# C++ Standard Template Library: Container

Zhijiang Dong
Dept. of Computer Science
Middle Tennessee State University
Email: zdong@mtsu.edu

**Abstract**

This article gives a brief review of containers defined in C++ Standard Template Library, especially *vector* and *deque*. Not all member functions/data of *vector* and *deque* are covered.

## I. Container

Container is an abstraction of data structures that are used to hold objects of the same type. The Standard Template Library (STL) provides a number of predefined containers, most of which are generalize as templates to hold any type element. Class templates and function templates allow us to define a class or function and defer certain data-type information until we are ready to use the class. For more information about class templates and function templates, please refer to the hand outs I gave to you in the class. There are three major categories of containers – sequence containers, associative containers and container adapters.

### A. Sequence containers

The sequence containers represent linear data structures. Three sequence containers are *vector*, *deque*, and *list*.

- *vector* provides a dynamic array structure with fast random access to any element. You can think vector as a resizable array. whenever you want to insert a new item, the vector is "expanded" automatically if there is no more space. Inserting a item to anywhere but the end is expensive because of elements shift. Similarly to the deletion.
  Vectors are good at:
  - Accessing individual elements by their position index (constant time).
  - Iterating over the elements in any order (linear time).
  - Add and remove elements from its end (constant amortized time).

  Compared to arrays, they provide almost the same performance for these tasks, plus they have the ability to be easily resized. Although, they usually consume more memory than arrays when their capacity is handled automatically (this is in order to accomodate for extra storage space for future growth).
  Compared to the other base standard sequence containers (deques and lists), vectors are generally the most efficient in time for accessing elements and to add or remove elements from the end of the sequence. For operations that involve inserting or removing elements at positions other than the end, they perform worse than deques and lists, and have less consistent iterators and references than lists.
- *deque* also provides a dynamic array structure with random access and adds fast insertion and deletion of elements at front as well as from the back. Very slightly slower than vector because of an extra level of indirection.
  Deque sequences have the following properties:
  - Individual elements can be accessed by their position index.
  - Iteration over the elements can be performed in any order.
  - Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

  Therefore they provide a similar functionality as the one provided by vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence and not only at its end. On the drawback side, unlike vectors, deques are not granted to have all its elements in contiguous storage locations, eliminating thus the possibility of safe access through pointer arithmetics.
  Both vectors and deques provide thus a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors are very similar to a plain array that grows by reallocating all of its elements in a unique block when its capacity is exhausted, the elements of a deques can be divided in several chunks of storage, with the class keeping all this information and providing a uniform access to the elements. Therefore, deques are a little more complex internally, but this generally allows them to grow more efficiently than the vectors with their capacity managed automatically, specially in large sequences, because massive reallocations are avoided.
  For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists.

- *list* is usually implemented as a doubly linked list. There is no random access to the elements. Insertion and deletion anywhere is fast.
  Compared to other base standard sequence containers (vectors and deques), lists perform generally better in inserting, extracting and moving elements in any position within the container, and therefore also in algorithms that make intensive use of these, like sorting algorithms.
  The main drawback of lists compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a list one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).
  Storage is handled automatically by the class, allowing lists to be expanded and contracted as needed.

### B. Associative containers

Associative containers contain key/value pairs, providing access to each *value* using a *key*. The associative containers are designed with an intention to optimize the retrieval of data by organizing the single data records in a specialized structure (e.g. in a tree) using *key*s for identification. The library provides four different kinds of associative containers: *map*, *multimap*, *set*, and *multiset*.

*set* and *map* support unique keys, that means that those containers may contain at most one element (data record) for each key. *multiset* and *multimap* support equal keys, so more than one element can be stored for each key. The difference between *set* (*multiset*) and *map* (*multimap*) is that a set (map) stores data which inherently contains the key expression. map (multimap) stores the key expression and the appropriate data separately, i.e. the key has not to be part of the data stored.

