【必做题】

第 1 题

设计一个函数模板，其中包括数据成员 T a[n] 以及对其进行排序的成员函数 sort()，模板参数 T 可实例化成字符串。

解答

若将题中「模板参数 T 可实例化成字符串」理解为 T 实例化为**字符串类 string**，由于 string 类带有关系运算符「<」「>」的重载，函数模板可以设计为：

```
#include <iostream>
#include <string>
using namespace std;

template <class T>
void sort(T* a, int N)
{
    for (int i = 0; i < N - 1; i++)
    {
        for (int j = i + 1; j < N; j++)
        {
            if (a[i] > a[j])
            {
                T temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

这样，这个函数模板就既可以用于 string，也可以用于其他众多标准类型数据和重载了关系运算符「<」「>」的数据，适用范围很广。例如这个程序：

```
int main()
{
    /*  按字符串类 string 实例化  */
    string strs[4] = { "alpha", "beta", "gamma", "delta" };
    sort(strs, 4);
    for (int i = 0; i < 4; i++)
        cout << strs[i] << " ";
    cout << endl;
    /*  按整型数 int 实例化  */
    int nums[5] = { 114, 514, 19, 181, 10 };
    sort(nums, 5);
    for (int i = 0; i < 5; i++)
        cout << nums[i] << " ";
    cout << endl;
```

```
    return 0;
}
```

其输出为：

```
alpha beta delta gamma
10 19 114 181 514
```

若将「模板参数 T 可实例化成字符串」理解为 T 实例化为**字符指针 char ***，即 T a[n] 实例化为字符串（的首地址）的数组，则这个函数模板设计为：

```
#include <iostream>
using namespace std;

template <class T>
void sort(T* a, int N)
{
    for (int i = 0; i < N - 1; i++)
    {
        for (int j = i + 1; j < N; j++)
        {
            int k = 0;
            while (a[i][k] == a[j][k] && a[i][k] != '\0' && a[j][k] != '\0')
                k++;
            if (a[i][k] > a[j][k])
            {
                T temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

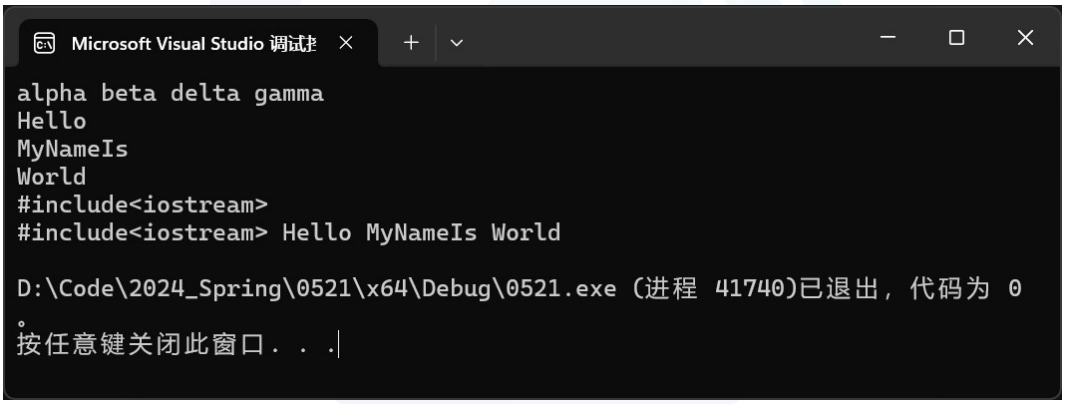
这个函数模板泛用性要差得多，只适用于 a 为可比较数据的二级指针的情况，即数据组的排序。例如：

```
int main()
{
    /*  按常字符串 const char* 实例化  */
    const char* strs[4] = { "alpha", "beta", "gamma", "delta" };
    sort(strs, 4);
    for (int i = 0; i < 4; i++)
```

```
        cout << strs[i] << " ";
    cout << endl;
    /*  按字符数组 char* 实例化  */
    char** varstrs = new char*[4];
    for (int i = 0; i < 4; i++)
    {
        varstrs[i] = new char[20];
        cin >> varstrs[i];
    }
    sort(varstrs, 4);
    for (int i = 0; i < 4; i++)
        cout << varstrs[i] << " ";
    cout << endl;
    for (int i = 0; i < 4; i++)
        delete[] varstrs[i];
    delete[] varstrs;
    return 0;
}
```

