find() in C++ STL

The **find()** function in C++ STL is used to find an element in a given range in any container. This function returns an iterator to the first element in the range [first,last) that compares equal to a given value *val*. If no such element is found, the function returns *last*.

The find() function does a linear search on the container provided in the range [first, last) and compares the value to be searched every time with the current value during the search operation.

Header File: The find() function is declared in the header file "algorithm".

Syntax:

```
find (InputIterator first, InputIterator last, Type val)
```

Parameters:

- **first,last** : Input iterators to the initial and final positions in a sequence. The range searched is [first, last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.
- **val** : Value to search in the range.

Return Value: This function returns an iterator to the first element in the range that compares equal to val. If no elements match, the function returns last.

Using find() with Vectors: Below program illustrates the use of find() function with vectors.

```cpp

// CPP program to illustrate
// find() function with Vector
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main ()
{
    vector<int> vec { 5, 10, 7, 10, 20 };

    // Iterator used to store the position
    // of searched element
    auto it = find(vec.begin(), vec.end(), 10);

    // If iterator is equals to the iterator
    // pointing to past-the-end element, then
    // 10 does not exist in vector
    if(it == vec.end())
        cout<<"Not Found";
    else
        // Otherwise, calculate position of 10
        // by subtracting iterator from begin iterator
        cout<<"Found at Pos: "<<(it-vec.begin());

    return 0;
}

```

Run

Output:

```
Found at Pos: 1
```

Using find() function with Arrays: We can also use the find() function with arrays or any other container. For using find() on arrays, we will have to pass the address of the range instead of iterators as shown in the below program.

```cpp

// CPP program to illustrate
// find() function with Arrays
#include<iostream>
#include<algorithm>
```

```cpp
#include<algorithm>

using namespace std;

int main ()
{
    int arr[] = { 5, 10, 7, 10, 20 };

    int *ptr = find(arr, arr + 6, 10);

    // If the pointer is equals to the address
    // of past-the-end element, then
    // 10 does not exist in array
    if(ptr == (arr + 6))
        cout<<"Not Found";
    else
        // Otherwise, calculate position of 10
        // by subtracting pointer with begin address
        cout<<"Found at Pos: "<<(ptr - arr);

    return 0;
}
```

Run

Output:

```
Found at Pos: 1
```

Since, the find() function searches for an element linearly in the range, therefore, it can be used with any other container like list, map, set, unordered_set, unordered_map, etc. However, some containers like, map, set, unordered_map, unordered_set provides their own implementation of the find() function which works in O(1) time on average, so, it is highly recommended to use the find() function provided by the specific container.

Time Complexity: The std::find() function takes O(N) time, where N is the number of elements in the range as it uses linear search internally for comparisons.

However, some containers like unordered_set, map, set etc. stores elements in some specific order and provides their own implementation of find() function which works in O(1) time complexity on average. Therefore, it is highly recommended to use find() function provided by those containers only while searching for an element.

- lower_bound() in C++ STL

The **lower_bound()** is a built-in function in C++ STL which returns an iterator pointing to the first element which is greater than or equal to a given a value in a sorted range.

Header File: The lower_bound() function is declared in the "algorithm" header file.

Parameters: It accepts three arguments:

1. **beginning iterator**: It is an iterator pointing to the first element in the range.
2. **end iterator**: It is an iterator pointing to the element just after the last element of the sorted range.
3. **value**: It is the value to be searched for.

Return Values:

- If the value to be searched is smaller or equal to the elements in the sorted range provided, then the lower_bound() function returns an iterator pointing to the first element which is greater than or equal to the given a value in a sorted range.
- If the value to be searched is greater than all the values in the sorted range, then the function returns an iterator pointing to the last element.

Note: lower_bound works only on sorted containers.

Calculating Position of an element: If the value to be searched exists in the container, and if the iterator supports the subtraction operator ("-") then we can easily calculate the position of the element by simply subtracting the beginning iterator of the container from the iterator returned by the function.

function.