- *map* provides access to elements using any type of key. This is a generalization of the idea of accessing a vector with an integer subscript
- *multimap* is the same as map, but allows a key to map into more than one element.
- *set* orders the elements that are added to it. A set contains only one copy of any value added to it. Please note it does not implement standard set operations like union, intersection.
- *multiset* is like set but allows more than one entry with the same value.

### C. Container adapters

Container adapters are based on other containers, and are used only to enforce access rules. Because there are special access restrictions, they have no iterators. There are three container adapters in STL: *stack*, *queue*, and *priority_queue*.

- *stack* allows only LIFO (Last In, First Out)
- *queue* allows only FIFO (First In, First Out)
- *priority_queue* always returns the element with the highest priority

### D. Common Operations of Container Classes

Every container class (sequence or associative containers) provides a default constructor, a copy constructor, and a destructor. The table I gives a summary of common operations of sequence and associative containers.

| **Operation** | **Description** |
|---|---|
| ContainerType c | The default constructor creates an empty container without any element. |
| ContainerType c1(c2) | The copy constructor copies a container of the same type. |
| ∼ContainerType() | Destructor deletes all elements and frees the memory. |
| c.size() | Returns the actual number of elements in the container c. |
| c.empty() | Returns if the containers is empty or not (equivalent to: c.size()==0, but might be faster). |
| c.max_size() | Returns the maximum number of elements possible. |
| c1 = c2 | Assigns all elements of c1 to c2. |
| c1.swap(c2) | Swaps the data of c1 and c2. |
| c.begin() | Returns the iterator for the first element. |
| c.end() | Returns an iterator for the position after the last element. |
| c.rbegin() | Returns a reverse iterator for the first element of a reverse iteration. |
| c.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration. |
| c.insert(pos, item) | Inserts a copy of item (return value and the meaning of pos differ). |
| c.erase(beg, end) | Removes all elements of the range [beg, end). |
| c.clear() | Removes all elements to make the container empty. |

TABLE I

COMMON OPERATIONS OF CONTAINER CLASSES

## II. VECTOR CLASS

The *vector* class is one of the basic classes implemented by the Standard Template Library. A vector is, essentially, a resizeable array; the *vector* class allows random access via the [] operator, but adding an element anywhere but to the end of a vector causes some overhead as all of the elements are shuffled around to fit them correctly into memory. Fortunately, the memory requirements are equivalent to those of a normal array. The header file for the STL vector library is vector. (Note that when using C++, header files drop the .h; for C header files - e.g. stdlib.h - you should still include the .h.) Moreover, the vector class is part of the *std* namespace, so you must either prefix all references to the vector template with std:: or include "using namespace std;" at the top of your program.

Vectors are more powerful than arrays because the number of functions that are available for accessing and modifying vectors. Unfortunately, the [] operator still does not provide bounds checking. There is an alternative way of accessing the vector, using the function **at**, which does provide bounds checking at an additional cost. Table II shows constructors, functions, and operators of the *vector* class. However, the common functions defined for all containers like size, empty, begin, end, etc. are shown in Table I, which assumes that T is some type (eg, int).

```
T e;
vector<T> v, v1;
vector<T>::iterator iter, iter2, beg, end;
vector<T>::reverse_iterator riter;  /* beg, end could also be here */
int i, n;
bool b;
```

