## 实验六Templates

## 学号: 09017423 姓名：杨彬

### 实验目的

1．To use function templates to conveniently create a group of related.

2．To distinguish between function templates and function–template specializations,

class templates and class-template specializations.

### 实验内容

**Exp1:** quickSort(快速排序) is a fast sorting algorithm, which is widely applied in practice. It is used on the principle of divide-and-conquer. quickSort works by partitioning a given array A[p . . r] into two non-empty sub-arrays A[p . . q] and A[q+1 . . r] such that every element in A[p . . q] is less than or equal to every element in A[q+1 . . r]. Then the two sub-arrays are sorted by recursive calls to quickSort.

The details of quickSort are described as follows:

1. Choose a pivot value(基准). You may take the value of the first element as pivot value, but it can be any value, which is in range of sorted values, even if

it doesn't present in the array.

2. Partition(划分). Rearrange elements in such a way that all elements which are lesser than the pivot go to the left part of the array, and all elements greater than the pivot go to the right part of the array. Values equal to the pivot

can stay in any position of the array.

3. Sort both sub-arrays. Apply quicksort algorithm recursively to the left and

the right parts of the array.

Write a function template quickSort under the reference of the sort program of Fig.8.15, and also write a diver program that inputs, sorts and outputs an int array

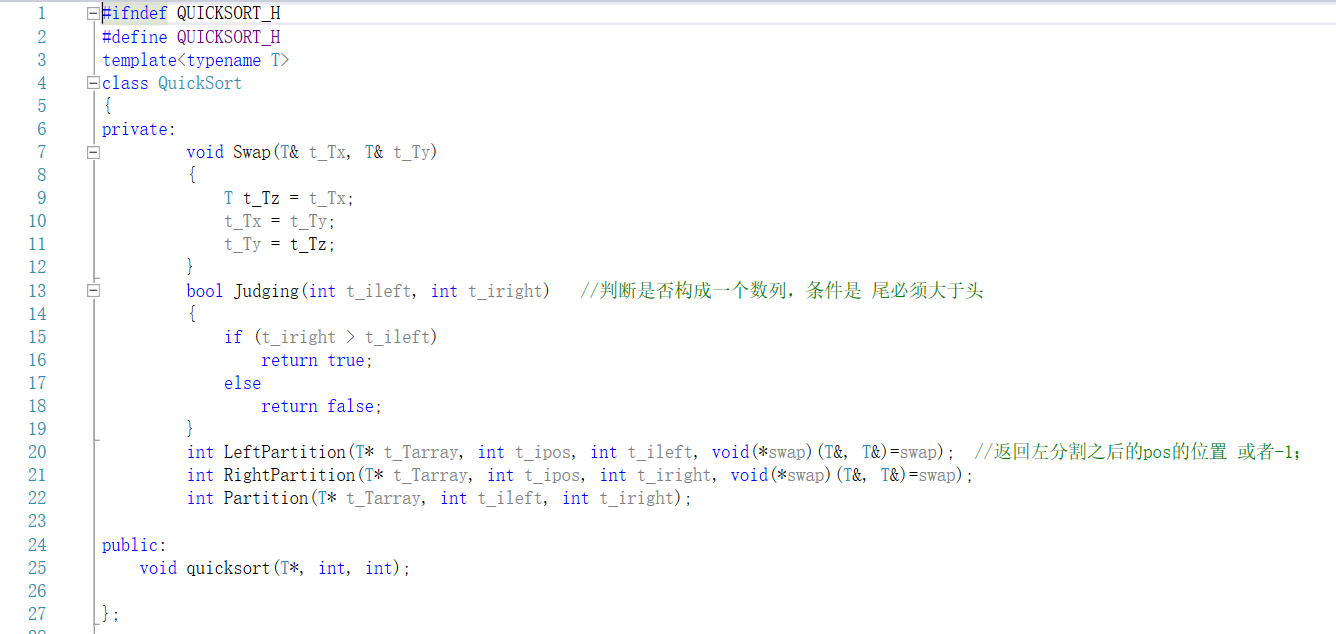
and a float array with 8 elements.

