Part: 71

Vector In C++ STL

In this video, we'll cover the Vectors in C++ STL. This is the tutorial we all were waiting for. Enough of the theory part. We'll go into our editors and code. So, to start, we'll have to include the header file <vector>. And the syntax we use to define a vector is:

```
vector<data_type> vector_name;
```

Code Snippet 1: Syntax of declaring a vector

And suppose we want to have a vector of integers; the following program would do the needful:

```
#include<iostream>
#include<vector>

int main(){

    vector<int> vec1;
    return 0;
}
```

Code Snippet 2: Declaring a vector of integers

One benefit of using vectors, is that we can insert as many elements we want in a vector, without having to put some size parameter as in an array. In an array of 10 elements, for adding the 11th one, we'll have to make an array again.

Vectors provide certain methods to be used to access and utilise the elements of a vector, first one being, the push_back method. To access all the methods and member functions in detail, you can visit this site , std::vector - C++ Reference. This will be very handy and useful to you. I'll show you how some of them work in a program. Refer to the code snippet 3.

- **push_back() and size():**

1. First of all, don't forget to include the header file, <vector>.
2. Vectors have a method, push_back(), to insert elements in it from the rear end.
3. We'll define a variable, size, to store the size of the vector.
4. We'll then run a loop of size length, to receive the user input and push them back in the vector vec1.
5. We'll then call the display function.
6. We want to have a display function to display the contents of the vector. And pass reference of vec1 to the function.
7. We have another method size() which returns the size of the vector. We'll use this to traverse through all the elements and print them.
8. So, this is how a vector gets used.

```
#include<iostream>
#include<vector>
using namespace std;
void display(vector<int> &v){
    for (int i = 0; i < v.size(); i++)
    {
        cout<<v[i]<<" ";
    }
```

```
        cout<<endl;
    }
    int main(){
        vector<int> vec1;
        int element, size;
        cout<<"Enter the size of your vector"<<endl;
        cin>>size;
        for (int i = 0; i < size; i++)
        {
            cout<<"Enter an element to add to this vector: ";
            cin>>element;
            vec1.push_back(element);
        }
        display(vec1);
        return 0;
    }
```

Code Snippet 3: A program to demonstrate the use of push_back and size methods

The output of the above program:

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Output of the above program

Similarly, we can even build float vectors, and we can even templatise the display function.

- **pop_back():**

This method of vectors, deletes the last element of the vector. Refer to the code snippet and the following output below.

```
display(vec1);
vec1.pop_back();
display(vec1);
```

Code Snippet 4: Using pop_back in a vector

So, now you can see how this method deleted the last element 7 from the vector.

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
5 3
PS D:\MyData\Business\code playground\C++ course>
```

Figure 2: Output of the above program

- **Insert** (iterator, element to insert)**:**

This method of vectors inserts an element to the position the iterator is pointing to. Now how to evoke that iterator? Refer to the snippet and the output below:

We can generate an iterator using the scope resolution iterator by the following syntax:

```
vector<int> :: iterator iter = vec1.begin();
```

Code Snippet 5: Declaring a vector iterator

Using **begin ()** points the iterator to the starting of the vector. We can now increment the pointer according to our choice and insert any element at that position.

```
display(vec1);
    vector<int> :: iterator iter = vec1.begin();
    vec1.insert(iter,566);
    display(vec1);
```

Code Snippet 6: Demonstrating an insert method

The output of the above program is:

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
566 5 3 7
PS D:\MyData\Business\code playground\C++ course>
```

Figure 3: Output of the above program

Similarly, **v.at(i)** can be used instead of **v[i]**. They will work the same.

We have different ways to declare a vector. I'll list some of them through the snippet below.

1. First one is a vector with no length and elements specified.
2. Second one is a vector of length 4 and no elements.
3. Third one is a vector made from the second one.
4. And last one, is a vector with length 6 and all the elements being 3.

```
vector<int> vec1;         //zero length integer vector
vector<char> vec2(4);     //4-element character vector
vector<char> vec3(vec2);  //4-element character vector from vec2
vector<int> vec4(6,3);    //6-element vector of 3s
```

Code Snippet 7: Demonstrating different ways to declare a vector

So, this was all the basics of a vector, and enough for you to get started using it. With this I'll finish today's tutorial. I'll recommend you all to visit my DSA playlist, Data Structures and Algorithms Course in Hindi to get acquainted with all the data structures and more.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll learn about Lists in C++ STL. Till then keep coding.

Part: 72

List In C++ STL

Before this tutorial, we covered templates, STL, and the last video was an efficient introduction to the vectors. Today, we'll learn about Lists in C++ STL.