Below program illustrate the lower_bound function in STL:

```cpp

// C++ program to illustrate lower_bound() in STL
#include<iostream>
#include<algorithm>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the
    // first occurrence of 20
    auto it = lower_bound(v.begin(), v.end(), 20);

    cout<<(*it)<<endl;

    // Returns an iterator pointing to the
    // element just greater than 25, which is 30
    it = lower_bound(v.begin(), v.end(), 25);

    cout<<(*it)<<endl;

    it = lower_bound(v.begin(), v.end(), 30);

    // Calculating position of 30
    cout<<"30 is at position: "<<(it-v.begin())<<endl;

    return 0;
}
```

Run

Output:

```
20

30

30 is at position: 4
```

Using lower_bound() on arrays

We can use lower_bound() function with arrays also. The idea is to pass address of elements instead of iterators. We can pass the address of first element and address just after the last element.

```cpp

// C++ program to illustrate lower_bound() in STL
#include<iostream>
#include<algorithm>

using namespace std;

int main()
{
    int arr[] = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the
    // first occurrence of 20
    auto it = lower_bound(arr, arr + 6, 20);

    cout<<(*it)<<endl;

    // Returns an iterator pointing to the
    // element just greater than 25, which is 30
    it = lower_bound(arr, arr+6, 25);

    cout<<(*it)<<endl;

    return 0;
}
```

```
26
```

Run

Output:

```
20

30
```

Using lower_bound() to perform Binary Search

The lower_bound() function is internally implemented using Binary Search. Therefore, to perform Binary Search on a container for searching a key using lower_bound() function, we will have to include the below condition in the program:

```
if(it = end_iterator || (*it) != key)

   Key not present

else

   Key is present
```

```
1
2  // C++ program to implement Binary Search using
3  // lower_bound() in STL
4  #include<iostream>
5  #include<algorithm>
6
7  using namespace std;
8
9  int main()
10 {
11     int arr[] = {10, 20, 20, 20, 30, 40};
12
13     // Returns an iterator pointing to the
14     // first occurrence of 20
15     auto it = lower_bound(arr, arr + 6, 20);
16
17     if(it == (arr + 6) || (*it)!=20)
18         cout<<"Not Present";
19     else
20         cout<<"Present";
21
22     return 0;
23 }
24
```

Run

Output:

```
Present
```

Time Complexity

The lower_bound() function accepts two iterator for determining the start and end of the sorted range. This function takes **O(logN)** time, if these iterators are random access iterators.

The iterators on Vectors, Arrays are random-access iterators and thus lower_bound() function works best on these containers with O(logN) time complexity where N is the number of operations.

Whereas, for non-random access containers like List, Deque, Map, Set this function may take more than O(logN) time and thus it is not preferred to use lower_bound with non-random access containers.

upper_bound() in C++ STL

The **upper_bound()** is a built-in function in C++ STL which returns an iterator pointing to the first element which is greater than a given a value in a sorted range.

Header File: The upper_bound() function is declared in the "algorithm" header file.

Parameters: It accepts three arguments:

1. **beginning iterator**: It is an iterator pointing to the first element in the range.
2. **end iterator**: It is an iterator pointing to the element just after the last element of the sorted range.
3. **value**: It is the value for which we need to search first element greater than it.

Return Values: The upper_bound() function returns an iterator pointing to the first element which is greater than a given a value in the sorted range.

Note: The upper_bound() fucntion works only on sorted containers.

Below program illustrate the upper_bound function in STL:

```cpp

// C++ program to illustrate upper_bound() in STL
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the
    // first element greater than 20
    auto it = upper_bound(v.begin(), v.end(), 20);

    // Using dereference operator to print
    // value of iterator
    cout<<(*it)<<endl;

    // Returns an iterator pointing to the
    // element greater than 30, which is 40
    it = upper_bound(v.begin(), v.end(), 30);

    cout<<(*it)<<endl;

    // To find index of first greater element
    // just subtract the iterator returned by
    // function with beginning iterator of container
    it = upper_bound(v.begin(), v.end(), 20);
```

Run

Output:

```
30
40
Index of first greater element of 20 : 4
```

Counting number of occurrences of a given Element: We can use the lower_bound() function and the upper_bound() function together to count the number of occurrences of a particular element.

