We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

Classroom Theory

Forward List in C++ STL

Forward list in C++ 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 list implements doubly linked list. While forward_list keeps track of the location of only next element, the simple list keeps track to both next and previous elements, thus increasing the storage space required to store each element. The drawback of the 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.

Header File: We need to include a forward_list header file to use the forward list in any program.

Example: This program shows the implementation of the forward_list.

```

// CPP program to implement
// doubly linked list
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{

    // Initialising a forward list
    forward_list<int> l = { 10, 15, 20 };

    // Pushing elements into the list
    l.push_front(5);

    // Pushing elements into the list
    l.push_front(3);

    // Popping out elements from list
    l.pop_front();

    // Displaying the list
    for (int x : l)
        cout << x << " ";
    return 0;
}

```

Run

Output:

```
5 10 15 20
```

Working:

1.
```
forward_list l = {10, 15, 20};
```
This creates a forward_list which is a singly-linked list with elements 10, 15 and 20. If no elements are passed into the list, then an empty list is created. The list looks like:
```
10 -> 15 -> 20
```

2.
```
l.push_front(5);
```
This adds an element at the front of the existing list. Now the list looks like:
```
5 -> 10 -> 15 -> 20
```

3.
```
l.push_front(3);
```
This adds an element at the front of the existing list. Now the list looks like:
```
3 -> 5 -> 10 -> 15 -> 20
```

4.
```
l.pop_front()
```
;
This removes the element at the front of the list. So the list will look like:
```
5 -> 10 -> 15 -> 20
```

5.
```
for(int x : l)

    cout << x << " ";
```
This prints the forward_list. The concept is similar to any display of containers. This will print the list as:
```
5 10 15 20
```

Important functions in forward_list

1. **assign() (https://www.geeksforgeeks.org/forward_list-assign-function-in-c-stl/)** : This function is used to assign values to forward list, its another variant is used to assign repeated elements.

2. **remove() (https://www.geeksforgeeks.org/forward_listremove-forward_listremove_if-c-stl/)**: This function removes all the elements from the forward list that is mentioned in its argument.

Example: The following program shows the implementation of the above two mentione functions.

```

// CPP program to implement
// working of assign() and remove()
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{

    // Declaring a forward list
    forward_list<int> l;

    // Assigning values to the list
    l.assign({ 10, 20, 30, 10 });

    // Removing elements from list
    l.remove(10);

    // Displaying the list
    for (auto it = l.begin(); it != l.end(); it++)
        cout << (*it) << " ";

    return 0;
}
```

Run

LIVE BATCHES

Output:

```
20 30
```

Working:

1. `forward_list l;`

   This creates an empty forward_list which is a singly-linked list with no elements.

2. `l.assign({10, 20, 30, 10});`

   This assigns 4 elements to the list. Now the list looks like:

   ```
   10 -> 20 -> 30 -> 10
   ```

3. `l.remove(10);`

   This removes all the 10's from the list. Now the list looks like:

   ```
   3 -> 5 -> 10 -> 15 -> 20
   ```

4. `l.pop_front()`

   ;
   This removes the element at the front of the list. So the list will look like:

   ```
   20 -> 30
   ```

5.
   ```
   for(auto it = l.begin(); it != l.end(); it++)

           cout << (*it) << " ";
   ```

   This prints the forward_list. The concept is similar to any display of containers. This will print the list as:

   ```
   20 30
   ```

Two variations of assign() function:

a. **Program 1:** The assign() function can also be used to assign one forward_list to another. This is implemented in the following way:

```

// CPP program to implement
// working of assign() and remove()
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{

    // Declaring a forward list
    forward_list<int> l;

    // Assigning values to the list
    l.assign({ 10, 20, 30, 10 });

    // Declaring another forward list
    forward_list<int> l2;

    // Assigning one list to another
    l2.assign(l.begin(), l.end());

    // Displaying the second list
    for (auto it = l2.begin(); it != l2.end(); it++)
```

```
        cout << (*it) << " ";

    return 0;
}

```

Run

Output:

```
10 20 30 10
```

b. **Program 2:** This is another way of using assign() function where two parameters are passed. This function assigns n number of elements to the forward_list and each element is initialised with value val.

