

# **Chapter 16**

## **Introduction to the Standard Template Library**

## Standard Template Library (STL)

We have looked at and implemented the stack and queue ADT. These structures are well suited for many software applications. Because of the constant need for these data structures it made sense to create a library of these useful functions. Developed at Hewlett-Packard by Alexander Stepanov and Meng Lee, the STL is a collection of libraries containing data structures and algorithms written in C++. They are not part of the original C++ language so older compilers may not support their use. The standard for C++ has now been finalized and the STL is part of the standard C++ libraries.

The STL is a collection of template classes with a type parameter used to represent the type of the data to be stored. These classes are often referred to as *container classes*. The STL is organized into *containers*, *iterators*, and *algorithms*.

**Containers** - A group of class templates that provide standardized, generic structures for storing data. Containers hold a collection of objects of a specific type.

The data that can be stored in these containers has been left unspecified and these containers are often referred to as abstract containers. Abstract containers rely heavily on templates.

There are two basic types of containers; *sequential containers* and *associative containers*. Like an array, sequence containers organize data in a sequential fashion. Associative containers organize their data using keys which allow elements stored in the container to be accessed randomly. This random access is more rapid than sequential access. Sequential containers include vector, list, stack, queue, deque, and priority\_queue. Associative containers include map, multimap, set and multiset. Unlike items contained in the standard libraries such as <iostream> and <iomanip> that work on specific types, these classes and algorithms are *generic* (template classes and template functions).

There are constraints on the types that can be held by a container. The element type must support assignment and we must also be able to copy objects of the element type. Containers of IO types are invalid as they do not support assignment or copying. Containers of references are also invalid.

**Iterators** - A means of accessing elements in the container including finding the predecessor and successor of an element. Iterators provide a way in which the algorithms can act upon the containers. The following iterator operations are supported for all of the library containers.

- \*iter - Returns a reference to the element referenced by iterator iter.
- iter-mem - Dereference iter and access the member (mem) from the element.
- ++iter  
iter++ - Increment iter (move to the next element).
- iter  
iter-- - Decrement iter (move to the previous element).
- iter1 == iter2 - Compare two iterators for equality or inequality. Two iterators are equal if they both reference the same element in the same container or if they are positioned "one past the end" of the container. An iterator positioned "one past the end" is sometimes called the **off-the-end** iterator.

**Algorithms** - A set of function templates that provide functions used to perform common operations such as sorting and searching container objects.

If you do an internet search for **C++ STL** you will find many sites with information, sample programs etcetera.

## **vector<T>**

A vector is similar to an array in that

- it holds a sequence of elements
- its elements are stored contiguously in memory
- the elements may be accessed using the subscript operator [ ]

A vector has some advantages over an array. A vector

- does not have to have its number of elements declared
- can tell you how many elements it contains
- will increase in size if you add an element when it is already full

A vector may be thought of as an array that can change size (shrink or expand). It stores a collection of values of its base type. To use the vector class you must

```
#include <vector>
```

in your program. The .h extension is not valid so you must always put the statement

```
using namespace std;
```

in your program.

Sample Declarations:

```
vector <float> fVals;           // declares fVals as a vector of floating point values
```

```
vector <int> iVals(20);         // declares iVals as a vector of integers and gives a starting  
                               // size of 20 – this is only a starting size, it will expand if  
                               // you add more than 20 elements
```

```
vector <int> iVals(20, 5);      // declares iVals as a vector of integers, gives a starting size  
                               // of 20 and stores 5 in each location
```

```
vector <int> iVals2(iVals);     // declares iVals2 as a vector in integers and all values in  
                               // iVals are copied to iVals2
```

If the constructor with the integer argument is used, the vector is initialized to zero for numeric types. If the vector components are class objects, the default constructor for the class is called.

Sample Program using vector with standard array notation:

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

int main()
{
    vector<string> stuName(5);
    vector<float> stuGpa(5);

    for(int i=0; i<5; i++)
    {
        cout << "Student " << i+1 << " name: ";
        getline(cin,stuName[i]);
        cout << "GPA: ";
        cin >> stuGpa[i];
        cin.ignore( );
    }

    cout << "\n\n";
    for(int i=0; i<5; i++)
    {
        cout << stuName[i] << setw(20-stuName[i].length()) << stuGpa[i] << endl;
    }
    return 0;
}
```

Sample Run:

```
Student 1 name: Joe Jones
GPA: 2.5
Student 2 name: Sally Smart
GPA: 4.0
Student 3 name: Fred Failure
GPA: 1.2
Student 4 name: Betty Bstudent
GPA: 3.0
Student 5 name: Larry Lazy
GPA: 0.5
```

```
Joe Jones          2.5
Sally Smart        4
Fred Failure       1.2
Betty Bstudent     3
Larry Lazy         0.5
```

Because a starting size was specified (that many elements exist), we can use the array subscript operator [ ] to store and retrieve elements in the vector.

Comment on the following:

```
stuName[6] = "Sam Student";
```

### Increasing the size of a vector:

To add elements to a vector (push an element onto the back of the vector increasing the total number of elements) use the *push\_back* member function.