- Get iterator pointing to the first occurrence of the given element.
- Get an iterator pointing to the first greater element.
- Subtract both of the iterators to get the count of occurrences of that element.

Below program illustrates this:

```cpp

// C++ program to count occurrences
// of an element
```

```cpp
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the
    // first occurrence of 20
    auto it = lower_bound(v.begin(), v.end(), 20);

    // Returns an iterator pointing to the first
    // element greater than 20, which is 30
    auto it2 = upper_bound(v.begin(), v.end(), 20);

    // Subtract both of the iterators to get count
    cout<<"Count of 20: "<<(it2-it);

    return 0;
}

```

Run

Output:

```
Count of 20: 3
```

Check if an element exists using upper_bound(): We can also use upper_bound function to check if an element exists in the container or not. The idea is to first get the iterator pointing to the first greater element of the Key to be searched.

- If this iterator points to the first element of the container, it means that all elements are greater than the key and the key does not exist.
- Otherwise, if this iterator is not pointing to the first element and value at the previous iterator of this equal to the key then the exists.

Below program illustrates the above approach:

```cpp

// C++ program to check if an element exists
// using upper_bound

#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the first
    // element greater than 20, which is 30
    auto it = upper_bound(v.begin(), v.end(), 20);

    if(it != v.begin() && *(it-1) == 20)
        cout<<"Element Exists";
    else
        cout<<"Element Doesn't Exist";

    return 0;
}

```

Run

Output:

```
Element Exists
```

Using upper_bound() on arrays

We can use upper_bound() function with arrays also. The idea is to pass address of elements instead of iterators. We can pass the address of first element and address just after the last element.

```cpp

// C++ program to illustrate upper_bound() in STL
#include<iostream>
#include<algorithm>

using namespace std;

int main()
{
    int arr[] = {10, 20, 20, 20, 30, 40};

    // Returns an iterator pointing to the
    // first greater element of 20
    auto it = upper_bound(arr, arr + 6, 20);

    cout<<(*it)<<endl;

    // Returns an iterator pointing to the
    // first element greater than 25, which is 30
    it = upper_bound(arr, arr+6, 30);

    cout<<(*it)<<endl;

    return 0;
}

```

Run

Output:

```
30
40
```

Time Complexity

The upper_bound() function accepts two iterator for determining the start and end of the sorted range. This function takes **O(logN)** time, if these iterators are random access iterators.

The iterators on Vectors, Arrays are random-access iterators and thus upper_bound() function works best on these containers with O(logN) time complexity where N is the number of operations.

Whereas, for non-random access containers like List, Deque, Map, Set this function may take more than O(logN) time and thus it is not preferred to use upper_bound with non-random access containers.

is_permutation() in C++ STL

The **is_permutation()** is a built-in function in C++ STL which is used to check whether two given containers are permutations of each other or not. This function is available in the "algorithm" header file.

Permutation: Two containers are said to be a permutation of each other if the arrangement of elements in the two containers are different.

For Example: The two containers {10, 20, 30, 5} and {20, 10, 5, 30} are permutations of each other.

Syntax:

```
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);

first1, last1: Input iterators to the initial and final positions of the first sequence.
first2 : Input iterator to the initial position of the second sequence.

Note: All of the iterators passed as a parameter to the function are
```

```
ForwardIterator, i.e., one can only move forward using these iterators in the container.

Return Value:
true : if all the elements in range [first1, last1]
compare equal to those of the range
starting at first2 in any order.
false : Any element missing or exceeding.
```

Working: As we saw in the syntax, the is_permutation() function accepts both starting and ending iterators for the first container and only ending iterator of the second iterator, what it actually does is start comparing all of the elements of the first container in the range **(first1, last1)** with the elements in the second container starting from iterator **frist2**.

The is_permutation() function also works when there are duplicate elements in a container.

