一．模板

c++提供了函数模板(function template.)所谓函数模板，**实际上是建立一个通用函数，其函数类型和形参类型不具体制定，用一个虚拟的类型来代表。这个通用函数就成为函数模板。**

凡是函数体相同的函数都可以用这个模板代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现不同函数的功能。

* c++提供两种模板机制:**函数模板**和**类模板**

1. 函数模板

使用方法：

template<class/typename T>后面跟函数体，如：

template<class T> void swap(T &a, T &b)

{  
 T tmp = a;

a = b;

b = tmp;

}

1.自动类型推导

当调用 swap(a, b)时，函数模板根据a,b的类型自动推导T的类型，按照a,b的类型来替换T。

2.显式指定类型

调用 swap<int>(a,b)，在前面添加类型，显式地告诉函数模板，T的类型为int。

要注意的是，模板在调用时必须要指定出T的类型，这里说的指定指的是前面提到的两种方式：

1.给出参数，让模板自动类型推导

2.显式指定类型

三. 普通函数重载函数模板

Template<class T>

Void myPrint(T a, T b) {}

Void myPrint(int a, int b){}

1.当普通函数重载了同名函数模板时，调用同名函数时，优先似乎用普通函数调用。

2.如果想强制调用模板，那么可以使用空参数列表：  
myPrint<>(a,b)

3.函数模板可以产生函数模板重载

Template<class T>

Void myPrint(T a, T b) {}

Template<class T>

Void myPrint(T a, T b, T c) {}

下面的函数模板重载了上面的函数模板，多了一个参数。

4.如果函数模板可以产生更好的匹配，那么优先调用函数模板

因为，更好的匹配意味着更少的类型转换操作，节省更多资源。

四 模板实现机制

模板函数：通过模板生成的函数，叫做模板函数

1. 模板并不能通用所有的数据类型，比如自定义的类型。
2. 模板并不能直接调用，而是要生成模板函数后才能够调用。
3. 编译器会对函数模板进行两次编译，在函数模板声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。

五．模板的局限性

当我们定义了模板函数时，有些类型是无法通过这个模板来处理的。

我们可以通过具体化自定义数据类型，来解决这个问题：

格式如下：

Template<> void mySwap<Person>(Person &p1, Person &p2) {}

可以看到，具体化模板函数，在函数的定义中直接指定了参数类型。

如果具体化能够优先匹配，则选择具体化。

六．实例：实现通用数组排序模板

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

using namespace std;

template<class T>

void mySwap(T& a, T& b)

{

T tmp = a;

a = b;

b = tmp;

}

template<class T>

void mySort(T arr[], int len)

{

for (int i = 0; i < len; ++i)

{

int max = i;

for (int j = i + 1; j < len; ++j)

{

if (arr[max] < arr[j])

{

//交换最大值下标

max = j;

}

}

if (max != i)

{

//交换最大值数据

mySwap(arr[max], arr[i]);

}

}

}

//输出数组元素的模板

template<class T>

void printArray(T arr[], int len)

{

for (int i = 0; i < len; ++i)

{

cout << arr[i] << " ";

}

cout << endl;

}

void test()

{

char charArr[] = "hello";

int num = sizeof(charArr) / sizeof(char);

mySort(charArr, num);

printArray(charArr, num);

}

int main()

{

test();

return 0;

}