Introduction to STL in C++

Benefits of Using STL in C++ :

Before diving into the specifics of STL, let's take a moment to understand why it is beneficial to use the Standard Template Library in C++.

- **Reusability :** The STL offers a set of versatile and reusable components, allowing developers to save time and effort by utilizing well-tested implementations of data structures and algorithms.
- **Efficiency:** STL components are designed to be efficient in terms of both time and memory. They are often implemented using optimized algorithms, leading to faster program execution.
- **Maintainability :** By using the STL, code becomes more maintainable and readable, as the library provides clear and standardized interfaces for common operations.
- **Portability:** Since STL is a part of the C++ standard, it is supported by all major C++ compilers and platforms, making code written with STL highly portable.

Components of STL in C++ :

In general, There are four components of STL in C++ :

- Algorithms
- Function
- Containers
- Iterators

Algorithms :

AlgorithmsA group of functions that are specifically created to be used on a variety of items are defined by the header algorithm. They influence containers and provide the contents of the containers the means for a variety of functions.

STL offers a wide range of algorithms that operate on containers. These algorithms provide functionalities like sorting, searching, and modifying container elements.

- **Sorting Algorithms**

Sorting algorithms arrange elements in a specific order, such as ascending or descending. Examples of sorting algorithms in STL include **std::sort**, **std::stable_sort**, and **std::partial_sort**.

- **Searching Algorithms**

Searching algorithms help find specific elements within containers. **std::find**, **std::binary_search**, and **std::lower_bound** are some commonly used search algorithms.

- **Modifying Algorithms**

Modifying algorithms alters the content of containers. Examples include **std::copy**, **std::transform**, and **std::fill**.

- **Numeric Algorithms**

Numeric algorithms perform operations on numeric elements within containers. Some of these algorithms include **std::accumulate**, **std::inner_product,** and **std::iota**.

Functions :

Function Definition Classes that overload the function call operator are included in the STL. These classes instances are referred to as function objects or functions. With the use of given arguments, functions enable the associated function's operation to be modified.

Containers :

Containers Definition Objects and data are kept in containers or container classes. There are only seven header files that give access to these "first-class" containers or container adaptors out of a total of seven "first-class" container classes and three container adaptor classes in the standards.

Types of Containers :

Following are the types of containers available in STL :

- Sequence Containers : Vector, List, Deque, Arrays
- Containers Adaptors : Queue, Priority Queue, Stack
- Associative Containers : Set, Multiset, Map, Multimap
- Unordered Associative Containers : Unordered Set, Unordered Multiset, Unordered Map, Unordered Multimap

Iterators :

Iterators Definition Iterators are tools for processing a list of values. They are the primary aspect of STL that permits generality.

Advantages of STL in C++ :
Programming style may be drastically altered by STL, enabling for more strong, reliable, and reusable code. By using STL, programmer may simplify their life and increase its efficiency. Additionally, STL is expandable, enabling the programmers to include their own containers and algorithms.

How to Include STL in C++ Code
To use STL in your C++ code, you need to include the appropriate header files that correspond to the components you want to use. For example, to work with vectors, you would include the < **vector >** header: #include int main() { // Your code here return 0; }

```
#include<vector>

int main() {
// Your code here
return 0;
}
```

Examples of Using STL in C++
Example 1: Using a Vector

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

int main() {
   std::vector numbers = {5, 2, 8, 1, 9};
   std::sort(numbers.begin(), numbers.end());

   for (const auto& num : numbers) {
      std::cout << num << " ";
   }

   return 0;
}
```

Sort in C++ STL

Sorting Algorithms in C++ STL
C++ STL offers multiple sorting algorithms to cater to different use cases. Let’s explore some of the most commonly used ones:

- **std::sort()**

The `std::sort()` function is the most basic sorting algorithm in C++ STL. It uses a variant of the intro sort algorithm, which combines quicksort, heapsort, and insertion sort. The efficient algorithm often outperforms other sorting techniques for most data sets.

- **std::stable_sort()**

If the order of equal elements needs to be preserved during sorting, `std::stable_sort()` is the ideal choice. It employs a stable sorting algorithm, typically based on merge sort, ensuring the relative order of equivalent elements remains unchanged.

- **std::partial_sort()**

The `std::partial_sort()` function partially sorts a container by placing the first N elements in sorted order. The remaining elements are not guaranteed to be in any specific order but will be greater than the sorted part.

- **std::nth_element()**

When finding the N-th element in a container without thoroughly sorting it, `std::nth_element()` comes in handy. It rearranges the elements such that the N-th element is sorted, with all aspects before it being less than or equal and all elements after it being greater than or equal.

- **std::make_heap() and std::sort_heap()**

These functions help create and sort heaps, respectively. A heap is a specialized data structure that allows efficient access to the maximum or minimum element.

Syntax of Sort() :