Below programs illustrate the is_permutation function:

- **Program 1**:

```cpp
// C++ program to illustrate is_permutation()

#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    // All elements in the two vectors are unique
    vector<int> v1 = {10, 20, 3, 5};
    vector<int> v2 = {20, 10, 5, 3};

    if(is_permutation(v1.begin(), v1.end(), v2.begin()))
    {
        cout<<"YES\n";
    }
    else
    {
        cout<<"NO\n";
    }

    // When elements in the two vectors are duplicate
    vector<int> v3 = {10, 20, 3, 5, 20};
    vector<int> v4 = {20, 10, 5, 3, 5};

    if(is_permutation(v3.begin(), v3.end(), v4.begin()))
    {
```

Run

Output:

```
YES
NO
```

- **Program 2**:

```cpp
// C++ program to illustrate is_permutation()

#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    // All elements in the two arrays are unique
    int a1[] = {30, 20, 10};
    int a2[] = {20, 10, 30};

    if(is_permutation(a1, a1+3, a2))
    {
```

```
        cout<<"YES\n";
    }
    else
    {
        cout<<"NO\n";
    }

    return 0;
}

```

Run

Output:

```
YES
```

Note: We can also use this function with other containers like forward_list, list, set, unordered_set, map, unordered_map etc. In the case of associative containers like map, unordered_map, the is_permutation() function only compares the keys.

Time Complexity: The worst-case time complexity of is_permutation() function is $O(N^2)$ as for every element in the first container, it calculates it's frequency in the first container and then in the second container, and if both of the frequencies are same then it moves forward in the first container.

Application: We can also use the is_permutation() function to *check whether two strings are anagram of each other*, but there is another function in C++ which does this in O(N) time complexity, so it is not recommended to use is_permutation() to use for this purpose.

- max_element() and min_element() in C++ STL

The **max_element()** function is a built-in function in C++ STL which is used to find the maximum element in a container whereas the **min_element()** function is used to find the minimum element in a container.

Header File: Both of these functions are available in the "algorithm" header file.

Syntax:

```
max_element(Iterator1, Iterator2);

          or,

min_element(Iterator1, Iterator2);

Iterator1: This is a forward iterator denoting the beginning of the range.
Iterator2: This is a forward iterator denoting the end of the range.

Return Value: It returns an iterator pointing to the maximum or
minimum element in the specified range respectively.
```

Below programs illustrate the max_element() and min_element() functions:

- **Program 1**: Using max_element() and min_element() on Vectors.

```

// C++ program to illustrate max_element()
// and min_element()

#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;

int main()
{
    vector<int> v = {10, 5, 30, 40, 90, 8};

    // returns iterator to max_element
    auto it1 = max_element(v.begin(), v.end());

    // returns iterator to min_element
    auto it2 = min_element(v.begin(), v.end());
```

```
20
21      // Print the max and min values
22      cout<<"Max Element: "<<*it1<<endl;
23      cout<<"Min Element: "<<*it2;
24
25      return 0;
26  }
27
```

Run

Output:

```
Max Element: 90
Min Element: 5
```

- **Program 2**: Using max_element() and min_element() on Arrays.

```
1
2   // C++ program to illustrate max_element()
3   // and min_element()
4
5   #include<iostream>
6   #include<algorithm>
7   #include<vector>
8
9   using namespace std;
10
11  int main()
12  {
13      int arr[] = {5, 6, 20, 90, 4, 8};
14
15      // Print the max element
16      cout<<"Max Element: "<<*max_element(arr, arr+6)<<endl;
17
18      // Print the min element
19      cout<<"Min Element: "<<*min_element(arr, arr+6);
20
21      return 0;
22  }
23
```

Run

Output:

```
Max Element: 90
Min Element: 4
```

Using max_element() and min_element() with user-defined function

We can also use the max_element() and min_element() with a user-defined function to compare values according to a specific criteria. Consider an example that we are given a set of points with X and Y coordinates and we need to find the point with maximum and minimum X accordinate.

In the above problem, we can pass a user-defined comparator function to the max_element() and min_element() function along with the range so that these functions calculates maximum and minimum on the basis of X coordinate only.