其输出为：

```
alpha beta delta gamma
Hello
MyNameIs
World
#include<iostream>
#include<iostream> Hello MyNameIs World
```



第 2 题

设计一个类模板，其中包括数据成员 T a[n] 以及在其中进行查找数据元素的函数 int search(T)，模板参数 T 可实例化成字符串。

解答

将题中「模板参数 T 可实例化成字符串」理解为 T 实例化为**字符串类 string**，由于 string 类带有关系运算符「==」的重载，类模板可以设计为：

```
#include <iostream>
#include <string>
using namespace std;

template <class T, int n>
class TArray
{
private:
    T a[n];
public:
    TArray()
    {
        for (int i = 0; i < n; i++)
            cin >> a[i];
    }
    int search(T);
};

template <class T, int n>
int TArray<T, n>::search(T target)
{
    int result = -1, count = 0;
    for (int i = 0; i < n; i++)
        if (a[i] == target)
        {
            count++;
            result = i;
        }
    if (count == 1) return result;
    else if (count == 0) return -1; // not found
    else return -2; // multiple found
}
```

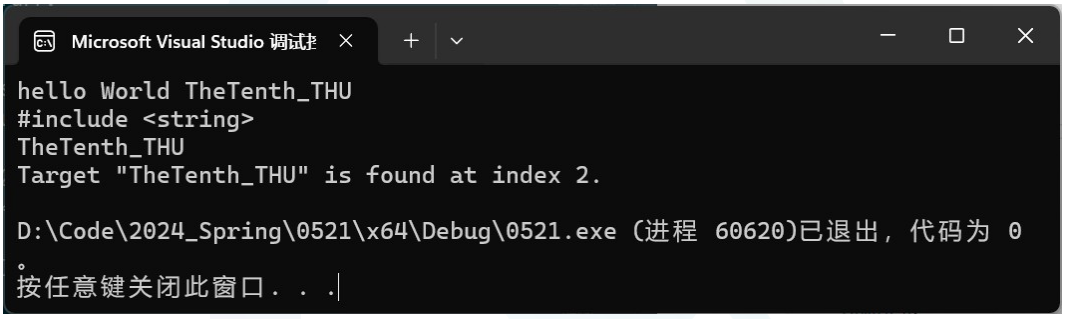
同样，这个类模板既可以用于 string，也可以用于其他众多标准类型数据和重载了关系运算符「==」的数据，适用范围很广。例如：

```
int main()
{
    TArray<string, 5> strarr;
    string target;
    cin >> target;
    int result = strarr.search(target);
```

```
    if (result == -1) cout << "Target \"" << target <<"\" is not found." << endl;
    else if (result == -2) cout << "Multiple \"" << target << "\" targets is found." <<
        endl;
    else cout << "Target \"" << target << "\" is found at index " << result << "." <<
        endl;
    return 0;
}
```

运行的结果为：

```
hello World TheTenth_THU
#include <string>
TheTenth_THU
Target "TheTenth_THU" is found at index 2.
```