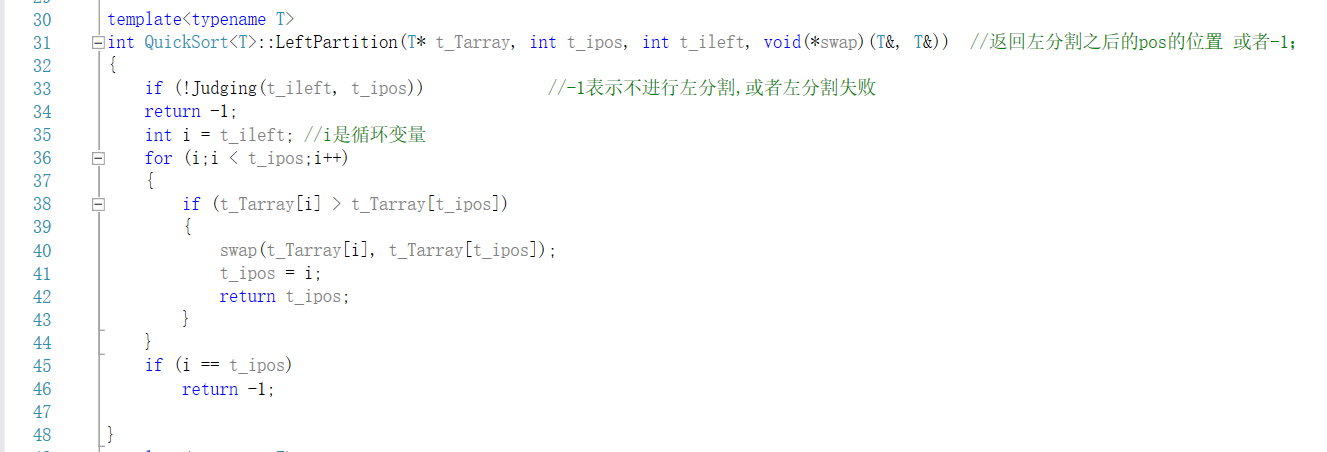
**Exp2:** **Problem Description**

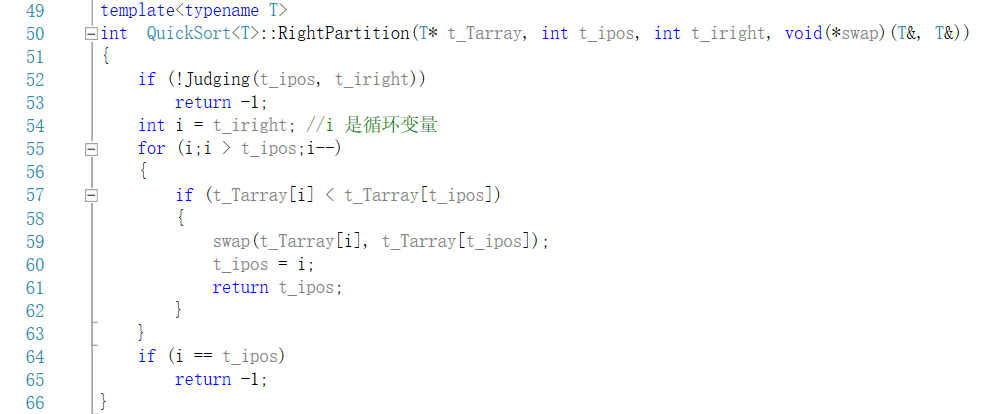
Use an int template nontype parameter numberOfElements and a type parameter elementType to help create a template for the Array class

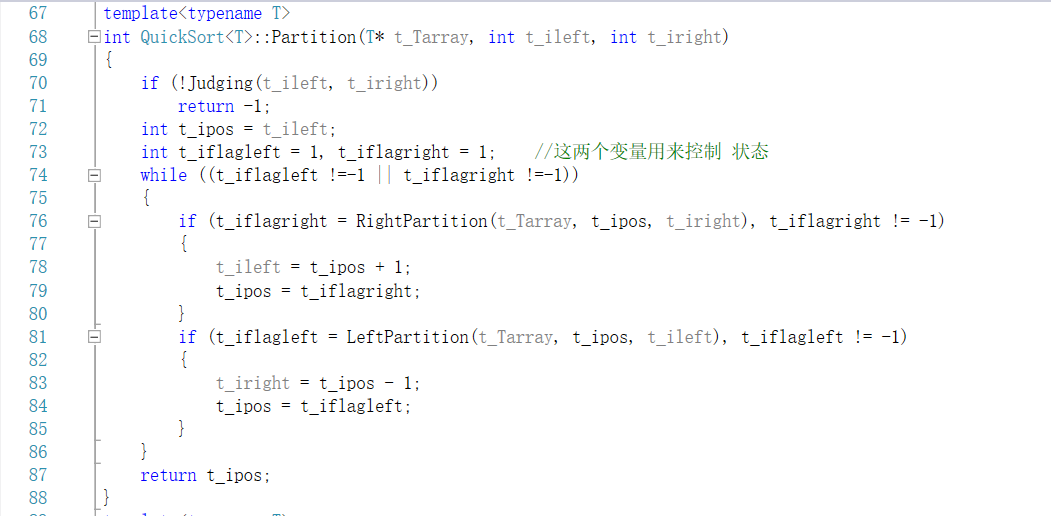
(Figs.11.6-11.7) we developed in Chapter 11. This template will enable Array objects to be instantiated with a specified number of elements of a specified element type at compile time.