Below program illustrates this:

```
1
2   // C++ program to illustrate max_element()
3   // and min_element()
4
5   #include<iostream>
6   #include<algorithm>
7   #include<vector>
8
9   using namespace std;
10
11  // Structure to declare Points
12  struct Point{
13
14      int x;
```

```
    int y;

    Point(int i, int j)
    {
        x = i;
        y = j;
    }
};

// User defined comparator function to compare
// according to the X-coordinate only
bool myCmp(Point p1, Point p2)
{
    return (p1.x < p2.x);
}

```

Run

Output:

```
Point with max X-coordinate: (90, 10)
Point with min X-coordinate: (2, 300)
```

Time Complexity: The max_element() and min_element() takes O(N) time in the worst case as they perform a linear search to find the maximum and minimum element respectively.

Note: Both of these functions can be used with other containers also like, List, forward_list, set, unordered_sset, map, unordered_map etc.

count() in C++ STL

The **count()** function is a built-in function in C++ STL which is used to count number of occurrences of a given element in a specified range in a container.

Header File: This function is available in the "algorithm" header file.

Syntax:

```
int count(Iterator first, Iterator last, T &val)

first, last: Input iterators to the initial and final positions
of the sequence of elements.
Val : Value to match
```

Below program illustrates the use of count() function:

- **Program 1**: Using count() with Vectors.

```
// C++ program for count in C++ STL for
// a vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> vect{ 30, 20, 5, 10, 6, 10, 10 };
    cout << "Number of times 10 appears : "
         << count(vect.begin(), vect.end(), 10);

    return 0;
}

```

Run

Output:

```
Number of times 10 appears : 3
```

- **Program 2**: Using count() with Arrays.

```cpp

// C++ program for count in C++ STL for
// an array
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int arr[] = { 30, 20, 5, 10, 6, 10, 10 };

    cout << "Number of times 10 appears : "
         << count(arr, arr+7, 10);

    return 0;
}

```

Run

Output:

```
Number of times 10 appears : 3
```

We can similarly use the count() function with other containers like List, String, Set, unordered_set, map, unordered_set etc.

Time Complexity and Internal Working

The count() function internally performs a linear search between the start and end iterators passed to it as parameters and counts the number of occurrences of the given element.

Thus, it takes O(N) time complexity in worst case, where N is the total number of elements present in between the range.

binary_search() in C++ STL

The **binary_search()** is builtin function in C++ STL which is used to check whether a specific element is present or not in a container. This function works only with sorted set of data or a container in which elements are sorted.

Header File: This function is available in the "algorithm" header file.

Binary Search: This search is performed on a sorted set of data by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Syntax:

```
bool binary_search(Iterator1, Iterator2 , Value);

Iterator1: It is the begin iterator of the sorted range.
Iterator2: This iterator points to past-the-end
           element of the sorted range.
Value: This is the value to be searched for.
```

Below programs illustrate the binary_search() function:

- **Program 1**: Using binary_search on Vectors.

```cpp

// C++ program for binary_search() in C++ STL
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 30, 40, 50};
```

```
12
13      int x = 20;
14
15      if(binary_search(v.begin(), v.end(), x))
16          cout<<"Found\n";
17      else
18          cout<<"Not Found\n";
19
20      return 0;
21  }
22
```

Run

Output:

```
Found
```

- **Program 2**: Using binary_search on Arrays.

```
1
2   // C++ program for binary_search() in C++ STL
3   #include <iostream>
4   #include <vector>
5   #include <algorithm>
6
7   using namespace std;
8
9   int main()
10  {
11      int arr[] = {10, 20, 30, 40, 50};
12
13      int x = 20;
14
15      if(binary_search(arr, arr+5, x))
16          cout<<"Found\n";
17      else
18          cout<<"Not Found\n";
19
20      return 0;
21  }
22
```

Run

Output:

```
Found
```

binary_search() with user defined comparator function

We can also use the binary_search() function to search for an element in the sorted range according the a specifc criteria by providing a user defined comparator function.