【选做题】

第 3 题

参考课件中「类模版派生实例代码」，用「模板类派生新的派生类」的方法（即 p18 第 2 种派生方式），重新改写下面（p19～21）的 154 行程序代码：（输出提示语因编译器编码问题尚未解决已改为英文）

```
#include <iostream>
using namespace std;

/* Node structure */
class SNode
{
public:
    SNode(int value);
    ~SNode() {};
    int m_value;
    SNode* m_next;
};

/* Class template for linked list */
template <class Type>
class TList
{
public:
    TList();
    ~TList();
    // Insert a new node with value
    virtual bool Insert(Type value);
    // Delete the first node with value
    bool Delete(Type value);
    // Check if the list contains a node with value
    bool Contain(Type value);
    // Print all the nodes in the list
    void Print();
protected:
    SNode* m_head;
};

/* Class template for set */
template <class Type>
class TSet : public TList<Type>
{
public:
    // Overload for TSet: Insert a new node with value, check if it already exists first
    bool Insert(Type value);
};

/* Definition */
SNode::SNode(int value)
{
    m_value = value;
    m_next = NULL;
```

```
}
template <class Type>
TList<Type>::TList()
{
    m_head = NULL;
}
template <class Type>
TList<Type>::~TList() // Destructor to delete all nodes still in the list
{
    SNode* p = m_head;
    for (; p != NULL; )
    {
        m_head = p->m_next;
        delete p;
        p = m_head;
    }
}
template <class Type>
bool TList<Type>::Insert(Type value)
{
    SNode* pTemp = new SNode(value);
    if (pTemp == NULL) return false; // Memory allocation failed
    pTemp->m_next = m_head;
    m_head = pTemp;
    return true;
}
template <class Type>
bool TList<Type>::Delete(Type value)
{
    SNode* p1, * p2;
    if (m_head->m_value == value)
    {
        p1 = m_head->m_next;
        delete m_head;
        m_head = p1;
        return true;
    }
    else
    {
        for (p1 = m_head, p2 = m_head->m_next; p2 != NULL; ) // Traverse the entire list
        {
            if (p2->m_value == value)
            {
                p1->m_next = p2->m_next;
                delete p2;
                return true;
            }
            else
            {
                p1 = p1->m_next;
                p2 = p2->m_next;
            }
        }
    }
    return false; // Value not found in the list
```

```
}
template <class Type>
bool TList<Type>::Contain(Type value)
{
    for (SNode* p = m_head; p != NULL; p = p->m_next)
    {
        if (p->m_value == value) return true;
    }
    return false;
}
template <class Type>
void TList<Type>::Print()
{
    cout << "The values of the nodes are:";
    for (SNode* p = m_head; p != NULL; p = p->m_next)
    {
        cout << " " << p->m_value << "; ";
    }
    cout << endl;
}
template <class Type>
bool TSet<Type>::Insert(Type value)
{
    if (!(TList<Type>::Contain(value)) && TList<Type>::Insert(value))
        return true; // Value not in the list , insert it successfully
    return false; // Value already exists in the list or insert failed for other reasons
}

// Test code
void main()
{
    TList<int> sIntList;
    sIntList.Insert(12);
    sIntList.Insert(24);
    sIntList.Insert(48);
    sIntList.Insert(96);
    sIntList.Insert(24); // Insert a duplicate value 24
    sIntList.Print();
    sIntList.Delete(24); // Delete one of the duplicate nodes with value 24
    sIntList.Print();

    TSet<int> sIntSet;
    sIntSet.Insert(12);
    sIntSet.Insert(24);
    sIntSet.Insert(48);
    sIntSet.Insert(96);
    sIntSet.Insert(24); // Insert a duplicate value 24
    sIntSet.Print();
    sIntSet.Delete(24); // Delete one of the duplicate nodes with value 24
    sIntSet.Print();

    cin.get(); // Pause the console window to see the output
}
```

