forward_list in C++ STL

Forward list in STL implements a singly linked list. Introduced from C++11, the forward list is useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.

It differs from the List container by the fact that forward_list keeps track of the location of only next element while list keeps track to both next and previous elements, thus increasing the storage space required to store each element. The drawback of forward list is that it cannot be iterated backwards and its individual elements cannot be accessed directly.

Forward List is preferred over the list when only forward traversal is required (same as singly linked list is preferred over doubly linked list) as we can save space. Some example cases are, chaining in hashing, adjacency list representation of a graph, etc.

Operations on Forward List : 1. assign() :- This function is used to assign values to forward list, its another variant is used to assign repeated elements.

```
1
2  // C++ code to demonstrate forward list
3  // and assign()
4  #include<iostream>
5  #include<forward_list>
6  using namespace std;
7
8  int main()
9  {
10     // Declaring forward list
11     forward_list<int> flist1;
12
13     // Declaring another forward list
14     forward_list<int> flist2;
15
16     // Assigning values using assign()
17     flist1.assign({1, 2, 3});
18
19     // Assigning repeating values using assign()
20     // 5 elements with value 10
21     flist2.assign(5, 10);
22
23     // Displaying forward lists
24     cout << "The elements of first forward list are : ";
25     for (int&a : flist1)
26         cout << a << " ";
27     cout << endl;
28
29     cout << "The elements of second forward list are : ";
30     for (int&b : flist2)
```

Run

Output:

```
The elements of first forward list are : 1 2 3
The elements of second forward list are : 10 10 10 10 10
```

2. push_front() :- This function is used to insert the element at the first position on forward list. The value from this function is copied to the space before first element in the container. The size of forward list increases by 1.

3. emplace_front() :- This function is similar to the previous function but in this no copying operation occurs, the element is created directly at the memory before the first element of the forward list.

4. pop_front() :- This function is used to delete the first element of list.

```
1
2  // C++ code to demonstrate working of
3  // push_front(), emplace_front() and pop_front()
4  #include<iostream>
5  #include<forward_list>
6  using namespace std;
7
```

```cpp
int main()
{
    // Initializing forward list
    forward_list<int> flist = {10, 20, 30, 40, 50};

    // Inserting value using push_front()
    // Inserts 60 at front
    flist.push_front(60);

    // Displaying the forward list
    cout << "The forward list after push_front operation : ";
    for (int&c : flist)
        cout << c << " ";
    cout << endl;

    // Inserting value using emplace_front()
    // Inserts 70 at front
    flist.emplace_front(70);

    // Displaying the forward list
    cout << "The forward list after emplace_front operation : ";
    for (int&c : flist)
       cout << c << " ";
```

Run

Output:

```
The forward list after push_front operation : 60 10 20 30 40 50
The forward list after emplace_front operation : 70 60 10 20 30 40 50
The forward list after pop_front operation : 60 10 20 30 40 50
```

4. insert_after() This function gives us a choice to insert elements at any position in forward list. The arguments in this function are copied at the desired position.

5. emplace_after() This function also does the same operation as above function but the elements are directly made without any copy operation.

6. erase_after() This function is used to erase elements from a particular position in the forward list.

```cpp

// C++ code to demonstrate working of
// insert_after(), emplace_after() and erase_after()
#include<iostream>
#include<forward_list>
using namespace std;

int main()
{
    // Initializing forward list
    forward_list<int> flist = {10, 20, 30} ;

    // Declaring a forward list iterator
    forward_list<int>::iterator ptr;

    // Inserting value using insert_after()
    // starts insertion from second position
    ptr =  flist.insert_after(flist.begin(), {1, 2, 3});

    // Displaying the forward list
    cout << "The forward list after insert_after operation : ";
    for (int&c : flist)
        cout << c << " ";
    cout << endl;

    // Inserting value using emplace_after()
    // inserts 2 after ptr
    ptr = flist.emplace_after(ptr,2);

    // Displaying the forward list
```

Run

Output:

```
The forward list after insert_after operation : 10 1 2 3 20 30
The forward list after emplace_after operation : 10 1 2 3 2 20 30
The forward list after erase_after operation : 10 1 2 3 2 30
```

7. remove() :- This function removes the particular element from the forward list mentioned in its argument.

