# The Standard Template Library (STL)

## 1. Introduction to the STL (17)

The Standard Template Library is an extensive collection of generic templates for useful data structures and algorithms. Most of the templates in the STL can be grouped into the following categories:

- **Containers**: Class templates for objects that store and organize data.
- **Iterators**: Class templates for objects that behave like pointers, and are used to access the individual data elements in a container.
- **Algorithms**: Function templates that perform various operations on elements of containers.

There are two types of container classes: *sequence* and *associative*.

Here are some sequence containers:

- array — A fixed-size container that is similar to an array
- deque — A double-ended queue. Like a vector but designed so that values can be quickly added to or removed from both the front and back.
- forward_list — A singly linked list of data elements. Values may be inserted to or removed from any position.
- list — A doubly linked list of data elements. Values may be inserted to or removed from any position.
- vector — A container that works like an expandable array. It automatically adjusts its size to accommodate the number of elements it contains.

Here are some associative containers:

- set — Stores a set of unique values that are sorted. Duplicates are not allowed.
- multiset — Stores a set of unique values that are sorted. Duplicates are allowed.
- map — Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed. The elements are sorted in order of their keys.
- multimap — Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed. The elements are sorted in order of their keys.
- unordered_set — Like a set, except that the elements are not sorted.
- unordered_multiset — Like a multiset, except that the elements are not sorted.
- unordered_map — Like a map, except that the elements are not sorted.
- unordered_multimap — Like a multimap, except that the elements are not sorted.

Here are some container adapter classes:

 

- stack — An adapter class that stores elements in a deque. A stack is a last-in, first-out (LIFO) container (like a stack of plates in the cafeteria).
- queue — An adapter class that stores elements in a deque. A queue is a fist-in, first-out (FIFO) container (like people lining up for service).
- priority_queue — An adapter class that stores elements in a vector. The element that you retrieve is always the one with the greatest value.

To use the container and container adapter classes, you need to include the necessary header file.

| Classes | Header File |
|---|---|
| array | <array> |
| deque | <deque> |
| forward_list | <forward_list> |
| list | <list> |
| map, multimap | <map> |
| queue, priority_queue | <queue> |
| set, multiset | <set> |
| stack | <stack> |
| vector | <vector> |

## 2. The array Class *(17.2)*

The simplest container in the STL is the array class. An array object works very much like a regular array. It is a fixed-size container that holds elements of the same data type. In fact, an array object uses a traditional array internally to store its elements.

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

int main() {
    // declare and initialize an array container of type string with size 6
    // names is of type array for holding strings
    array<string, 6> names = { "Jack", "Jill", "Mary", "Tom" };
    names[4] = "Dr Hwang";
    for (int i = 0; i < names.size(); i++) {
        cout << names[i] << endl;
    }
}
```

 

## 3. The queue Class

A queue is a fist-in, first-out (FIFO) container (like people lining up for service). You can only access the front element of the queue. Some member functions are:

- **push**(element) – adds an element to the end of the queue and increases the size of the queue by one.
- **pop**() – removes the front element from the queue and reduces size of the queue by one.
- **front**() – returns the element at the front of the queue.
- **empty**() – returns true if the queue is empty.

In the next example, note the order in which the elements are pushed and popped from the queue.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    // a queue for storing strings
    queue<string> q;
    q.push("Jack");       // front of queue
    q.push("Jill");
    q.push("Mary");
    q.push("Tom");        // back of queue

    cout << "Contents of queue:" << endl;
    while (!q.empty()) {
        cout << q.front() << endl;  // get element at the front of the queue
        q.pop();
    }
}
```

```
Contents of queue:
Jack
Jill
Mary
Tom
```

## 4. The stack Class

A stack is a last-in, first-out (LIFO) container (like a stack of plates in the cafeteria). You can only access the top element of the stack. Some member functions are:

- **push**(element) – adds an element to the top of the stack and increases the size of the stack by one.
- **pop**() – removes the top element from the stack and reduces size of the stack by one.
- **top**() – returns the element at the top of the stack.
- **empty**() – returns true if the stack is empty.

 

In the next example, note the order in which the elements are pushed and popped from the stack.

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
    // a stack for storing ints
    stack<int> s;
    s.push(4);      // bottom of stack
    s.push(3);
    s.push(2);
    s.push(1);      // top of stack

    cout << "Contents of stack:" << endl;
    while (!s.empty()) {
        cout << s.top() << endl;    // get element at the top of the stack
        s.pop();
    }
}
```

```
Contents of stack:
1
2
3
4
```

## 5. The map class

Each element in a map has two parts: a key and a value. Map elements are commonly referred to as *key-value pairs*. Keys must be unique. The elements are sorted.

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

int main() {
    map<int, string> employees;     // a key-value pair; key is int and value is
string
    employees[110] = "Beth";                        // adding 110
    employees[326] = "Jake";                        // adding 326
    employees[110] = "Emily";                       // replacing 110
    employees.emplace(210, "Chris");                // adding 210
    employees.emplace(482, "Mary");                 // adding 210

    cout << "Value for 482 is " << employees.at(482) << endl;

    employees.erase(482);                           // deleting 482

    // using an iterator to print out all the pairs
    for (auto element : employees) {
        cout << element.first << " " << element.second << endl;
    }
    cout << endl;
}
```

```
Value for 482 is Mary
110 Emily
210 Chris
326 Jake
```

## 6. Iterators

The following example shows the creation and insertion for a vector. Also shows how **iterators** are used.

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

int main() {
    vector<int> data(10); // declare a vector of size 10
    // store the values 0 to 9 in the vector
    for (int i = 0; i < 10; i++) {
        data[i] = i;
    }

    // using an iterator to print out the 10 values
    vector<int>::iterator it;
    for (it = data.begin(); it < data.end(); it++) {
        cout << *it << " ";
```

 

```
    }
    cout << endl;

    // insert element after the 2nd element
    it = data.begin() + 2;
    data.insert(it, 88);
    cout << "Vector size is " << data.size() << endl;

    // another way to print out the 10 values
    for (auto element : data) {
        cout << element << " ";
    }
    cout << endl;
}
```

```
0 1 2 3 4 5 6 7 8 9
Vector size is 11
0 1 88 2 3 4 5 6 7 8 9
```

## 7. Algorithms

Many commonly used algorithms are written as function templates in STL such as sorting, searching, partition, permutation, heap, and many others.

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

int main() {
    vector<int> data = { 10,1,9,2,8,3,7,4,6,5 };


    // insert element after the 2nd element
    vector<int>::iterator it;
    it = data.begin() + 2;
    data.insert(it, 88);
    cout << "Vector size is " << data.size() << endl;

    // using an iterator to print out the 10 values
    for (it = data.begin(); it < data.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;

    sort(data.begin(), data.end());

    // another way to print out the 10 values
    for (auto element : data) {
        cout << element << " ";
    }
    cout << endl;


    int searchValue = 8;
    if (binary_search(data.begin(), data.end(), searchValue)) {
```

```
        cout << "Value found" << endl;
    }
    else {
        cout << "Value not found" << endl;
    }
}
```

```
Vector size is 11
10 1 88 9 2 8 3 7 4 6 5
1 2 3 4 5 6 7 8 9 10 88
Value found
```

## 8. *Problems* (Problems with an asterisk are more difficult)

1. Write a program