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

```
//T只能对当前函数有效
template<typename T>
void swapAll(T &a, T &b)
{
 T tmp = a;
 a = b;
 b=tmp;
 return;
}

void test01()
{
 int a = 10, b = 20;
 //函数调用时 更具实参的类型 会自动推导T的类型
 swapAll(a, b);
 cout<<a<<" "<<b<<endl;

 char a1 = 'a', b1 = 'b';
 swapAll(a1, b1);
 cout<<a1<<" "<<b1<<endl;
}
```

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【函数模板的注意点】（重要）

1、函数模板 和 普通函数 都识别。（优先选择 普通函数）

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

```
 T tmp = a;
 a = b;
 b=tmp;
 cout<<"函数模板"<<endl;
 return;
}

void swapAll(int &a, int &b)
{
 int tmp = a;
 a = b;
 b=tmp;
 cout<<"普通函数"<<endl;
 return;
}
void test01()
{
 int a = 10, b = 20;
 //函数调用时 更具实参的类型 会自动推导T的类型
 swapAll(a, b);
 cout<<a<<" "<<b<<endl;
}
```

2、函数模板 和 普通函数 都识别。强制使用函数模板

```
//强制使用函数模板
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。

```
#if 1
template<typename T>
void myPrintAll(T a, T b)
{
 cout<<a<<" "<<b<<endl;
 cout<<"函数模板"<<endl;
```

```
}
#endif
void myPrintAll(int a, int b)
{
 cout<<a<<" "<<b<<endl;
 cout<<"普通函数"<<endl;
}
void test02()
{
 myPrintAll(10, 20);//普通函数
 myPrintAll('a','b');//函数模板
 myPrintAll(10,'b');//普通函数
 //支持自动类型转换
 myPrintAll<int>(10,'b');
}
```

```
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

```
template<typename T>
void myPrintAll(T a)
{
 cout<<a<<endl;
 cout<<"函数模板"<<endl;
}

template<typename T>
```

```
void myPrintAll(T a, T b)
{
 cout<<a<<" "<<b<<endl;
 cout<<"函数模板"<<endl;
}
```

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）

```
#include <iostream>

using namespace std;
class Data
{
 friend ostream& operator<<(ostream& out, Data ob);
private:
 int data;
public:
 Data(){}
 Data(int data)
 {
 this->data = data;
 }
};
ostream& operator<<(ostream& out, Data ob)
{
 out<<ob.data;
 return out;
}


template<typename T>
void myPrintAll(T a)
{
 cout<<a<<endl;
 cout<<"函数模板"<<endl;
}
int main(int argc, char *argv[])
{
 myPrintAll(10);
```

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

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
```

```
template<> void myPrintAll<Data>(Data a)
{
 cout<<a.data<<endl;
 cout<<"函数模板"<<endl;
}
int main(int argc, char *argv[])
{
 myPrintAll(10);
 myPrintAll('a');

 Data ob1(100);
 myPrintAll(ob1);

 return 0;
}
```

知识点4【类模板】（重要）

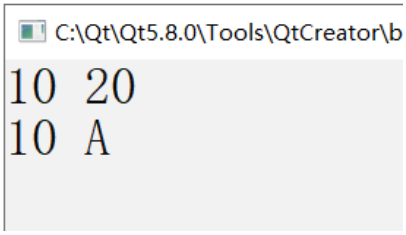
1、类模板的定义

类模板 将类中类型 抽象成虚拟类型。

```
//类模板
template<class T1, class T2>
class Data
{
private:
 T1 a;
 T2 b;
public:
 Data(){}
 Data(T1 a, T2 b)
 {
 this->a = a;
 this->b = b;
 }
 void showData()
 {
 cout<<a<<" "<<b<<endl;
 }
};
```

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();
}
```



类模板 的成员函数 在类外实现：

```
//类模板
template<class T1, class T2>
class Data
{
private:
 T1 a;
 T2 b;
public:
 Data(){}
  Data(T1 a, T2 b);
  void showData();
};
template<class T1, class T2>
Data<T1,T2>::Data(T1 a, T2 b)
{
 this->a = a;
 this->b = b;
}

template<class T1, class T2>
void Data<T1,T2>::showData()
{
 cout<<a<<" "<<b<<endl;
}
```