另外，链表插入要求是在指定的结点之后插入新结点。结点指定是指输入结点序号值。例如，在 3 号结点之后插入等。

解答

改用「模板类派生新的派生类」的方法，就是把用 int 类型实例化的 TList 类模板作为基类去派生集合类，此时集合类还使用 Type 虚拟类型已无意义，故直接改名为 intSet。得到程序如下，其中主要修改在第 27、54～75、135～140 行：

```
#include <iostream>
using namespace std;

/* Node structure */
class SNode
{
public:
    SNode(int value);
    ~SNode() {};
    int m_value;
    SNode* m_next;
};
SNode::SNode(int value)
{
    m_value = value;
    m_next = NULL;
}

/* Class template for linked list */
template <class Type>
class TList
{
public:
    TList();
    ~TList();
    // Insert a new node with value after the specified index (default is the head of the
        list)
    virtual bool Insert(Type value, int index = -1);
    // Delete the first node with value
    bool Delete(Type value);
    // Check if the list contains a node with value
    bool Contain(Type value);
    // Print all the nodes in the list
    void Print();
protected:
    SNode* m_head;
};
template <class Type>
TList<Type>::TList()
{
    m_head = NULL;
}
template <class Type>
TList<Type>::~TList() // Destructor to delete all nodes still in the list
```

```
{
    SNode* p = m_head;
    for (; p != NULL; )
    {
        m_head = p->m_next;
        delete p;
        p = m_head;
    }
}
template <class Type>
bool TList<Type>::Insert(Type value, int index)
{
    SNode* pTemp = new SNode(value);
    if (pTemp == NULL) return false; // Memory allocation failed
    if (index == -1) // Insert at the head of the list
    {
        pTemp->m_next = m_head;
        m_head = pTemp;
        return true;
    }
    for (SNode* p = m_head; p != NULL; p = p->m_next)
    {
        index--;
        if (index == -1) // Insert at the specified index
        {
            pTemp->m_next = p->m_next;
            p->m_next = pTemp;
            return true;
        }
    }
    return false; // Given index is out of range
}
template <class Type>
bool TList<Type>::Delete(Type value)
{
    SNode* p1, * p2;
    if (m_head->m_value == value)
    {
        p1 = m_head->m_next;
        delete m_head;
        m_head = p1;
        return true;
    }
    else
    {
        for (p1 = m_head, p2 = m_head->m_next; p2 != NULL; ) // Traverse the entire list
        {
            if (p2->m_value == value)
            {
                p1->m_next = p2->m_next;
                delete p2;
                return true;
            }
            else
            {
```

```
                p1 = p1->m_next;
                p2 = p2->m_next;
            }
        }
    }
    return false; // Value not found in the list
}
template <class Type>
bool TList<Type>::Contain(Type value)
{
    for (SNode* p = m_head; p != NULL; p = p->m_next)
    {
        if (p->m_value == value) return true;
    }
    return false;
}
template <class Type>
void TList<Type>::Print()
{
    cout << "The values of the nodes are:";
    for (SNode* p = m_head; p != NULL; p = p->m_next)
    {
        cout << " " << p->m_value << "; ";
    }
    cout << endl;
}

/* Class template for set of integers */
template <class Type>
class intSet : public TList<int>
{
public:
    // Overload for intSet: Insert a new node with value, check if it already exists first
    bool Insert(int value, int index = -1);
};
template <class Type>
bool intSet<Type>::Insert(int value, int index)
{
    if (!(TList<int>::Contain(value)) && TList<int>::Insert(value, index))
        return true; // Value not in the list, insert it successfully
    return false; // Value already exists in the list or insert failed for other reasons
}

// Test code
void main()
{
    TList<int> sIntList;
    sIntList.Insert(12); // Insert a new node with value 12 at the head of the list
    sIntList.Insert(24, 0); // Insert a new node with value 24 after the head of the list
    sIntList.Insert(48); // Insert a new node with value 48 at the head of the list
    sIntList.Insert(96, 2); // Insert a new node with value 96 after the second node
    sIntList.Insert(24, 1); // Insert a duplicate value 24 after the first node
    sIntList.Print();
    sIntList.Delete(24); // Delete one of the duplicate nodes with value 24
```

```
153     sIntList.Print();
154
155     intSet<int> sIntSet;
156     sIntSet.Insert(12); // Insert a new node with value 12 at the head of the list
157     sIntSet.Insert(24, 0); // Insert a new node with value 24 after the head of the list
158     sIntSet.Insert(48); // Insert a new node with value 48 at the head of the list
159     sIntSet.Insert(96, 2); // Insert a new node with value 96 after the second node
160     sIntSet.Insert(24, 1); // Insert a duplicate value 24 after the first node
161     sIntSet.Print();
162     sIntSet.Delete(24); // Delete one of the duplicate nodes with value 24
163     sIntSet.Print();
164
165     cin.get(); // Pause the console window to see the output
166 }
```

程序运行结果为：

```
The values of the nodes are: 48;  12;  24;  24;  96;
The values of the nodes are: 48;  12;  24;  96;
The values of the nodes are: 48;  12;  24;  96;
The values of the nodes are: 48;  12;  96;
```