We can define a user-defined comparator function, and pass it to the binary_search() function as an additional parameter based on which the binary_search function will search element.

Consider an example where we are given a set of points in a 2-D space with both X and Y coordinates. All of these points are arranged in a sorted order according to their X-coordinates. The task is to check is a given point is present in the set of points with the same X-coordinate.

Below program uses binary_search() to solve the above problem:

```
1
2   // C++ program for binary_search() in C++ STL
3   #include <iostream>
4   #include <vector>
5   #include <algorithm>
6
7   using namespace std;
8
9   // Structure for a Point
10  struct Point
```

```
{
    int x;
    int y;

    Point(int i, int j)
    {
        x = i;
        y = j;
    }
};

// User-defined comparator function to
// compare values according to X-coordinate
bool myCmp(Point p1, Point p2)
{
    return p1.x < p2.x;
}

int main()
{
```

Run

Output:

```
Found
```

Time Complexity and Internal Working

The binary_search() function internally uses the lower_bound() function to perform the search operation. Therefore, it is highly recommended to read the post or watch the tutorial of lower_bound() function first.

The binary_search() function internally calls the lower_bound() function as shown below:

```
if(it = end_iterator || (*it) != key)
   Key not present
else
   Key is present
```

The binary_search() function works in O(logN) time complexity if the container on which it is used supports random access iterators otherwise it will take O(N) time.

- fill() in C++ STL

The **fill()** in C++ is a built-in function which is used to fill values in a container. It takes a begin iterator, end iterator, and value, and fills the range between begin and end iterator [begin, end) with the given value.

Note: The fill() function fills the value at location starting from begin and upto end, that is, excluding end.

Syntax:

```
fill(Iterator begin, Iterator end, value);
```

Below programs illustrate the fill() function:

- **Program 1**: Below program fills the complete vector 5.

```
// C++ program for fill() in C++ STL
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 30, 40};

    // Fill the vector with 5
    fill(v.begin(), v.end(), 5);
```

```
14
15      for(auto x:v)
16          cout<<x<<" ";
17
18      return 0;
19  }
20
```

Run

Output:

```
5 5 5 5
```

- **Program 2**: We can also fill a part of the vector or a sub-vector as shown below:

```
1
2   // C++ program for fill() in C++ STL
3   #include <iostream>
4   #include <vector>
5
6   using namespace std;
7
8   int main()
9   {
10      vector<int> v = {10, 20, 30, 40};
11
12      // Fill the vector with 5 skipping the first
13      // element and last element
14      fill(v.begin() + 1, v.end()-1, 5);
15
16      for(auto x:v)
17          cout<<x<<" ";
18
19      return 0;
20  }
21
```

Run

Output:

```
10 5 5 40
```

- **Program 3:** Filling an array with the fill() function.

```
1
2   // C++ program for fill() in C++ STL
3   #include <iostream>
4   #include <vector>
5
6   using namespace std;
7
8   int main()
9   {
10      int arr[] = {10, 20, 30, 40};
11
12      // Fill the complete array with 5
13      fill(arr, arr + 4, 5);
14
15      for(auto x:arr)
16          cout<<x<<" ";
17
18      return 0;
19  }
20
```

Run

Output:

```
5 5 5 5
```

Note: We can also fill a part of an array similar to vector by simply passing the begin address of the range instead of the base address and past-the-end address of the range.

- **Program 4**: Filling a List. We can only fill a complete list as list doesnot allows increment and decrement operations directly on begin() and end() iterators.

```cpp

// C++ program for fill() in C++ STL
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int> l = {10, 20, 30, 40};

    // Fill the list with 5
    fill(l.begin(), l.end(), 5);

    for(auto x:l)
        cout<<x<<" ";

    return 0;
}

```

Run

Output:

```
5 5 5 5
```

- **Program 5**: Filling a string. We can also use the fill() function to fill a complete string or substring similar to that of arrays.

```cpp

// C++ program for fill() in C++ STL
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s = "geeks";

    // Fill the string with g
    fill(s.begin(), s.end(), 'g');

    cout<<s;

    return 0;
}

```

Run

Output:

```
ggggg
```

Time Complexity: The fill() function working in **O(N)** time complexity as it traverses the container between the begin and end iterators and updates each value. So, in the worst case, it will take O(N) time.