2、函数模板作为类模板的友元

```
//类模板
template<class T1, class T2>
class Data
```

```
{
 template<typename T3, typename T4>
 friend void myPrintData(Data<T3, T4> &ob);
private:
 T1 a;
 T2 b;
public:
 Data(){}
 Data(T1 a, T2 b);
 void showData();
};

//函数模板
template<typename T3, typename T4> void myPrintData(Data<T3, T4> &ob)
{
 cout<<ob.a<<" "<<ob.b<<endl;
}
void test02()
{
 Data<int, char> ob1(100, 'A');
 myPrintData(ob1);
}
```

3、普通函数作为类模板的友元

```
//类模板
template<class T1, class T2>
class Data
{
 friend void myPrintData(Data<int, char> &ob);
private:
 T1 a;
 T2 b;
public:
 Data(){}
 Data(T1 a, T2 b);
 void showData();
};

void myPrintData(Data<int, char> &ob)
```

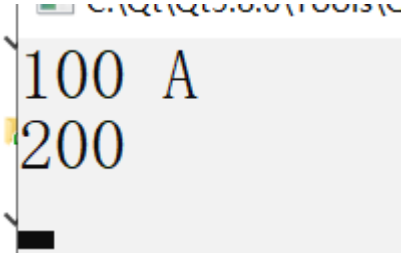
```
{
 cout<<ob.a<<" "<<ob.b<<endl;
}
void test02()
{
 Data<int, char> ob1(100, 'A');
 myPrintData(ob1);
}
```

知识点5【类模板的继承】（重要）

1、类模板 派生处 普通类

```
#include <iostream>

using namespace std;
//类模板
template<class T1, class T2>
class Base
{
private:
 T1 a;
 T2 b;
public:
 Base(){}
 Base(T1 a, T2 b);
 void showData();
};
template<class T1, class T2>
Base<T1,T2>::Base(T1 a, T2 b)
{
 this->a = a;
 this->b = b;
}

template<class T1, class T2>
void Base<T1,T2>::showData()
{
 cout<<a<<" "<<b<<endl;
}

//类模板派生处普通类
```

```
class Son1:public Base<int,char>
{
public:
 int c;
public:
 Son1(int a, char b, int c):Base<int,char>(a, b){
 this->c = c;
 }
};
int main(int argc, char *argv[])
{
 Son1 ob1(100, 'A', 200);
 ob1.showData();
 cout<<ob1.c<<endl;
 return 0;
}
```



2、类模板 派生处 类模板

```
#include <iostream>

using namespace std;
//类模板
template<class T1, class T2>
class Base
{
private:
 T1 a;
 T2 b;
public:
 Base(){}
 Base(T1 a, T2 b);
 void showData();
};
template<class T1, class T2>
```

```cpp
Base<T1,T2>::Base(T1 a, T2 b)
{
 this->a = a;
 this->b = b;
}

template<class T1, class T2>
void Base<T1,T2>::showData()
{
 cout<<a<<" "<<b<<endl;
}

//类模板派生处类模板
template<class T1, class T2, class T3>
class Son1:public Base<T1,T2>
{
public:
 T3 c;
public:
 Son1(T1 a, T2 b, T3 c):Base<T1,T2>(a, b){
 this->c = c;
 }
};
int main(int argc, char *argv[])
{
 Son1<int,char, int> ob1(100, 'A', 200);
 ob1.showData();
 cout<<ob1.c<<endl;
 return 0;
}

```

```
100 A
200
```

知识点6【类模板头文件 和源文件分离问题】（了解）

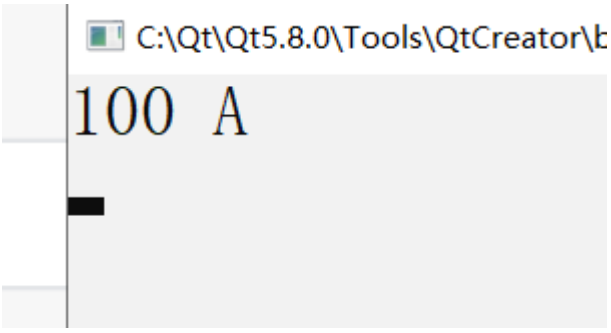
类模板 最好存储在.hpp中

data.hpp

```
#ifndef DATA_H
#define DATA_H
#include<iostream>
using namespace std;
template<class T1, class T2>
class Data
{
private:
 T1 a;
  T2 b;
 public:
  Data();
  Data(T1 a, T2 b);
  void showData(void);
 };

 template<class T1, class T2>
 Data<T1, T2>::Data()
 {
  cout<<"无参构造"<<endl;
 }
 template<class T1, class T2>
 Data<T1, T2>::Data(T1 a, T2 b)
 {
  this->a = a;
  this->b = b;
 }
 template<class T1, class T2>
 void Data<T1, T2>::showData(void)
 {
  cout<<a<<" "<<b<<endl;
 }
 #endif // DATA_H