8. remove_if() :- This function removes according to the condition in its argument.

```cpp

// C++ code to demonstrate working of remove() and
// remove_if()
#include<iostream>
#include<forward_list>
using namespace std;

int main()
{
    // Initializing forward list
    forward_list<int> flist = {10, 20, 30, 25, 40, 40};

    // Removing element using remove()
    // Removes all occurrences of 40
    flist.remove(40);

    // Displaying the forward list
    cout << "The forward list after remove operation : ";
    for (int&c : flist)
        cout << c << " ";
    cout << endl;

    // Removing according to condition. Removes
    // elements greater than 20. Removes 25 and 30
    flist.remove_if([](int x){ return x>20;});

    // Displaying the forward list
    cout << "The forward list after remove_if operation : ";
    for (int&c : flist)
       cout << c << " ";
```

Run

Output:

```
The forward list after remove operation : 10 20 30 25
The forward list after remove_if operation : 10 20
```

9. splice_after() :- This function transfers elements from one forward list to other.

```cpp

// C++ code to demonstrate working of
// splice_after()
#include<iostream>
#include<forward_list> // for splice_after()
using namespace std;

int main()
{
    // Initializing forward list
    forward_list<int> flist1 = {10, 20, 30};

    // Initializing second list
    forward_list<int> flist2 = {40, 50, 60};

    // Shifting elements from first to second
    // forward list after 1st position
    flist2.splice_after(flist2.begin(),flist1);

    // Displaying the forward list
    cout << "The forward list after splice_after operation : ";
    for (int&c : flist2)
```

```
        cout << c << " ";
    cout << endl;

    return 0;
}

```

Run

Output:

```
The forward list after splice_after operation : 40 10 20 30 50 60
```

Some more methods of forward_list:

- front()- This function is used to reference the first element of the forward list container.
- begin()- begin() function is used to return an iterator pointing to the first element of the forward list container.
- end()- end() function is used to return an iterator pointing to the last element of the list container.
- cbegin()- Returns a constant iterator pointing to the first element of the forward_list.
- cend()- Returns a constant iterator pointing to the past-the-last element of the forward_list.
- before_begin()- Returns a iterator which points to the position before the first element of the forward_list.
- cbefore_begin()- Returns a constant random access iterator which points to the position before the first element of the forward_list.
- max_size()- Returns the maximum number of elements can be held by forward_list.
- resize()- Changes the size of forward_list.

- List in C++ STL

Lists are sequence containers that allow non-contiguous memory allocation. So far, in sequential containers we have seen Vectors and forward_list. Vector is implemented using dynamically allocated arrays, forward_list is implemented as Singly-linked lists whereas, a **List in C++ is implemented using doubly linked lists**.

As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.

There are many advantages and additional operations available in List container such as insert at the beginning, insert at the end, remove from the front, remove from the back, which were not there in forward_list container.

Creating a List: To create a list, include the header file and use the below syntax:

```
list list_name;

Here, data_type denotes the type of data which we
want to store in our List or doubly linked list.
```

Below is a sample program to create a list and insert two elements at back and one element at front:

```

#include<iostream>
#include<list>

using namespace std;

int main()
{
    // Create a List
    list<int> l;

    // Push two elements at back
    l.push_back(10);
```

```
    l.push_back(20);

    // Push an element to front
    l.push_front(5);

    // Traverse and print elements
    for(auto x: l)
    {
        cout<<x<<" ";
    }

    return 0;
}
```

Run

Output:

```
5 10 20
```

Note: Since List are implemented using doubly linked list, both of the above operation runs in O(1) time complexity.

We may also initialize a list at the time of creating it. Below is an example to illustrate this:

```
list l = {10, 2, 5, 20};

// The above statement creates a list of type integer
// and initializes it with elements provided in braces
```

Let's now check two more functions to remove elements from front and end of a List:

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

using namespace std;

int main()
{
    // Create a List
    list<int> l = {10, 2, 5, 20};

    // Remove elements from front and back
    l.pop_front();
    l.pop_back();

    // Traverse and print elements
    for(auto itr = l.begin(); itr != l.end(); itr++)
    {
        cout<<*itr<<" ";
    }

    return 0;
}
```

Run

Output:

```
2 5
```

Note: In the above program we have used iterators to traverse the list. Since, elements in list are non-contiguous, we can not use simple for loop to traverse the elements.

Inserting elements in the List

We have already seen push_back() and push_front() functions for inserting elements to the List. But what if we want to insert an element at a

specific position. We may use the insert() function to insert elements at somewhere in between the first and last elements of a List.

The **list::insert()** function is used to insert the elements at any position of list. This function takes 3 elements, position, number of elements to insert and value to insert. If not mentioned, number of elements is default set to 1.

Syntax:

```
insert(pos_iter, ele_num, ele)
```

Parameters: This function takes in three parameters:

- **pos_iter**: Position in the container where the new elements are inserted.
- **ele_num**: Number of elements to insert. Each element is initialized to a copy of val.
- **ele**: Value to be copied (or moved) to the inserted elements.

Return Value: This function returns an iterator that points to the first of the newly inserted elements.

Below program illustrate the insert() function:

```cpp

#include<iostream>
#include<list>

using namespace std;

int main()
{
    // Create a List
    list<int> l = {10, 20, 30};

    // Iterator pointing to first element
    auto itr = l.begin();

    // Advance the Iterator to point to
    // second element
    itr++;

    // Insert 15 at second position keeping the
    // interator at second position only
    itr = l.insert(itr, 15);

    // Insert 7 two times at position 2
    l.insert(itr, 2, 7);

    // Print elemnet at front and rear end
    cout<<l.front()<<" "<<l.back();

    // Print size of the list
    cout<<" "<<l.size();
```

Run

Output:

```
10 30 6
```

Note: We have discussed some additional functions in the above program. The function front() returns element present at the front of the List, the function back() returns element present at the rear end and the function size() return the total number of elements in the List.

Deleting elements from List

We can delete elements from a List, either using erase() function or remove() function. The erase() function deletes elements present at a specific position or range whereas the remove() function deletes all occurrences of a given element from the List.

Let us learn about each of these two functions in details:

1. **erase()**: The **list::erase()** is a built-in function in C++ STL which is used to delete elements from a list container. This function can be used to remove a single element or a range of elements from the specified list container.

   Syntax:

Syntax:

```
iterator list_name.erase(iterator position)

or,

iterator list_name.erase(iterator first, iterator last)
```

Parameters: This function can accepts different parameters based on whether it is used to erase a single element or a range of element from the list container.

- **position**: This parameter is used when the function is used to delete a single element. This parameter refers to an iterator which points to the element which is need to be erased from the list container.
- **first**, **last**: These two parameters are used when the list is used to erase elements from a range. The parameter *first* refers to the iterator pointing to the first element in the range and the parameter *last* refers to the iterator pointing to the last element in the range which is needed to be erased. This erases all the elements in the range including the element pointed by the iterator *first* but excluding the element pointed by the iterator *last*.

Return Value: This function returns an iterator pointing to the element in the list container which followed the last element erased from the list container.

2. **remove()**: The **list::remove()** is a built-in function in C++ STL which is used to remove elements from a list container. It removes elements comparing to a value. It takes a value as the parameter and removes all the elements from the list container whose value is equal to the value passed in the parameter of the function.

   Syntax:

   ```
   list_name.remove(val)
   ```

   Parameters: This function accepts a single parameter *val* which refers to the value of elements needed to be removed from the list. The remove() function will remove all the elements from the list whose value is equal to val.

Below program illustrate the erase() and remove() function:

```cpp

#include<iostream>
#include<list>

using namespace std;

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

    // Iterator pointing to first element
    auto itr = l.begin();

    // Erase element pointed by itr
    itr = l.erase(itr);

    // Remove all occurrences of 40
    l.remove(40);

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

    return 0;
}

```

Run

Output:

```
20 30 20
```

Merging two Lists

We can merge two sorted Lists directly using the built-in merge() function. The **list::merge()** is an inbuilt function in C++ STL which merges two sorted lists into one. The lists should be sorted in ascending order. It merges two sorted lists into a single sorted list.

Syntax:

```
list1_name.merge(list2_name)
```

Parameters: The function accepts a single mandatory parameter list2_name which specifies the list to be merged into list1.

Below program illustrate this:

```cpp

#include <bits/stdc++.h>
using namespace std;

int main()
{
    // declaring the lists
    // initially sorted
    list<int> list1 = { 10, 20, 30 };
    list<int> list2 = { 40, 50, 60 };

    // merge operation
    list2.merge(list1);

    cout << "List:  ";

    for (auto it = list2.begin(); it != list2.end(); ++it)
        cout << *it << " ";

    return 0;
}

```

Run

Output:

```
List:  10 20 30 40 50 60
```

Time Complexity Analysis

Let's now look at time complexity of working of every function we have discussed so far. Since Lists are internally implemented using doubly linked lists, so it maintains both head and tail pointers, pointing to the first and last elements, and hence it can perform a lot of operations in O(1) time complexity.

Below table lists all of the functions we have discussed so far with their respective time complexities:

Function	Description	Time Complexity
front()	Returns element at front.	O(1)
back()	Returns element at end.	O(1)
size()	Returns size of the List.	O(1)
begin()	Returns iterator pointing to first element.	O(1)
end()	Returns iterator pointing to last element.	O(1)

erase(itr)	Erases element pointed by itr.	O(1)
push_front()	Inserts element at front.	O(1)
push_back()	Inserts element at back.	O(1)
pop_front()	Removes element from front.	O(1)
pop_back()	Removes element from end.	O(1)
reverse()	Reverses the list.	O(N)
remove()	Removes all occurrences of a particular element.	O(N)
sort()	Sorts the linked list.	O(N*logN)