data>(Data a);
11     private:
12         int data;
13     public:
14         Data(){}
15         Data(int data)
16         {
17             this->data = data;
18         }
19     };
20
21 //函数模板具体化

```

```

22 template<> void myPrintAll<Data>(Data a)
23 {
24     cout<<a.data<<endl;
25     cout<<"函数模板"<<endl;
26 }
27 int main(int argc, char *argv[])
28 {
29     myPrintAll(10);
30     myPrintAll('a');
31
32     Data ob1(100);
33     myPrintAll(ob1);
34
35     return 0;
36 }

```

知识点4 【类模板】（重要）

1、类模板的定义

类模板 将类中类型 抽象成虚拟类型。

```

1 //类模板
2 template<class T1, class T2>
3 class Data
4 {
5     private:
6         T1 a;
7         T2 b;
8     public:
9         Data(){}
10        Data(T1 a, T2 b)
11        {
12            this->a = a;
13            this->b = b;
14        }
15        void showData()
16        {
17            cout<<a<<" "<<b<<endl;
18        }
19 };

```

类模板 实例化对象 不能自动类型推导（重要）

```

void test01()
{
    //类模板 实例化对象 不能自动类型推导
    //Data ob(10, 20);
    //类模板 实例化对象 必须指明T的类型
    Data<int,int> ob1(10,20);
    ob1.showData();
    Data<int,char> ob2(10,'A');
    ob2.showData();
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\b

```

10 20
10 A

```

类模板 的成员函数 在类外实现:

```

1 //类模板
2 template<class T1, class T2>
3 class Data
4 {
5     private:
6         T1 a;
7         T2 b;
8     public:
9         Data(){}
10        Data(T1 a, T2 b);
11        void showData();
12 };
13 template<class T1, class T2>
14 Data<T1,T2>::Data(T1 a, T2 b)
15 {
16     this->a = a;
17     this->b = b;
18 }
19
20 template<class T1, class T2>
21 void Data<T1,T2>::showData()
22 {
23     cout<<a<<" "<<b<<endl;
24 }

```

2、函数模板作为类模板的友元

```

1 //类模板
2 template<class T1, class T2>
3 class Data

```



```

4 {
5     template<typename T3, typename T4>
6     friend void myPrintData(Data<T3, T4> &ob);
7 private:
8     T1 a;
9     T2 b;
10 public:
11     Data(){}
12     Data(T1 a, T2 b);
13     void showData();
14 };
15
16 //函数模板
17 template<typename T3, typename T4> void myPrintData(Data<T3, T4> &ob)
18 {
19     cout<<ob.a<<" "<<ob.b<<endl;
20 }
21 void test02()
22 {
23     Data<int, char> ob1(100, 'A');
24     myPrintData(ob1);
25 }

```

3、普通函数作为类模板的友元

```

1 //类模板
2 template<class T1, class T2>
3 class Data
4 {
5     friend void myPrintData(Data<int, char> &ob);
6 private:
7     T1 a;
8     T2 b;
9 public:
10     Data(){}
11     Data(T1 a, T2 b);
12     void showData();
13 };
14
15 void myPrintData(Data<int, char> &ob)

```

```

16 {
17     cout<<ob.a<<" "<<ob.b<<endl;
18 }
19 void test02()
20 {
21     Data<int, char> ob1(100, 'A');
22     myPrintData(ob1);
23 }

```

知识点5 【类模板的继承】（重要）

1、类模板 派生处 普通类

```

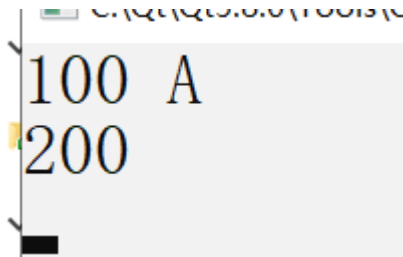
1 #include <iostream>
2
3 using namespace std;
4 //类模板
5 template<class T1, class T2>
6 class Base
7 {
8 private:
9     T1 a;
10    T2 b;
11 public:
12    Base(){}
13    Base(T1 a, T2 b);
14    void showData();
15 };
16 template<class T1, class T2>
17 Base<T1,T2>::Base(T1 a, T2 b)
18 {
19     this->a = a;
20     this->b = b;
21 }
22
23 template<class T1, class T2>
24 void Base<T1,T2>::showData()
25 {
26     cout<<a<<" "<<b<<endl;
27 }
28
29 //类模板派生处普通类

```

```

30 class Son1:public Base<int,char>
31 {
32 public:
33     int c;
34 public:
35     Son1(int a, char b, int c):Base<int,char>(a, b){
36         this->c = c;
37     }
38 };
39 int main(int argc, char *argv[])
40 {
41     Son1 ob1(100, 'A', 200);
42     ob1.showData();
43     cout<<ob1.c<<endl;
44     return 0;
45 }