```

main.cpp

```cpp
#include <iostream>
#include"data.hpp"
using namespace std;

int main(int argc, char *argv[])
{
 Data<int,char> ob1(100,'A');
 ob1.showData();
 return 0;
}

```



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

```cpp
#ifndef MYARRAY_HPP
#define MYARRAY_HPP
#include<iostream>
#include<string.h>
#include<string>
using namespace std;
template<class T>
class MyArray
{
 template<typename T1>
 friend ostream& operator<<(ostream& out, MyArray<T1> ob);
private:
 T *arr;//保存数组首元素地址
 int size;//大小
 int capacity;//容量
public:
```

```
 MyArray();
 MyArray(int capacity);
 MyArray(const MyArray &ob);
 ~MyArray();
 MyArray& operator=(MyArray &ob);

 void pushBack(T elem);
 void sortArray();
};

#endif // MYARRAY_HPP

template<class T>
MyArray<T>::MyArray()
{
 capacity = 0;
 size = 0;
 arr = NULL;
}

template<class T>
MyArray<T>::MyArray(int capacity)
{
 this->capacity = capacity;
 size = 0;
 arr = new T[this->capacity];
 memset(arr, 0,sizeof(T)*capacity);
}

template<class T>
MyArray<T>::MyArray(const MyArray &ob)
{
 if(ob.arr == NULL)
 {
 arr = NULL;
 size = 0;
 capacity=0;
 }
 else
 {
```

```
 capacity = ob.capacity;
 size = ob.size;
 arr = new T[capacity];
 memcpy(arr, ob.arr, sizeof(T)*capacity);
 }

}

template<class T>
MyArray<T>::~MyArray()
{
 if(arr != NULL)
 {
 delete [] arr;
 }
}

template<class T>
MyArray<T> &MyArray<T>::operator=(MyArray<T> &ob)
{
 //判断this->arr是否存在空间
 if(arr != NULL)
 {
 delete [] arr;
 arr=NULL;
 }

 size = ob.size;
 capacity = ob.capacity;
 arr = new T[capacity];
 memset(arr,0,sizeof(T)*capacity);
 memcpy(arr, ob.arr, sizeof(T)*capacity);
 return *this;
}

template<class T>
void MyArray<T>::pushBack(T elem)
{
 if(size==capacity)//满
 {
```

```
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;
```

```
  }
  }
  }
  }
  return;
}
template<typename T1>
ostream& operator<<(ostream& out, MyArray<T1> ob)
{
 int i=0;
 for(i=0;i<ob.size;i++)
 {
 out<<ob.arr[i]<<" ";

 }

 return out;
}
```

main.cpp

```
#include <iostream>
#include "myarray.hpp"
using namespace std;
class Person
{
 friend ostream& operator<<(ostream& out, Person ob);
private:
 int num;
 string name;
  float score;
public:
 Person(){}
 Person(int num,string name, float score)
 {
 this->num = num;
 this->name = name;
 this->score = score;
 }
 bool operator>(const Person &ob)
 {
 return num>ob.num;
```

```
 }
};
ostream& operator<<(ostream& out, Person ob)
{
 int i=0;

 out<<ob.num<<" "<<ob.name<<" "<<ob.score<<endl;
 return out;
}

int main(int argc, char *argv[])
{
 MyArray<int> ob1(5);
 ob1.pushBack(10);
 ob1.pushBack(30);
 ob1.pushBack(20);
 ob1.pushBack(50);
 ob1.pushBack(40);

 cout<<ob1<<endl;
 ob1.sortArray();
 cout<<ob1<<endl;

 MyArray<char> ob2(5);
 ob2.pushBack('A');
 ob2.pushBack('C');
 ob2.pushBack('D');
 ob2.pushBack('B');
 ob2.pushBack('F');

 cout<<ob2<<endl;
 ob2.sortArray();
 cout<<ob2<<endl;

 MyArray<Person> ob3;
 ob3.pushBack(Person(100,"lucy", 88.8f));
 ob3.pushBack(Person(103,"bob", 89.9f));
 ob3.pushBack(Person(105,"tom", 98.8f));
 ob3.pushBack(Person(102,"德玛", 99.8f));

 cout<<ob3<<endl;
```

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