```
objectName.push_back(value);
```

As the size is changing, it is useful to use the *size* member function to report the current size of the vector.

```
objectName.size( ); // size( ) function returns an unsigned int
```

You can also check the *capacity* (amount of memory currently allocated for the vector) of a vector. The capacity and size are not necessarily the same. The relationship

```
capacity >= size
```

is always true.

```
objectName.capacity( ); // returns the number of elements for which there is memory  
// allocated
```

If you wish to have more control over the memory allocated for a vector you can use the member functions *reserve*, and *resize*.

```
objectName.reserve(50); // sets the capacity to at least 50 elements
```

```
objectName.reserve(objectName.size( ) + 20); // sets the capacity to 20 elements  
// more than the current size
```

The *resize( )* function changes the actual size of the vector.

```
objectName.resize(15); // resizes the vector to 15 elements
```

If the size of the vector was greater than 15, only the first 15 elements are retained. If the size was smaller than 15, the new elements are initialized as described previously.

The following program demonstrates the use of the `push_back()` function.

```
int main()
{
    vector<string> stuName(5);
    vector<float> stuGpa(5);
    char resp;
    string tempNm;
    float tempGpa;

    for(int i=0; i<5; i++)
    {
        cout << "Student " << i+1 << " name: ";
        getline(cin,stuName[i]);
        cout << "GPA: ";
        cin >> stuGpa[i];
        cin.ignore( );
    }

    // increase the size of the vector
    cout << "Add another student (Y/N)? ";
    cin >> resp;
    while(toupper(resp) != 'N')
    {
        cin.ignore( );
        cout << "Student name: ";
        getline(cin,tempNm);
        cout << "GPA: ";
        cin >> tempGpa;
        stuName.push_back(tempNm);
        stuGpa.push_back(tempGpa);
        cout << "Add another student (Y/N)? ";
        cin >> resp;
    }

    cout << "\n\nThere are currently " << stuName.size() << " students.\n\n";
    for(unsigned int i=0; i<stuName.size(); i++)
    {
        cout << stuName[i] << setw(25-stuName[i].length()) << stuGpa[i] << endl;
    }

    return 0;
}
```

Output:

```
Student 1 name: Ted Talkative
GPA: 1.5
Student 2 name: Lucy Late
GPA: 2.0
Student 3 name: Greg Goodstudent
GPA: 3.75
Student 4 name: Sally Smart
GPA: 3.9
Student 5 name: Larry Lazy
GPA: 0.5
Add another student (Y/N)? y
Student name: Vera Verysmart
GPA: 4.0
Add another student (Y/N)? y
Student name: Tom Talented
GPA: 3.5
Add another student (Y/N)? n
```

There are currently 7 students.

The vector capacity is 10.

Ted Talkative	1.5
Lucy Late	2
Greg Goodstudent	3.75
Sally Smart	3.9
Larry Lazy	0.5
Vera Verysmart	4
Tom Talented	3.5

**IMPORTANT NOTE:** When elements are added to a vector or deque using an `insert` or `push` operation, the container may need to be relocated. For this reason, iterators may become invalid. This can occur even when the container is not relocated. When writing loops that add elements to a vector or deque, the program must ensure that the iterator is refreshed each trip through the loop. Don't store the value of the iterator returned from the `end()` function for later use. Update each time you wish to use it.

## Member Functions for vector<T>

- vector<Type> v; - construct a vector of capacity 0
- vector<Type> v(n); - construct a vector of capacity n, size n
- vector<Type> v(n, initVal); - construct a vector of capacity n, size n, and initialize each element to initVal
  
- v.empty() - returns true if v contains no values (size = 0)
- v.size() - returns the number of items currently in v
- v.capacity() - returns the number of items that can be stored in v
- v.reserve(n) - expand v so its capacity is n (size is not affected)
  
- v.push\_back(val); - add value to end of v
- v.pop\_back(); - remove last element of v
- v.front(); - returns a reference to the first element in v
- v.back(); - returns a reference to the last element in v

## Iterator Member Functions

- v.begin() - returns an iterator positioned at the first value in v
- v.end() - returns an iterator positioned immediately after the last value in v (one-past-the-end-of)
- v.rbegin() - returns a reverse iterator positioned at the last element in v
- v.rend() - returns a reverse iterator positioned one element before the first element in v
- v.insert(pos, val); - insert val into v at iterator position pos
- v.insert(pos, n, val); - insert n copies of val into v at iterator position pos
- v.erase(pos); - erase the value at iterator position pos from v
- v.erase(pos1, pos2) - erase the values in v from iterator position pos1 to pos2

## Iterator Operators

In addition to the operations outlined for all containers, the vector also supports

- iter + n - The addition or subtraction of an integral value to the current position will move the iterator to a new position in the vector.
- iter - n - The value must be a valid container address or one past the end.

## Array vs. Vector

### Reasons to use a vector:

-

-

-

### Reasons to use an array:

-

-

-

**If you don't need all of the features of a vector, use an array.**

## Algorithms in the STL (a partial list)

<code>count(beg, end, val)</code>	- return the number of times <code>val</code> occurs in the sequence
<code>accumulate(beg, end, init)</code>	- return the sum of the values in the sequence from <code>beg</code> to <code>end</code> ( <code>init</code> is the starting value for the accumulator)
<code>fill(beg, end, val);</code>	- places <code>val</code> in each location in the range <code>beg</code> to <code>end</code>
<code>find(beg, end, val)</code>	- return an iterator to <code>val</code> if it is present in the sequence otherwise return <code>end</code> (unsorted sequence)
<code>sort(beg, end);</code>	- sort the sequence in ascending order
<code>binary_search(beg, end, val)</code>	- return true if <code>val</code> is in the sequence (must be sorted) and false otherwise
<code>reverse(beg, end);</code>	- reverse the order of the values in the sequence
<code>random_shuffle(beg, end);</code>	- randomly shuffle the values in the sequence
<code>replace(beg, end, old, new)</code>	- replace the old value with the new value in the specified range
<code>max_element(beg, end)</code>	- return an iterator to the maximum value in the sequence
<code>min_element(beg, end)</code>	- return an iterator to the minimum value in the sequence
<code>for_each(beg, end, f);</code>	- applies function <code>f</code> to each element in the sequence (if <code>f</code> returns a value it is discarded)
<code>unique(beg, end);</code>	- replace any consecutive occurrences of the same value with only one instance of the value and returns the end of the smaller range of elements

You will need the following preprocessor directives in your program:

```
#include <vector>
#include <algorithm>
#include <numeric>
```

```

// This program demonstrates the use of an iterator
int main()
{
    vector<int> vals;
    vector<int> :: iterator p;
    int key;

    for(int i = 0; i < 10; ++i)
    {
        vals.push_back(i*2+1);
    }

    // output using an iterator
    for(p = vals.begin( ); p < vals.end( ); ++p)
    {
        cout << *p << endl;
    }

    // remove items using pop_back
    for(int i = 0; i < 3; ++i)
    {
        vals.pop_back();
    }

    cout << "\n\nAfter Pop_Back\n";
    for(p = vals.begin( ); p < vals.end( ); ++p)
    {
        cout << *p << endl;
    }
    return 0;
}

```

Name \_\_\_\_\_

### STL Lab

```
int main()
{
    // declare a vector of floating point values and an appropriate iterator

    float key;

    // fill the vector with 10 values from the keyboard

    // use an iterator to output the values

    // remove 3 items

    // output the values again

    // random shuffle the vector and output again

    // sort the vector and output again

    // read in a key and use the binary search algorithm

    return 0;
}
```

## Container Adaptors

We have been talking about Abstract Data Types since CS 1B. We have used the stack as a common example and indicated that it could be implemented in a variety of ways (array, vector, linked list etc.). Containers in the STL such as vector, list, or deque could be used to implement a stack. A stack is a less complex type than a queue or vector but these are appropriate as they provide a means to accomplish the basic stack operations. The interface should however be limited to operations appropriate for a stack. To ensure that only appropriate operations are provided we could make our own stack class containing a private member from the STL such as list, deque, or vector and supply a public interface that included only push, pop, top, and empty.

Because appropriate containers are supplied by the STL, the `stack` designation is called a container adaptor.

### The Stack Container Adaptor

Declarations:

<code>stack&lt;int&gt;</code>	A stack of integers using the default deque implementation.
<code>stack&lt;string, vector&lt;string&gt; &gt;</code>	A stack of strings using a vector implementation.
<code>stack&lt;float, list&lt;float&gt; &gt;</code>	A stack of floats using a list implementation.
<code>stack&lt;T, deque&lt;T&gt; &gt;</code>	

The stack adaptor provides functions called push, pop, top, empty, and size. For this reason, any container that supports the `push_back`, `pop_back`, `size`, `empty`, and `back` operations may be used.