```



```

100 A
200

```

2、类模板 派生处 类模板

```

1 #include <iostream>
2
3 using namespace std;
4 //类模板
5 template<class T1, class T2>
6 class Base
7 {
8 private:
9     T1 a;
10    T2 b;
11 public:
12    Base(){}
13    Base(T1 a, T2 b);
14    void showData();
15 };
16 template<class T1, class T2>

```

```

17 Base<T1,T2>::Base(T1 a, T2 b)
18 {
19     this->a = a;
20     this->b = b;
21 }
22
23 template<class T1, class T2>
24 void Base<T1,T2>::showData()
25 {
26     cout<<a<<" "<<b<<endl;
27 }
28
29 //类模板派生处类模板
30 template<class T1, class T2, class T3>
31 class Son1:public Base<T1,T2>
32 {
33 public:
34     T3 c;
35 public:
36     Son1(T1 a, T2 b, T3 c):Base<T1,T2>(a, b){
37         this->c = c;
38     }
39 };
40 int main(int argc, char *argv[])
41 {
42     Son1<int,char, int> ob1(100, 'A', 200);
43     ob1.showData();
44     cout<<ob1.c<<endl;
45     return 0;
46 }
47

```

```

100 A
200

```

知识点6 【类模板头文件 和源文件分离问题】（了解）

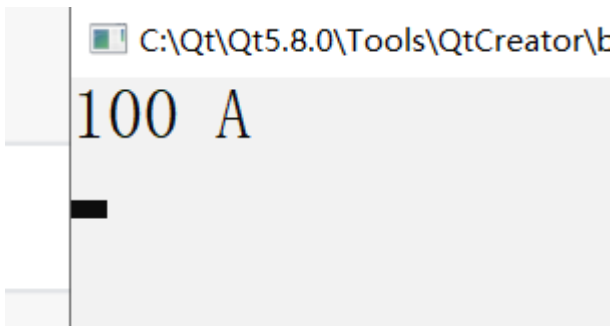
类模板 最好存储在.hpp中

data.hpp

```
1  #ifndef DATA_H
2  #define DATA_H
3  #include<iostream>
4  using namespace std;
5  template<class T1, class T2>
6  class Data
7  {
8  private:
9      T1 a;
10     T2 b;
11 public:
12     Data();
13     Data(T1 a, T2 b);
14     void showData(void);
15 };
16
17 template<class T1, class T2>
18 Data<T1, T2>::Data()
19 {
20     cout<<"无参构造"<<endl;
21 }
22 template<class T1, class T2>
23 Data<T1, T2>::Data(T1 a, T2 b)
24 {
25     this->a = a;
26     this->b = b;
27 }
28 template<class T1, class T2>
29 void Data<T1, T2>::showData(void)
30 {
31     cout<<a<<" "<<b<<endl;
32 }
33 #endif // DATA_H
34
```

main.cpp

```
1 #include <iostream>
2 #include "data.hpp"
3 using namespace std;
4
5 int main(int argc, char *argv[])
6 {
7     Data<int, char> ob1(100, 'A');
8     ob1.showData();
9     return 0;
10 }
11
```



知识点7 【设计数组类模板】（了解）

数组类模板：可以存放任意数据类型

myarray.hpp

```
1 #ifndef MYARRAY_HPP
2 #define MYARRAY_HPP
3 #include<iostream>
4 #include<string.h>
5 #include<string>
6 using namespace std;
7 template<class T>
8 class MyArray
9 {
10     template<typename T1>
11     friend ostream& operator<< (ostream& out, MyArray<T1> ob);
12 private:
13     T *arr; //保存数组首元素地址
14     int size; //大小
15     int capacity; //容量
16 public:
```

```

17  MyArray();
18  MyArray(int capacity);
19  MyArray(const MyArray &ob);
20  ~MyArray();
21  MyArray& operator=(MyArray &ob);
22
23  void pushBack(T elem);
24  void sortArray();
25 };
26
27 #endif // MYARRAY_HPP
28
29 template<class T>
30 MyArray<T>::MyArray()
31 {
32     capacity = 0;
33     size = 0;
34     arr = NULL;
35 }
36
37 template<class T>
38 MyArray<T>::MyArray(int capacity)
39 {
40     this->capacity = capacity;
41     size = 0;
42     arr = new T[this->capacity];
43     memset(arr, 0, sizeof(T)*capacity);
44 }
45
46 template<class T>
47 MyArray<T>::MyArray(const MyArray &ob)
48 {
49     if(ob.arr == NULL)
50     {
51         arr = NULL;
52         size = 0;
53         capacity=0;
54     }
55     else
56     {