Syntax:

```
assign(int n, val)
```

```

// CPP program to implement
// working of assign() and remove()
#include <forward_list>
#include <iostream>
using namespace std;

int main()
{

    // Declaring a forward list
    forward_list<int> l;

    // Assigning values to the list
    l.assign(5, 10);

    // Displaying the second list
    for (auto it = l.begin(); it != l.end(); it++)
        cout << (*it) << " ";

    return 0;
}

```

Run

Output:

```
10 10 10 10 10
```

3. **insert_after() (https://www.geeksforgeeks.org/forward_list-insert_after-function-in-c-stl/)**: 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. Look into the linked article to get more details.

4. **emplace_after() (https://www.geeksforgeeks.org/forward_list-emplace_after-and-emplace_front-in-c-stl/)** This function also does the same operation as above function but the elements are directly made without any copy operation. For a large object of data the emplace_after() function is more optimised than the insert_after() function.

5. **erase_after() (https://www.geeksforgeeks.org/forward_listclear-forward_listerase_after-c-stl/)** This function is used to erase elements from a particular position in the forward_list.

Example: This program shows the implementation of the above mentioned functions.

```
// CPP program to implement working of above mentioned functions
#include <forward_list>
#include <iostream>
using namespace std;
int main(){
    // Declaring a forward list
    forward_list<int> l1 = { 15, 20, 30 };
    // Using insert_after() function
    // to insert elements to the list
    // at a particular place
    auto it = l1.insert_after(l1.begin(), 10);
    // inserting a set of elements to the list
    it = l1.insert_after(it, { 2, 3, 5 });
    // The function is similar to the insert()
    // function
    it = l1.emplace_after(it, 40);
    // Removes an element from the forward_list
    it = l1.erase_after(it);
    // Displays the elements in the forward_list
    for (int x : l1)
        cout << x << " ";
    // Removes an element from it till end()
    it = l1.erase_after(it, l1.end());
    // Displays the elements in the forward_list
    for (int x : l1)
        cout << x << " ";

    return 0;
}
```

Run

Output:

```
15 10 2 3 5 40 30 15 10 2 3 5 40 30
```

Working:

1. `forward_list l1 = {15, 20, 30};`

   This creates a forward_list with the following elements:

   ```
   15 -> 20 -> 30
   ```

2. `auto it = l1.insert_after(l1.begin(), 10);`

   This inserts the element 10 after the l1.begin() position i.e., after 15. The element 10 is inserted at (l1.begin() + 1) position. The iterator it, returns the position of the inserted elements. Here it points to the recently entered element 10. The list now looks like this:

   ```
   15 -> 10 -> 20 -> 30
   ```

3. `it = l1.insert_after(it, {2, 3, 5});`

   This inserts the set of elements {2, 3, 5} after the position pointed by it i.e., after 10. The iterator it, returns the position of the inserted elements. Here it points to the recently entered element 5. The list now looks like this:

   ```
   15 -> 10 -> -> 2 -> 3 -> 5 -> 20 -> 30
   ```

4. `it = l1.emplace_after(it, 40);`

   This inserts the element 40 after the position pointed by it i.e., after 5. The iterator it is pointing to element 40. Now the list will look like:

   ```
   15 -> 10 -> -> 2 -> 3 -> 5 -> 40 -> 20 -> 30
   ```

5. `it = l1.erase_after(it);`

   This deletes the element next to the element pointed by the iterator. As the iterator was pointing to 40, so the next element to 40 i.e., 20 gets deleted. Now the list will look like:

   ```
   15 -> 10 -> -> 2 -> 3 -> 5 -> 40 -> 30
   ```

A variation of erase_after() function: Program: The erase_after() function can take two parameters as a range and delete all the elements in-between. This function returns the address to the element after the last removed element from the list. In the following program the it iterator would point to the element after 30 i.e., l1.end().

```
// CPP program to implement working of above mentioned functions
#include <forward_list>
#include <iostream>
using namespace std;

int main(){

    // Declaring a forward list
    forward_list<int> l1 = { 15, 20, 30 };

    // Using insert_after() function
    // to insert elements to the list
    // at a particular place
    auto it = l1.insert_after(l1.begin(), 10);

    // inserting a set of elements to the list
    it = l1.insert_after(it, { 2, 3, 5 });

    // The function is similar to the insert()
    // function
    it = l1.emplace_after(it, 40);

    // Removes all elements from it till end()
    it = l1.erase_after(it, l1.end());

    // Displays the elements in the forward_list
    for (int x : l1)
        cout << x << " ";
    return 0;
}
```