- rotate() in C++ STL

The **rotate()** function is a built-in function in C++ STL which is used to rotate the order of the elements in the range [first, last), in such a way that the element pointed by middle becomes the new first element.

Header File: The function is defined in header .

Syntax:

```
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);

first, last: Forward Iterators to the initial and final positions of
             the sequence to be rotated
middle: Forward Iterator pointing to the element within the range [first, last)
        that is moved to the first position in the range.
```

Below program illustrates the rotate() function:

- **Program 1**: Rotating a Vector.

```cpp

// C++ program for rotate() in C++ STL
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 30, 40, 50, 60};

    // Rotating the vector by 3rd position
    // Here mid iterator will be iterator
    // pointing to 3rd element: begin() + 2
    rotate(v.begin(), v.begin() + 2, v.end());

    for(auto x: v)
        cout<<x<<" ";

    return 0;
}

```

Run

Output:

```
30 40 50 60 10 20
```

- **Program 2**: Rotating an Array.

```cpp

// C++ program for rotate() in C++ STL
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};

    // Rotating the array by 3rd position
    // Here mid iterator will be iterator
    // pointing to 3rd element: arr + 2
    rotate(arr, arr + 2, arr + 6);

    for(auto x: arr)
        cout<<x<<" ";

    return 0;
}

```

Run

Output:

```
30 40 50 60 10 20
```

Note: The rotate() function can be used with other containers also which support forward iterators, like List, Deque, String etc.

Time Complexity: It works in O(N) time complexity, as it traverses the complete container to rotate it.

- accumulate() in C++ STL

The **accumulate()** is a built-in function in C++ STL which is used to find the sum of values in a given range in a container. It takes the range and an initial sum value as arguments and returns the sum of all elements in the range with the initial value passed as a parameter to the function.

This function can also be used to perform subtraction operation or any other user-defined operation also but by default, the behavior of the accumulate() function is to find the sum only.

Syntax:

```
accumulate(first, last, sum);

first: Iterator pointing to the first element of the range.
last: Iterator pointing to the past-the-end element of the range.
sum: initial value of the sum
```

Below program illustrates the accumulate() function:

- **Program 1**: Program to find sum of elements in a range [begin, end) of a container with an intial value.

```cpp

// C++ program for accumulate() in C++ STL

#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = {10, 20, 30};

    int int_res = 0;

    // Sum of all values of vector with int_res
    cout<<accumulate(v.begin(), v.end(), int_res)<<endl;

    int_res = 100;
    // Calculating sum of all values again with int_res
    cout<<accumulate(v.begin(), v.end(), int_res)<<endl;

    return 0;
}

```

Run

Output:

```
60
160
```

- **Program 2**: Program to subtract all values of vector from a given initial value. To do this we can pass an additional parameter to the accumulate function "minus" as shown in the program below.

```cpp

// C++ program for accumulate() in C++ STL

#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int main()
{
```

```cpp
    vector<int> v = {10, 20, 30};

    int int_res = 100;

    // Subtracting all values of vector by int_res
    cout<<accumulate(v.begin(), v.end(), int_res, minus<int>() );

    return 0;
}

```

Run

Output:

```
40
```

- **Program 3**: We can also use the accumulate() function with a user-defined comparator function to perform specific operation like product as shown below:

```cpp

// C++ program for accumulate() in C++ STL

#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

// User-defined comparator function
int myFun(int x, int y)
{
    return x*y;
}

int main()
{
    vector<int> v = {10, 20, 30};

    int int_res = 1;

    // Calculating product with int_res
    cout<<accumulate(v.begin(), v.end(), int_res, myFun );

    return 0;
}

```

Run

Output:

```
6000
```

Time Complexity: It works in **O(N)** time complexity as it starts with the begin iterator and goes up to the end iterator and performs the specified operation.