A List is a bi-directional linear storage of elements. Few key features as to why a list should be used is,

1. It gives faster insertion and deletion operations.
2. Its access to random elements is slow.

What makes a list different from an array?

An array stores the elements in a contiguous manner in which inserting some element calls for a shift of other elements, which is time taking. But in a list, we can simply change the address the pointer is pointing to. I'll show you how these work via an illustration.

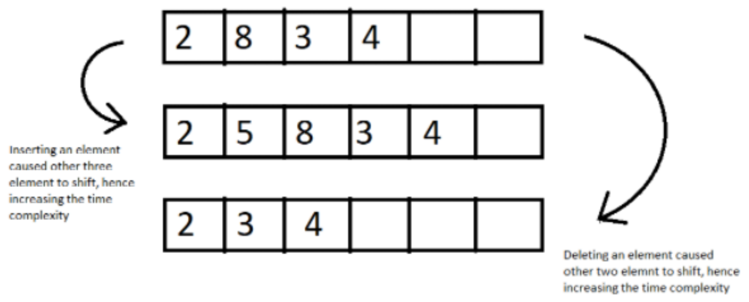
2 8 3 4
2 5 8 3 4
2 3 4
Inserting an element caused other three element to shift, hence increasing the time complexity
Deleting an element caused other two elemnt to shift, hence increasing the time complexity

Figure 1: Inserting and deleting in an array

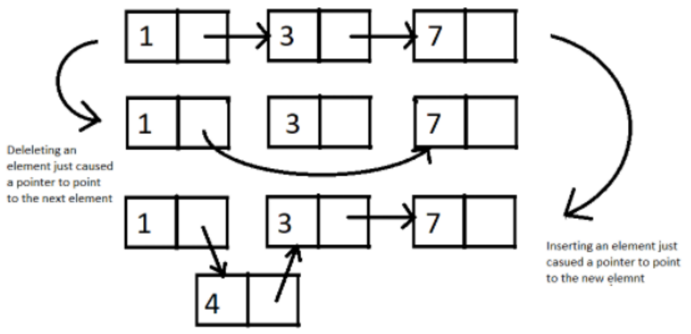
1 3 7
1 3 7
1 3 7
4
Deleleting an element just caused a pointer to point to the next element
Inserting an element just casued a pointer to point to the new elemnt

Figure 2: Insertion and deletion in a list

Let's move on to our editors and write some code using lists and its methods.

Understanding code snippet 2:

- Before using lists, we must include the header file <list>.
- Using a simple program, we'll iterate through the list and display its contents.
- As we did for vectors, first define a list list1.
- And push_back a few elements, and pass the list to a display function via reference.
- Due to the fact that a list element cannot be directly accessed by its index, we must traverse through each element and print them.
- We define a list iterator using this syntax:

```
list<int> :: iterator it;
```

Code Snippet 1: Syntax for defining a list iterator

- We use two methods, **`begin()`** and **`end()`** to define the starting and the end of the loop. **`end()`** returns the pointer next to the last element.
- We dereference the list iterator, using * to print the element at that index.

```
#include<iostream>
#include<list>

using namespace std;

void display(list<int> &lst){
    list<int> :: iterator it;
    for (it = lst.begin(); it != lst.end(); it++)
    {
        cout<<*it<<" ";
    }

}

int main(){

    list<int> list1;  //empty list of 0 length

    list1.push_back(5);
    list1.push_back(7);
    list1.push_back(1);
    list1.push_back(9);
    list1.push_back(12);

    display(list1);

    return 0;
}
```

Code Snippet 2: A program using list

```
5 7 1 9 12
PS D:\MyData\Business\code playground\C++ course>
```

Figure 3: Output of the above program

We can also enter elements in a list using the iterator and its dereferencer. See the snippet below.

```
int main(){

    list<int> list2(3);  //empty list of length 3
    list<int> :: iterator it = list2.begin();
    *it = 45;
    it++;
    *it = 6;
    it++;
    *it = 9;
    it++;

    display(list2);

    return 0;
}
```

Code Snippet 3: Inserting in list using its iterator

```
45 6 9
PS D:\MyData\Business\code playground\C++ course>
```

Figure 4: Output of the above program

- **Using pop_back() and pop_front():**

We can use pop_back() to delete one element from the back of the list everytime we call this method and pop_front() to delete elements from the front. These commands decrease the size of the list by 1. Let me show you how these work by using them for list1 we made.

```
    list1.pop_back();
    display(list1);
    list1.pop_front();
    display(list1);
```