Run

Output:

```
15 10 2 3 5 40
```

6. **clear() (https://www.cdn.geeksforgeeks.org/forward_listclear-forward_listerase_after-c-stl/):** This function is used to remove all the elements of the forward list container, thus making its size 0.
Syntax:

```
forwardlistname.clear()
```

Parameters: No parameters are passed.
Result: All the elements of the forward list are removed (or destroyed)
Examples:

```
Input  : flist{1, 2, 3, 4, 5};
         flist.clear();
Output : flist{}

Input  : flist{};
         flist.clear();
Output : flist{}
```

7. **empty() (https://www.cdn.geeksforgeeks.org/forward_listfront-forward_listempty-c-stl/):** This function returns a boolean value indicating whether the forward_list is empty, i.e. whether its size is 0 or not. **Syntax:**

```
forwardlistname.empty()
```

Parameters: No parameters are passed.
Returns: True, if the list is empty else false.
Examples:

```
Input  : forward_list forwardlist{1, 2, 3, 4, 5};

         forwardlist.empty();

Output : False



Input  : forward_list forwardlist{};

         forwardlist.empty();

Output : True
```

8. **reverse() (https://www.cdn.geeksforgeeks.org/forward_listreverse-in-c-stl/):** This is an inbuilt function in CPP STL which reverses the order of the elements present in the forward_list.
   Syntax:

   ```
   forwardlist_name.reverse()
   ```

   Parameter: The function does not accept any parameter.
   Return value: The function has no return value. It reverses the forward list.
   Example:

   ```
   Input  : forward_list l = {10, 20, 30};
            l.reverse();

   Output : {30, 20, 10}
   ```

9. **merge() (https://www.geeksforgeeks.org/forward_list-merge-in-c-stl/):** This is an inbuilt function in C++ STL which merges two sorted forward_lists into one. This function merges the second list into the first, thus emptying the second list.
   The merge() function can be used in two ways:
   - Merge two forward lists that are sorted in ascending order into one.

   - Merge two forward lists into one using a comparison function.

   Syntax:

   ```
   forwardlist_name1.merge(forward_list& forwardlist_name2)
                     or
   forwardlist_name1.merge(forward_list& forwardlist_name2,
                           Compare comp)
   ```

   Parameters: The function accepts two parameters which are specified as below:
   forwardlist_name2 – Another forward list of the same type which is to be merged
   comp – A comparison function which should return true or false.

   Return value: The function does not return anything.

   Example:

   ```
   Input : forward_list l1 = {10, 20, 30};
           forward_list l2 = {5, 15};

   Operation : l1.merge(l2)

   Output : l1 = {5, 10, 15, 20, 30}
            l2 = {}
   ```

10. **sort() (https://www.geeksforgeeks.org/stdforward_listsort-c-stl/)**: This function is used to sort the elements of the container by changing their positions.
    Syntax:

    ```
    forwardlistname.sort()
    ```

Parameters: No parameters are passed.
Result: The elements of the container are sorted in ascending order.
Examples:

```
Input  : forward_list l1 = {5, 15, 10};

         l1.sort();

Output : {5, 10, 15}



Input  : myflist{"This", "is", "Geeksforgeeks"};

         myflist.sort();

Output : Geekforgeeks, This, is
```

Note: There is another variation of merge() function where it can take a parameter to determine the order of sorting the forward_list.

Time Complexities: Here are the time complexities of few important functions implemented over a forward_list.

1. insert_after(): To insert one element it takes O(1) time and to insert m elements, it will take O(m) time.
2. erase_after(): To erase one element it takes O(1) time and to erase m elements, it will take O(m) time.
3. push_front(): This takes O(1) time.
4. pop_front(): This takes O(1) time.
5. reverse(): This takes O(n) time.
6. sort(): This takes O(nlogn) time.
7. remove(): This takes O(n) time.
8. assign(): To assign one element it takes O(1) time and to assign m elements, it will take O(m) time.

- 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 <list> and use the below syntax:

```
list<data_type> 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<int> 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:

```
#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();
    return 0;
}
```

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:

   ```
   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.

LIVE BATCHES

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

```

#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:

```

#include <bits/stdc++.h>
```

```
#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

LIVE BATCHES

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)

Report An Issue

If you are facing any issue on this page. Please let us know.

GeeksforGeeks

▲