

Problem Solving

STL PART 1

SA, 20 NOV 2021





Standard Template Library (STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators.
- The STL contains several kinds of entities. The three most important
 - Containers
 - Iterators
 - Algorithms



Containers

- A container is a holder object that stores a collection of other objects (its elements), they are implemented as class templates, which allows a great flexibility in the data types, whether the data consists of built-in types such as int and float, or of class objects.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Q: Why not use C++ arrays in all data storage situations?





Type of Containers

Sequence	Adaptors	Associative	Unordered Associative
Array	Queue	Set	Unordered Set
Vector	Priority Queue	Multiset	Unordered Multiset
List	Stack	Map	Unordered map
Forward List		Multimap	Unordered Multimap

Q: Why not use C++ arrays in all data storage situations?

A: The answer is efficiency. An array is awkward or slow in many situations and its not dynamic.

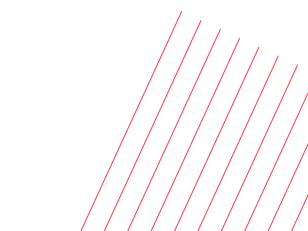





Sequence Containers

Implementation of data structures that can be accessed in a sequential manner, each element is related to the others by its location along the chain like arrays.

Container	Characteristic	Advantages and Disadvantages
array	Fixed size	<ul style="list-style-type: none">• Quick random access (by index number).• Items cannot be inserted or deleted.
vector	Expandable array	<ul style="list-style-type: none">• Quick random access (by index number).• Size of the vector can be changed (insert or delete).
list	Doubly linked list	<ul style="list-style-type: none">• Quick to insert or delete at any location.• Slow random access.
deque	Like vector, but can be accessed at either end	<ul style="list-style-type: none">• Quick random access (by index number).• Slow to insert or erase in the middle.





Associative Containers

- An associative container is not sequential; instead it uses keys to access data.
- It stores and sort data in a structure called a tree, which provides fast searching, insertion and deletion

Container	Characteristics
set	<ul style="list-style-type: none">• Stores only the key objects.• Only one key of each value allowed.
multiset	<ul style="list-style-type: none">• Stores only the key objects.• Multiple key values allowed.
map	<ul style="list-style-type: none">• Associates key object with value object.• Only one key of each value allowed.
multimap	<ul style="list-style-type: none">• Associates key object with value object.• Multiple key values allowed.





Adaptors Containers

- Used to provide different interface to the sequence containers limiting functionality.

Container	Implementation	Characteristics
stack	Can be implemented as vector, list, or deque	<ul style="list-style-type: none">Insert (push) and remove (pop) at one end only
queue	Can be implemented as list or deque	<ul style="list-style-type: none">Insert (push) at one end, remove (pop) at other
Priority queue	Can be implemented as vector or deque	<ul style="list-style-type: none">Insert (push) in random order at one end, remove (pop) in sorted order from other end

Sequence Containers



1. Arrays

- An array works fine when we have to implement sequential data structures with static size, i.e. we have to define its maximum size during its initialization and it cannot contain elements greater than its maximum size.



2. Vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted.
- Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens.
- Inserting and erasing at the beginning or in the middle is linear in time.

■ Fast insertion and removal at end.

■ Slow insert/remove at the beginning or in the middle.

■ Slow search.





How to create Vector?

- SYNTAX for creating a vector is: `vector<object_type> vector_name;`
- There are many ways to create a vector including:

```
#include <vector>

using namespace std;

int main()
{
    // Vector being a dynamic array, doesn't needs size during declaration.
    vector<int> vector_name;
    // vector_name_2 = {"C++", "Java", "Python", "Coder"}.
    vector<string> vector_name_2{"C++", "Java", "Python", "Coder"};
    // vector_name_3 = {"Test", "Test", "Test", "Test"}.
    vector<string> vector_name_3(3, "Test");
}
```



Iterators

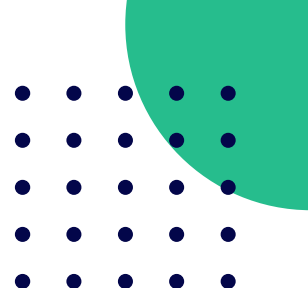
- Iterators are a generalization of the concept of pointers: they point to elements in a container.
 - We can use iterators to move through the contents of the container.
 - They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them.
 - A pointer can point to elements in an array and can iterate through them using the increment operator (`++`).
-
- But, all iterators do not have similar functionality as that of pointers.





Member functions in Vector

Functions based on iterator concept



Container	Purpose
<code>begin()</code>	Returns an iterator to the start of the container, for iterating forwards through the container
<code>end()</code>	Returns an iterator to the past-the-end location in the container, used to end forward iteration
<code>rbegin()</code>	Returns a <code>reverse_iterator</code> to the end of the container, for iterating backward through the container
<code>rend()</code>	Returns a <code>reverse_iterator</code> to the beginning of the container; used to end backward iteration


Container	Purpose
<code>cbegin()</code>	Returns a constant iterator pointing to the first element in the container.
<code>cend()</code>	Returns a constant iterator pointing to the theoretical element that follows the last element.
<code>crbegin()</code>	Returns a constant reverse iterator pointing to the last element in the container (reverse beginning). It moves from last to first element
<code>crend()</code>	Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the container.