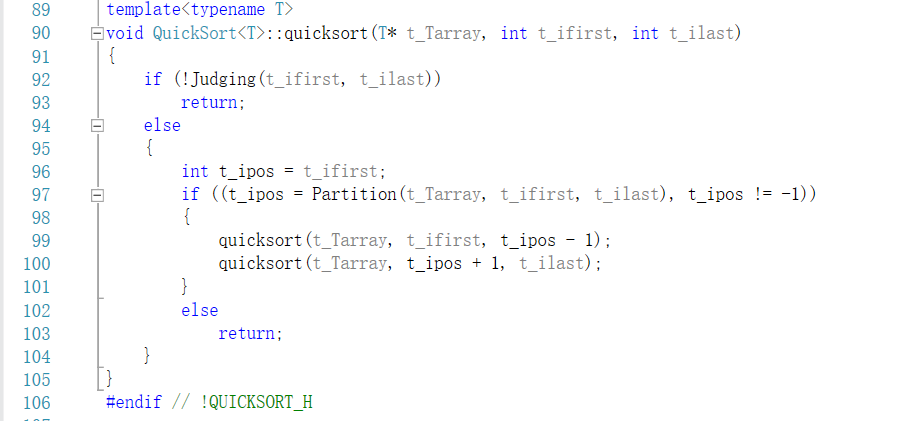
### 实验代码及结果：

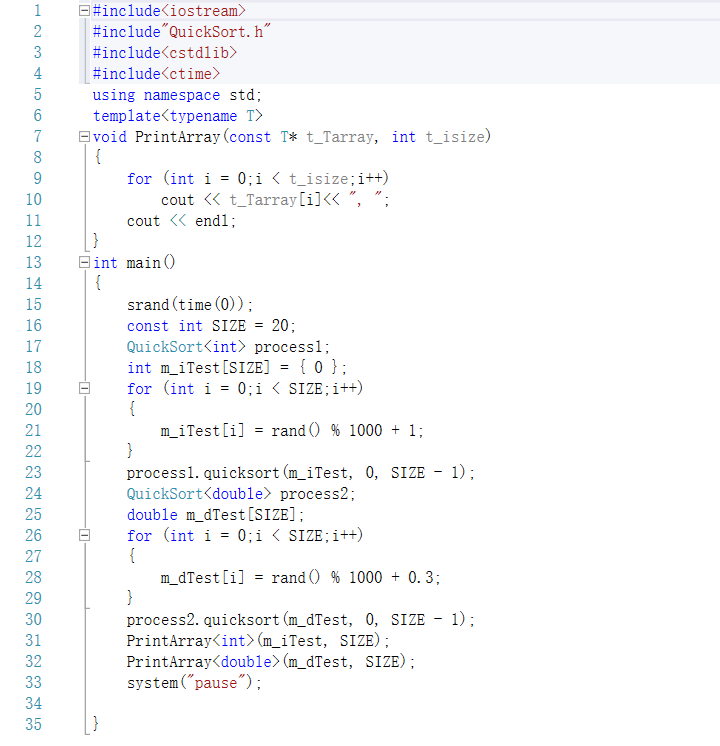
**Exp1:** 

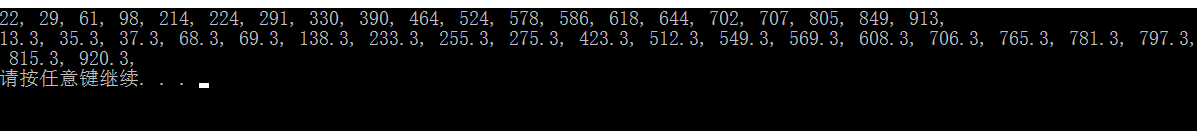




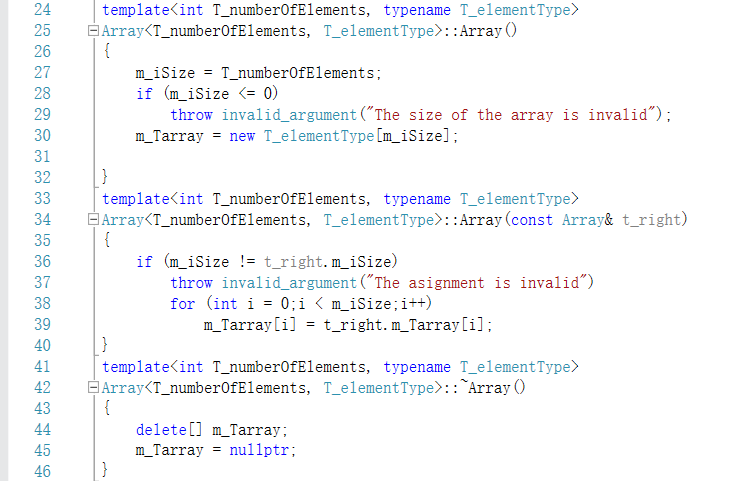


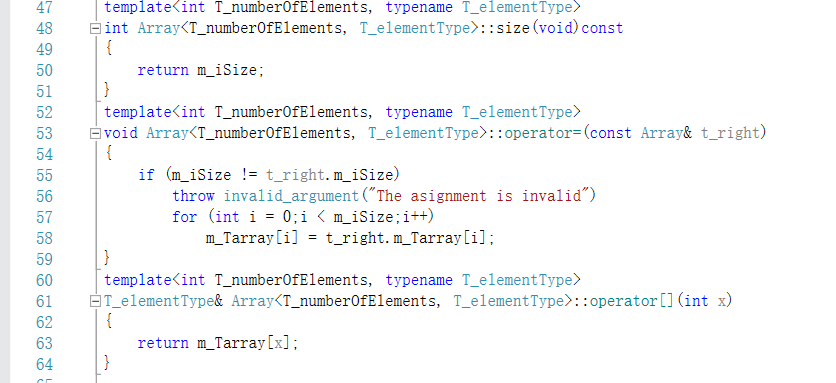


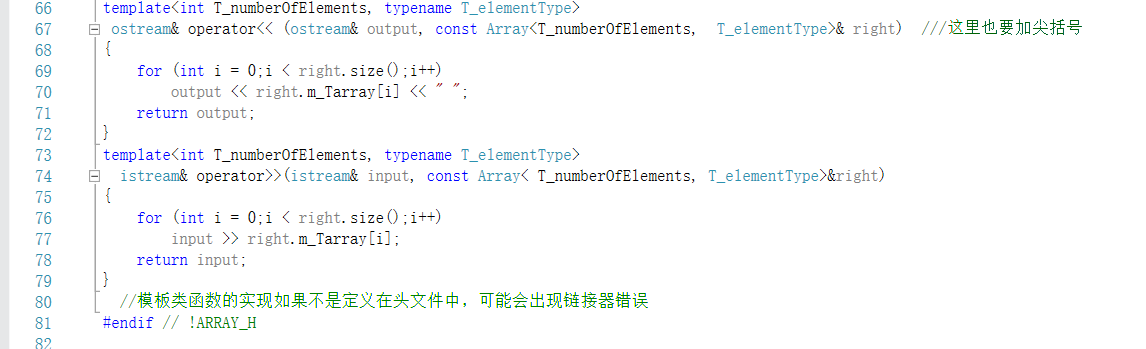


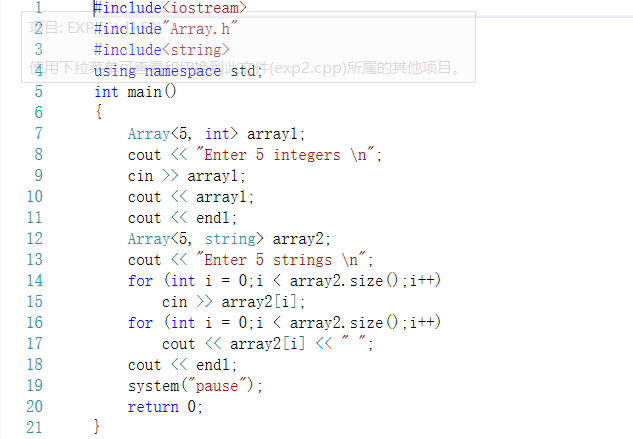


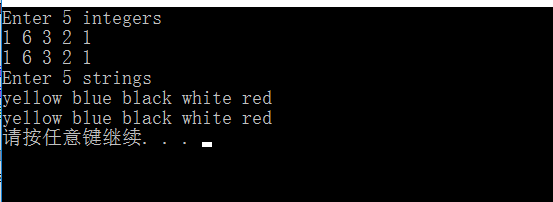
**Exp2:** 





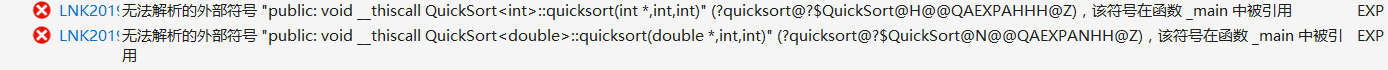


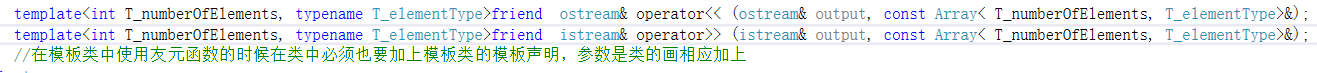
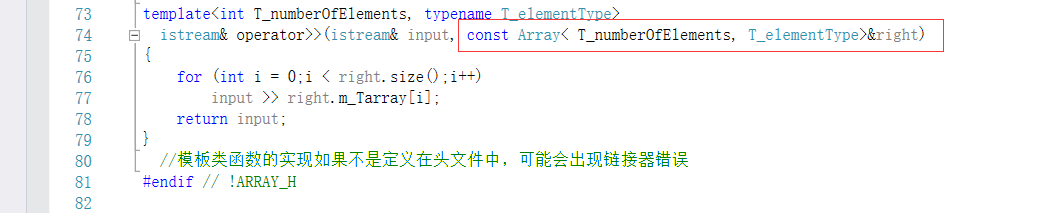