Code Snippet 4: Using pop_back and pop_front in list

The output of the above program is:

```
5 7 1 9
7 1 9
PS D:\MyData\Business\code playground\C++ course>
```

Figure 5: Output of the above program

- **Using remove():**

We can remove an element from a list by passing it in the list remove method. It will delete all the occurrences of that element. The remove method receives one value as a parameter and removes all the elements which match this parameter. Refer to the use of remove in the below snippet.

```
int main(){

    list<int> list1;  //empty list of 0 length

    list1.push_back(5);
    list1.push_back(7);
    list1.push_back(1);
    list1.push_back(9);
    list1.push_back(9);
    list1.push_back(12);

    list1.remove(9);
    display(list1);

    return 0;
}
```

Code Snippet 5: Deleting elements in list using remove()

The output of the above program is:

```
5 7 1 12
PS D:\MyData\Business\code playground\C++ course>
```

Figure 6: Output of the above program

- **Using sort():**

We can sort a list in ascending order using its sort method. Look for the demo below.

```
    display(list1);
    list1.sort();
    display(list1);
```

Code Snippet 6: Sorting elements in list using sort()

```
5 7 1  9 12
1 5 7  9 12
PS D:\MyData\Business\code playground\C++ course>
```

Figure 7: Output of the above program

I consider this much to be enough for lists. There are still a lot of them, but you will require no more, and even if you feel like exploring more, move onto **`std::list`** - C++ Reference and read about all the lists methods. This was all from my side. For more information on linked lists, visit my DSA playlist, Data Structures and Algorithms Course in Hindi.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll learn about Maps in C++ STL. Till then keep coding.

Part: 73

Map In C++ STL

So far, we have learned about vectors and lists in C++ STL, and today we will be learning about maps in C++ STL. It is important to clarify that whatever I have taught and whatever I will be teaching in the coming tutorials about STL isn't everything. And it is definitely not all. These are just the most important STL containers we will use. You have already seen how to explore more about STL from C++ - Containers.

We will now discuss maps, and because it is impractical to have every method on our fingers, I'd ask you all to also refer to the following website `std::map` - C++ Reference

A map in C++ STL is an associative container which stores key value pairs. To elaborate, a map stores a key of some data type and its corresponding values of some data type. For example: a teacher wants to store the marks of students which in future can be accessed by their names. Here, keys are the student names, and their marks are the corresponding values. Refer to the illustration below:

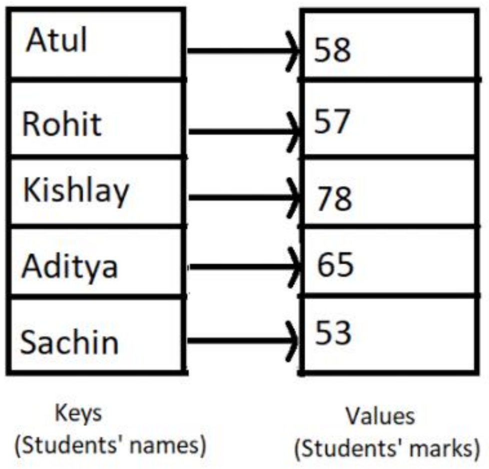

Figure 1: Illustrative diagram of a key value pairs

We can now shift to our editors and see how maps can be used in C++. Don't forget to include the header file <map>.

The syntax for declaring a map is:

```
map <data_type_of_key,   data_type_of_value>  variable_name;
```

Code Snippet 1: Syntax for declaring a map

And we can now write the program for storing the key value pairs of students' names and students' marks keeping in mind the illustration above. Refer to the snippet below.

Understanding code snippet 2:

1. Include the header file map and string (if using string methods).
2. Let's create a map in which the key is a string (names) and the values are integers (marks), and we'll call it marksMap.
3. And to assign some key a value, we use the index method. Here the index of a map element will be the students' name and the value will be the marks.
4. Make some 4-5 elements.

5. Identify the iterator of this map by using the scope resolution operator.
6. Loop through the map elements using two map methods; **begin()** to point at the beginning of the map, and **end()** to point next to the last element of the map.
7. While we loop through the map, we use the dereference operator * to fetch the element present where the pointer is pointing to. And since a map stores element in a key value pair, we can use its first and second method to access the keys and the values respectively. **.first** accesses the first value of a pair that is our map key here, and **.second** accesses the second value of the pair that is our map values here.
8. There is one thing to keep in mind: Maps always sort these pairs by the key elements. You can review the output of the following snippet to see how these pairs are sorted.

```
#include<iostream>
#include<map>
#include<string>

using namespace std;

int main(){

    // Map is an associative array
    map<string, int>  marksMap;
    marksMap["Atul"] = 58;
    marksMap["Rohit"] = 57;
    marksMap["Kishlay"] = 78;
    marksMap["Aditya"] = 65;
    marksMap["Sachin"] = 53;

    map<string,int> :: iterator iter;
    for (iter = marksMap.begin(); iter != marksMap.end(); iter++)
    {
        cout<<(*iter).first<<" "<<(*iter).second<<"\n";
    }


    return 0;
}
```