| **Operation** | **Description** |
|---|---|
| vector<T> v; | Creates an empty vector of T's. |
| vector<T> v(n); | Creates a vector of *n* default values. |
| vector<T> v(n, e); | Creates a vector of *n* copies of *e*. |
| vector<T> v(beg, end); | Creates a vector with elements copied from range beg..end. |
| ∼vector<T>(); | Destroys all elements and frees memory. |
| i = v.capacity(); | Maximum number of elements before reallocation. |
| v[i] = e; | Sets *i*th element. Subscripts from zero. |
| v.**at**(i) = e; | As subscription, but may throw *out_of_range exception*. |
| v.**front**() = e; | **front** returns the reference to the 1st element. Same as v[0] = e; |
| v.**back**() = e; | **back** returns the reference to the last element. Same as v[v.size()-1) = e |
| v.**push_back**(e); | Adds *e* to end of *v*. Expands *v* if necessary. |
| v.**pop_back**(); | Removes last element of *v*. |
| iter2 = v.**insert**(iter, e); | Inserts a copy of *e* at *iter* position and returns its position. |
| v.**insert**(iter, n, e); | Inserts *n* copies of *e* starting at *iter* position. |
| v.**insert**(iter, beg, end); | Inserts all elements in range *beg..end*, starting at *iter* position. |
| iter2 = v.**erase**(iter); | Removes element at *iter* position and returns position of next element. |
| iter = v.**erase**(beg, end); | Removes range *beg..end* and returns position of next element. |
| e = v[i]; | ith element. No range checking. |
| e = v.**at**(i); | As subscription, but may throw *out_of_range*. |
| e = v.**front**(); | First element. No range checking. |
| e = v.**back**(); | Last element. No range checking. |
| ++ iter; | Preincrement *iter* to next element. Use in both forward and reverse iterators normally. Can also postincrement. |
| −− iter; | Similar to ++ iter; |
| iter2 = iter + i; | Iterator *i* elements after iter. += assignment also defined. |
| iter2 = iter - i; | Iterator *i* elements before iter. -= assignment also defined. |
| e = *iter; | Dereference the iterator to get the value. |

TABLE II

OPERATIONS OF THE *vector* CLASS

The following is an example showing the general usage of vector, and two different approaches to create and initialize a two-dimensional array.

```
#include <iostream>
#include <string>
#include <vector>
#include <iomanip>

using namespace std;

int main() {
    vector<string> v;  // creates an empty vector of string

    // insert four elements into the vector
    v.push_back("one");
    v.push_back("two");
    v.push_back("three");
    v.push_back("four");


    cout << "Using at functions to access elements" << endl;
    for (int i=0; i<v.size(); i++ )
        cout << v.at(i) << endl;
    cout << endl;

    cout << "Using [] to access elements" << endl;
    for (int i=0; i<v.size(); i++ )
        cout << v[i] << endl;
    cout << endl;

    cout << "Using iterator to access elements" << endl;
    for (vector<string>::iterator iter=v.begin(); iter != v.end(); iter++ )
        cout << *iter << endl;


    // The following create a two-dimensional array
    cout << endl << "Two dimensional array example" << endl;

    int     x, y;
    cout << "Please enter row number: ";
    cin >> x;
    cout << "Please enter column number: ";
    cin >> y;

    vector< vector<int> >       graph;  // create an empty vector of an empty vector of integer
    vector<int>     row(5);

    for ( int i=0; i<x; i++ )
    {
        row.clear();
        for (int j=0; j<y; j++ )
            row.push_back(i+j);
        graph.push_back( row );
    }

    /* Another approach
    vector< vector<int> >       graph(x, vector<int>(y));
    // Create an empty two-dimensional array with x rows and y columns

    for ( int i=0; i<graph.size(); i++ )
        for (int j=0; j<graph[i].size(); j++ )
            graph[i][j] = i + j;
    */


    for (int i=0; i<graph.size(); i++)
    {
        for (int j=0; j<graph[i].size(); j++)
            cout << setw(2) << graph[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

## III. DEQUE CLASS

The Standard Template Library (STL) sequence container *deque* (double-ended queue) arranges elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container. However, unlike a vector, the *deque* class also supports efficient insertion and deletion at the front of the container. Table III shows the general functions of the class *deque*.