```

```

57     capacity = ob.capacity;
58     size = ob.size;
59     arr = new T[capacity];
60     memcpy(arr, ob.arr, sizeof(T)*capacity);
61 }
62
63 }
64
65 template<class T>
66 MyArray<T>::~MyArray()
67 {
68     if(arr != NULL)
69     {
70         delete [] arr;
71     }
72 }
73
74 template<class T>
75 MyArray<T> &MyArray<T>::operator=(MyArray<T> &ob)
76 {
77     //判断this->arr是否存在空间
78     if(arr != NULL)
79     {
80         delete [] arr;
81         arr=NULL;
82     }
83
84     size = ob.size;
85     capacity = ob.capacity;
86     arr = new T[capacity];
87     memset(arr,0,sizeof(T)*capacity);
88     memcpy(arr, ob.arr, sizeof(T)*capacity);
89     return *this;
90 }
91
92 template<class T>
93 void MyArray<T>::pushBack(T elem)
94 {
95     if(size==capacity)//满
96     {

```



```
97 //扩展容量
98 capacity = (capacity==0?1:2*capacity) ;
99 //申请空间
100 T *tmp = new T[capacity];
101
102 if(arr != NULL)
103 {
104 //将旧空间的内容拷贝到新空间
105 memcpy(tmp,arr,sizeof(T)*size);
106 //释放旧空间
107 delete [] arr;
108 }
109
110 arr = tmp;
111 }
112
113 arr[size] = elem;
114 size++;
115 return;
116 }
117
118 template<class T>
119 void MyArray<T>::sortArray()
120 {
121 if(arr == NULL)
122 {
123 cout<<"容器为空间"<<endl;
124 }
125 else
126 {
127 int i=0,j=0;
128 for(i=0;i<size-1;i++)
129 {
130 for(j=0;j<size-i-1;j++)
131 {
132 if(arr[j] > arr[j+1])
133 {
134 T tmp = arr[j];
135 arr[j] = arr[j+1];
136 arr[j+1] = tmp;
```

```

137     }
138     }
139     }
140     }
141     return;
142 }
143 template<typename T1>
144 ostream& operator<<(ostream& out, MyArray<T1> ob)
145 {
146     int i=0;
147     for(i=0;i<ob.size;i++)
148     {
149         out<<ob.arr[i]<<" ";
150
151     }
152
153     return out;
154 }

```

main.cpp

```

1  #include <iostream>
2  #include "myarray.hpp"
3  using namespace std;
4  class Person
5  {
6      friend ostream& operator<<(ostream& out, Person ob);
7  private:
8      int num;
9      string name;
10     float score;
11 public:
12     Person(){}
13     Person(int num,string name, float score)
14     {
15         this->num = num;
16         this->name = name;
17         this->score = score;
18     }
19     bool operator>(const Person &ob)
20     {
21         return num>ob.num;

```

```

22     }
23 };
24 ostream& operator<<(ostream& out, Person ob)
25 {
26     int i=0;
27
28     out<<ob.num<<" "<<ob.name<<" "<<ob.score<<endl;
29     return out;
30 }
31
32 int main(int argc, char *argv[])
33 {
34     MyArray<int> ob1(5);
35     ob1.pushBack(10);
36     ob1.pushBack(30);
37     ob1.pushBack(20);
38     ob1.pushBack(50);
39     ob1.pushBack(40);
40
41     cout<<ob1<<endl;
42     ob1.sortArray();
43     cout<<ob1<<endl;
44
45     MyArray<char> ob2(5);
46     ob2.pushBack('A');
47     ob2.pushBack('C');
48     ob2.pushBack('D');
49     ob2.pushBack('B');
50     ob2.pushBack('F');
51
52     cout<<ob2<<endl;
53     ob2.sortArray();
54     cout<<ob2<<endl;
55
56     MyArray<Person> ob3;
57     ob3.pushBack(Person(100,"lucy", 88.8f));
58     ob3.pushBack(Person(103,"bob", 89.9f));
59     ob3.pushBack(Person(105,"tom", 98.8f));
60     ob3.pushBack(Person(102,"德玛", 99.8f));
61
62     cout<<ob3<<endl;

```

```
63  ob3.sortArray();
64  cout<<ob3<<endl;
65  return 0;
66 }
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proces

```
10 30 20 50 40
10 20 30 40 50
A C D B F
A B C D F
100 lucy 88.8
103 bob 89.9
105 tom 98.8
102 德玛 99.8

100 lucy 88.8
102 德玛 99.8
103 bob 89.9
105 tom 98.8
```