Code Snippet 2: A program to store names and marks using map

As you can see, the map pairs got sorted according to its key.

```
Aditya 65
Atul 58
Kishlay 78
Rohit 57
Sachin 53
PS D:\MyData\Business\code playground\C++ course>
```

Figure 2: Output of the above program

We have one more method to insert elements in a map. We can use **.insert()**

Syntax for using .insert is:

```
marksMap.insert({pair_1,pair_2......pair_n});
```

Code Snippet 3: Syntax for inserting pairs in map

We will insert some elements into our map in snippet 2 by using the insert method.

```
marksMap.insert( { {"Rohan", 89}, {"Akshat", 46} } );
```

Code Snippet 4: Program to insert two pairs in a map.

We can see the output to check if the above two pairs got inserted into the map or not.

```
Aditya 65
Akshat 46
Atul 58
Kishlay 78
Rohan 89
Rohit 57
Sachin 53
PS D:\MyData\Business\code playground\C++ course>
```

Figure 3: Output of the above program

So, yes, it worked. And the output was correct.

And I'd ask you all to explore more of these methods from the links I gave you above and start writing codes using them. This is the best way to learn. For example, you can use the **`size()`** method to get the size of the map container , **empty()** method to check if the map container is empty or not, and it returns a boolean. Therefore, this shouldn't be a big deal for you.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. You will surely enjoy them all. See you all in the next tutorial where we'll learn about Maps in C++ STL. Till then keep coding.

Part: 74

Function Objects (Functors) In C++ STL

In the last tutorial we completed learning about some of the most commonly used containers, vector, list, map and their methods. Today we'll start with function objects in C++ STL.

What is a function object?

A function object is a function wrapped in a class so that it is available as an object.

That is, we can then use a function as an object. The question that might have been raised in your mind would be, **why to substitute a function with an object**? The answer is to make them all usable in an Object-Oriented Programming paradigm. Now what does that mean? We'll try decoding the purpose of using functions as an object via a program. So, hold onto your editors.

Understanding code snippet 1:

- Be sure to include the header file < functional> before you do anything else.
- And let's create an array of some 6 elements.
- Suppose we want to sort this array in ascending order. So we'll include a header file <algorithm> and write the syntax of the sort object which is,

```
sort(address of first element, address of last element);
```

Code Snippet 1: Syntax for sort algorithm

- And let's just sort from the beginning to the 5th element.
- And run a loop to see the resultant array.

```
#include<iostream>
#include<functional>
#include<algorithm>

using namespace std;

int main(){

    // Function Objects (Functor) : A function wrapped in a class so that it is
available like an object
    int arr[] = {1, 73, 4, 2, 54, 7};
    sort(arr,arr+5);
    for (int i = 0; i < 6; i++)
    {
        cout<<arr[i]<<endl;
    }

    return 0;
}
```

Code Snippet 2: Program to sort an array in ascending order

```
Output of the above program is given below.  And you'll notice that
the last element remained untouched.
1
2
4
54
73
7
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Output of the above program

But what if we wanted to sort the same array in descending order, since the sort function can default sort in ascending order only? So, here comes our saviour, **functional objects**. Our sort function also takes a third parameter which is a functor (functional object).

Let's see how they work via the snippet below:

- Among all the different functors we have, the one to help this sort function to sort the array in descending order, is the **greater<int>()**.

```
sort( arr, arr+6, greater< int >( ));
```

Code Snippet 1: Syntax for using a functor in an algorithm

- And that's it. Our array will now get sorted in descending order.

See the output after the above changes we made:

```
73
54
7
4
2
1
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Output of the above program after using a functor greater<int>()

It would be unnecessarily lengthy to review all the functors, as my role was to introduce them to you and show you how they are used.

In addition, I invite you all to explore the other function objects on the site Function objects. You should go through them lightly because a lot of them would be overwhelming to you as a beginner. We use most of these functors for our STL algorithms, so you might want to check all those algorithms on Functions in <algorithm> - C++ Reference. You can go through them at your own pace. Incorporate them into your programs. Take advantage of them and use them how you see fit.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. You will surely enjoy them all. Looking forward to seeing you all next time. Till then keep coding.