### 遇到的问题及解决过程

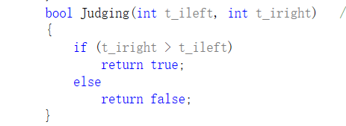
**Exp1:**1在使用类模板的时候在函数定义时发生编译错误，通过查阅课本以及网络，在每个函数定义之前加上template语句，并且将类名改成解决了问题

2.将类的头文件和类的实现分开，即分成.h文件和.cpp文件两个文件，但是对于类模板来说，编译的时候，出现了链接器错误。通过查阅网络，我将实现和头文件都放到了头文件当中，顺利地解决了问题。

**Exp2:**这里主要遇到的问题就是在模板类中使用友元函数，我按照模板类中非友元函数的写法来写友元函数，但是出现了编译错误，经过查阅相关的资料，我在友元函数声明的时候也加上template语句，改成 解决了问题

### 实验体会

**Exp1:**初次使用类模板的心得，使用类模板与使用一般的类不同，首先是在函数定义方面要注意其特殊的格式。第二因为类模板的特殊性，在编译的时候编译器并没有将类模板实例化，只有在运行的时候才会将类模板实例化，因此使用类模板很容易出现链接器错误。（上面提到的错误）为了解决这个问题，我找到两个方法第一将类模板的函数的实现也放在头文件中，二在类模板头文件中类的定义结束后加上include” .cpp”

2在实现快速排序的程序中，我第一次在一些关键的地方加入了一些判断函数，并且尝试让函数返回一些特殊的值来判断函数执行的状态（是否成功，发生了那种错误）等等 事实证明，在相对比较复杂的程序中加入这种判断错误的语句是非常必要的，它能够防止程序崩溃，并且通过一些特殊的返回值（通常是int）来方便我们分析产生错误的原因。

3对模块化编程更深层次的理解，现在我对于模块话编程的理解：将一个大的程序，划分成几个大模块，在将大的模块划分成更小的模块，（通常是函数）那么我们就将大的问题分成了小的问题，我们只需要考虑一个个小模块怎么实现，以及模块之间的接口即可。这样化大为小，不仅能提高我们解决问题的能力，同时还能优化代码。

**Exp2:1首先是在模板类中使用友元函数的格式于在普通的类中使用友元函数和在模板类中定义普通的函数有比较大的区别，应该注意区分。其次模板并不局限于类型，也可以是一些不确定的值，比如整数等等。Eg:这个实验**