```
#include <string>
#include <deque>
#include <iostream>
```

```
T e;
deque<T>::iterator iter;
```

| Operation | Description |
|---|---|
| deque<T> d; | Creates an empty double-ended queue of T's. |
| d[i] | Returns a REFERENCE to the element whose position is *i*, or undefined if *i* is greater than the size of the deque. |
| d.**at**(i) | As subscription, but throw *out_of_range exception* if i is greater than the size of the queue. |
| d.**front**(); | **front** returns the reference to the 1st element. Same as v[0] = e; |
| d.**back**() | **back** returns the reference to the last element. Same as v[v.size()-1) = e |
| d.**push_back**(e); | Adds *e* to end of *d*. Expands *d* if necessary. |
| d.**pop_back**(); | Removes last element of *d*. |
| d.**push_front**(e); | Adds *e* to beginning of *d*. Expands *d* if necessary. |
| d.**pop_front**(); | Removes the element at the beginning of *d*. |
| iter2 = d.**insert**(iter, e); | Inserts a copy of *e* at *iter* position and returns its position. |

TABLE III

OPERATIONS OF THE *deque* CLASS

```
#include <algorithm>

using namespace std;

void print( deque<string> );

void example1();

void example2();

int main( void ) {
    cout << "Example1 .................." << endl;
    example1();
    cout << "End of Example1........................." << endl;

    cout << endl << endl;

    cout << "Example2 ............................." << endl;
    example2();
    cout << "End of Example2........................." << endl;

    system("pause");

    return 0;
}

void example1()
{
    deque<char> dequeObject1;
    char str[] = "Using a deque.";

    int i;

    for(i = 0; i<strlen(str); i++) {
          dequeObject1.push_front(str[i]);
          dequeObject1.push_back(str[i]);
    }

    cout << "Original dequeObject1:\n";
    for(i = 0; i <dequeObject1.size(); i++)
          cout << dequeObject1[i];
    cout << "\n\n";

    for(i = 0; i <strlen(str); i++)
          dequeObject1.pop_front();
    cout << "dequeObject1 after popping front:\n";
    for(i = 0; i <dequeObject1.size(); i++)
          cout << dequeObject1[i];
    cout << "\n\n";

    deque<char> dequeObject2(dequeObject1);
    cout << "dequeObject2 original contents:\n";
    for(i = 0; i <dequeObject2.size(); i++)
          cout << dequeObject2[i];
    cout << "\n\n";
    for(i = 0; i <dequeObject2.size(); i++)
          dequeObject2[i] = dequeObject2[i]+1;

    cout << "dequeObject2 transposed contents:\n";
    for(i = 0; i <dequeObject2.size(); i++)
          cout << dequeObject2[i];
    cout << "\n\n";
```

```
    deque<char>::iterator p = dequeObject1.begin();
    while(p != dequeObject1.end()) {
            if(*p == 'a') break;
            p++;
    }

    dequeObject1.insert(p, dequeObject2.begin(), dequeObject2.end());

    cout << "dequeObject1 after insertion:\n";
    for(i = 0; i <dequeObject1.size(); i++)
          cout << dequeObject1[i];
    cout << "\n\n";

    return;
}


void example2() {
    deque<string> dequeObject;

    dequeObject.push_back( "yak" );
    dequeObject.push_back( "zebra" );
    dequeObject.push_front( "cat" );
    dequeObject.push_front( "canary" );

    print(dequeObject);

    dequeObject.pop_front();
    dequeObject.pop_back();

    print(dequeObject);
    //list operations on a deque:
    dequeObject.erase(find( dequeObject.begin(), dequeObject.end(), "cat" ));
    print(dequeObject);
    dequeObject.insert( dequeObject.begin(), "canary" );
    print(dequeObject);
    int sz = dequeObject.size();
    dequeObject.resize( 5 );
    dequeObject[sz] = "fox";
    dequeObject[sz+1] = "elephant";
    dequeObject[sz+2] = "cat";
    print( dequeObject );
    dequeObject.erase( dequeObject.begin() + 2 );
    print( dequeObject );

    //sorting a deque. sort function is defined in STL
    sort( dequeObject.begin(), dequeObject.end() );
    print( dequeObject );

    return;
}

void print( deque<string> d ) {
    deque<string>::const_iterator iter;
    cout << "The number of items in the deque:" << d.size() << endl;
    for ( iter = d.begin(); iter != d.end(); iter++ )
         cout << *iter << " ";
    cout << endl << endl;
}
```