```

#include <stack>
using namespace std;

// Stack adaptor illustration
int main()
{
    int data[10] =
    {
        45, 32, 18, 79, 12, 65, 57, 4, 16, 35
    };
    stack<int> s;
    cout << "\nCurrent size is " << s.size() << "\n\n";
    cout << "Push 6 elements\n";
    for(int i = 0; i < 6; ++i)
        s.push(data[i]);
    cout << "\nCurrent size is " << s.size() << "\n\n";
    cout << "Pop 3 elements\n";
    for(int i = 0; i < 3; ++i)
    {
        cout << setw(5) << s.top();
        s.pop();
    }
    cout << "\nCurrent size is " << s.size() << "\n\n";
    cout << "Push 4 elements\n";
    for(int i = 6; i < 10; ++i)
        s.push(data[i]);
    cout << "\nCurrent size is " << s.size() << "\n\n";
    cout << "Pop all elements\n";
    while(!s.empty())
    {
        cout << setw(5) << s.top();
        s.pop();
    }
    cout << "\nCurrent size is " << s.size() << "\n\n";
    return 0;
}

```

Current size is 0

Push 6 elements

Current size is 6

Pop 3 elements

65 12 79

Current size is 3

Push 4 elements

Current size is 7

Pop all elements

35 16 4 57 18 32 45

Current size is 0

## The Queue Container Adaptor

Recall, a queue is a FIFO (first in, first out) structure. The queue container adaptor can be applied to any container that supports `empty`, `size`, `front`, `back`, `push_back`, and `pop_front` (not supported by vector) operations. These are supported by `list` and `deque`. The queue adaptor provides operations `push`, `pop`, `empty`, `front`, `back`, and `size`.

Why was `pop_front` not supplied in vector?

## The Priority Queue Adaptor

The priority queue adaptor may be implemented using any container that supports `push_back`, `pop_back`, `front`, `empty`, and `size` operations. A comparison function object named `comp` is used for comparisons when retrieving the largest element. Both `vector` and `deque` supply the necessary operations.

Following are programs demonstrating the queue and priority queue adaptors. Study the output and compare it to the output provided in the previous example.

```

#include <queue>
#include <list>
using namespace std;

// queue adaptor using list illustration
int main()
{
    int data[10] =
    {
        45, 32, 18, 79, 12, 65, 57, 4, 16, 35
    };
    queue<int, list<int> > q;
    cout << "\nCurrent size is " << q.size() << "\n\n";
    cout << "Add 6 elements\n";
    for(int i = 0; i < 6; ++i)
        q.push(data[i]);
    cout << "\nCurrent size is " << q.size() << "\n\n";
    cout << "Remove 3 elements\n";
    for(int i = 0; i < 3; ++i)
    {
        cout << setw(5) << q.front(); // top is replaced with front
        q.pop();
    }
    cout << "\nCurrent size is " << q.size() << "\n\n";
    cout << "Add 4 elements\n";
    for(int i = 6; i < 10; ++i)
        q.push(data[i]);
    cout << "\nCurrent size is " << q.size() << "\n\n";
    cout << "Remove all elements\n";
    while(!q.empty())
    {
        cout << setw(5) << q.front();
        q.pop();
    }
    cout << "\nCurrent size is " << q.size() << "\n\n";
    return 0;
}

```

Current size is 0

Add 6 elements

Current size is 6

Remove 3 elements

45 32 18

Current size is 3

Add 4 elements

Current size is 7

Remove all elements

79 12 65 57 4 16 35

Current size is 0

```

#include <queue>
using namespace std;

// priority-queue adaptor illustration
int main()
{
    int data[10] =
    {
        45, 32, 18, 79, 12, 65, 57, 4, 16, 35
    };
    priority_queue<int> p;
    cout << "\nCurrent size is " << p.size() << "\n\n";
    cout << "Add 6 elements\n";
    for(int i = 0; i < 6; ++i)
        p.push(data[i]);
    cout << "\nCurrent size is " << p.size() << "\n\n";
    cout << "Remove 3 elements\n";
    for(int i = 0; i < 3; ++i)
    {
        cout << setw(5) << p.top();
        p.pop();
    }
    cout << "\nCurrent size is " << p.size() << "\n\n";
    cout << "Add 4 elements\n";
    for(int i = 6; i < 10; ++i)
        p.push(data[i]);
    cout << "\nCurrent size is " << p.size() << "\n\n";
    cout << "Remove all elements\n";
    while(!p.empty())
    {
        cout << setw(5) << p.top();
        p.pop();
    }
    cout << "\nCurrent size is " << p.size() << "\n\n";
    return 0;
}

```

Current size is 0

Add 6 elements

Current size is 6

Remove 3 elements

79 65 45

Current size is 3

Add 4 elements

Current size is 7

Remove all elements

57 35 32 18 16 12 4

Current size is 0