Member functions in Vector

Iterators can be used to traverse the container, and we can de-reference the iterator to get the value of the element it is pointing to.

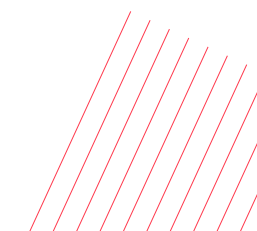




```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // creates an vector
    vector<string> my_vector{"C++", "Java", "Python", "Coder"};

    for (vector<int>::iterator i = my_vector.begin(); i != my_vector.end() ; i++)
        cout << *i << " ";

    /* in the above for loop iterator I iterates though the vector v and
       *operator is used of printing the element pointed by it. */
    return 0;
}
```





Member functions in Vector

Capacity functions

Container	Purpose
<code>size()</code>	Returns the number of elements in the vector.
<code>max_size()</code>	Returns the maximum number of elements that the vector can hold.
<code>capacity()</code>	Returns the size of the storage space currently allocated to the vector expressed as number of elements.
<code>resize(n)</code>	Resizes the container so that it contains 'n' elements.

Container	Purpose
<code>empty()</code>	Returns whether the container is empty.
<code>reserve()</code>	Requests that the vector capacity be at least enough to contain n elements.
<code>shrink_to_fit()</code>	Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.



Member functions in Vector

Modifiers functions

Container	Purpose
<code>push_back()</code>	It push the elements into a vector from the back
<code>pop_back()</code>	It is used to pop or remove elements from a vector from the back.
<code>insert()</code>	It inserts new elements before the element at the specified position
<code>erase()</code>	It is used to remove elements from a container from the specified

Container	Purpose
<code>clear()</code>	It is used to remove all the elements of the vector container
<code>assign()</code>	It assigns new value to the vector elements by replacing old ones.
<code>emplace_back()</code>	It is used to insert a new element into the vector container, the new element is added to the end of the vector.
<code>emplace()</code>	It extends the container by inserting new element at position.





Examples:

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

int main()
{
    // Assign vector
    vector<int> v;

    // fill the array with 10 five times
    v.assign(5, 10);

    cout<<"The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
        cout<<v[i]<<" ";

    return 0;
}
```

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

int main()
{
    // Assign vector
    vector<int> v{40, 556, 5634};



    // removes the first element
    v.erase(v.begin());

    cout<<"\nThe first element is: "<<v[0];
    return 0;
}
```

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

int main()
{
    // Assign vector
    vector<int> v;
    v.push_back(1);
    v.push_back(2);

    cout<<v.size()<<"\n";
    return 0;
}
```





3. Deque

- Deque is a double ended queues are sequence containers with the feature of expansion and contraction on both the ends.
- They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.

■ Fast insert/remove at beginning and the end.

■ Slow insert/remove in the middle

■ Slow search



Examples:

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

using namespace std;

int main ()
{
    deque<int> dq{1, 5, 8, 9, 3};

    dq.push_back(10);
    /* now dq is : {1, 5, 8, 9, 3, 10} */

    dq.push_front(20);
    /* now dq is : {20, 1, 5, 8, 9, 3, 10} */

    for(auto &it : dq)
        cout<<it<<" ";
    cout<<"\n";
}
```

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

using namespace std;

int main ()
{
    int a[]={153, 7456, 17 , 43 ,24 , 5, 543};
    deque<int> dq(a, a+7);

    deque<int>::iterator i;

    i=dq.begin()+2;
    /* i points to 3rd element in dq */

    dq.insert(i,15);
    /* now dq {153, 7456, 15, 17, 43, 24, 5, 543} */

    for(auto &it : dq)
        cout<<it<<" ";
    cout<<"\n";
}
```



References

Articles

■ harmash.com.

- <https://harmash.com/cplusplus/cplusplus-stl/>

■ Vector (cplusplus.com).

- <https://www.cplusplus.com/reference/vector/vector/>

■ Vector (cplusplus.com).

- <https://www.cplusplus.com/reference/deque/deque/deque/>

Videos

■ Adel Nasim (Intro).

- <https://www.youtube.com/watch?v=4hhz69S15wU>

■ Adel Nasim (vector 1).

- <https://www.youtube.com/watch?v=AMnultLTdII>

■ Adel Nasim (vector 2).

- <https://www.youtube.com/watch?v=4b3Glt6zWY4>

■ Adel Nasim (vector 3).

- https://www.youtube.com/watch?v=4om6SL_cF50

■ Adel Nasim (deque).

- <https://www.youtube.com/watch?v=PPFhtX23oXc>





THANKS!

Any Questions?!