```
sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

The function doesn’t produce a result. Just the items/elements from the first to the last iterations are updated. The third parameter, comp, must be a function that establishes the hierarchy of the components to be sorted in. If nothing else is supplied, the sorting happens in ascending order.

Example of Sorting in Ascending Order :

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

int main() {
    vector arr;
    arr.push_back(34);
    arr.push_back(70);
    arr.push_back(20);
    arr.push_back(56);
    arr.push_back(1);

     cout<<"The element of array before sorting : ";
    for(int i=0;i < arr.size();i++){
         cout<< arr[i]<<" ";
    }
    cout<< endl;
    // sort function to sorting the elements
    sort(arr.begin(), arr.end());
    cout<<"The element of array after sorting : ";
    for(int i=0;i < arr.size();i++){
         cout<< arr[i]<<" ";
    }

    return 0;
}
```

Searching in STL C++

ow Binary Search Work:
Binary search operates by repeatedly dividing the search space in half, comparing the middle element with the target value, and narrowing down the search to the appropriate subarray. This process continues until the target element is found or the search space is exhausted. **Time Complexity :** O(log n)

Implementing Binary Search in C++ using STL.
Syntax of binary_search() :

```
binary_search(start_address, end_address, element to be find);
```

This algorithm’s primary principle is to divide an array in half repeatedly (divide and conquer) until either the element is located or all the elements have been used up. It operates by comparing the middle item in the array with our target; if they match, it returns true; if they don’t, the left sub-array is searched. The search is carried out in the appropriate sub-array if the middle term is less than the target element.

Array must be sorted to apply binary search in stl

Example of Searching in STL :

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

int main() {
    int arr[] = {2,4,3,43,4,35,67,99,85,34};
    int n = sizeof(arr)/sizeof(arr[0]);
    int a = 43;
    sort(arr, arr+n);
    if(binary_search(arr, arr+n, a)){
        cout<<"Element " << a<<" is present in the array";
    }
    else{
        cout<<"Element " << a <<" is not found in the array";
    }
    return 0;
}
```

lower_bound() and upper_bound() Functions

STL provides two essential functions for performing binary search on sorted containers: **lower_bound()** and **upper_bound().**

The lower_bound() Function

The **lower_bound()** function returns an iterator pointing to the first element in the container that is not less than the target value. It effectively identifies the range's lower bound where the target element could be present.

Lower Bound Example :

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

int main() {
    std::vector nums = {1, 2, 2, 3, 4, 5, 5, 5, 6, 7};

    // Binary search requires the input range to be sorted.
    std::sort(nums.begin(), nums.end());

    int target = 5;

    // Using lower_bound to find the first occurrence of the target.
    std::vector::iterator lower = std::lower_bound(nums.begin(), nums.end(), target);

    if (lower != nums.end() && *lower == target) {
        std::cout << "Lower bound of " << target << " is at index: " << std::distance(nums.begin(),
lower) << std::endl;
    } else {
        std::cout << target << " not found in the vector." << std::endl;
    }

    return 0;
}
```

The upper_bound() Function :

On the other hand, the **upper_bound()** function returns an iterator pointing to the first element in the container that is greater than the target value. It helps identify the upper bound of the range.

Upper Bound Example:

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

int main() {
    std::vector nums = {1, 2, 2, 3, 4, 5, 5, 5, 6, 7};

    // Binary search requires the input range to be sorted.
    std::sort(nums.begin(), nums.end());

    int target = 5;

    // Using upper_bound to find the first element greater than the target.
    std::vector::iterator upper = std::upper_bound(nums.begin(), nums.end(), target);

    if (upper != nums.end()) {
        std::cout << "Upper bound of " << target << " is at index: " << std::distance(nums.begin(),
upper) << std::endl;
    } else {
        std::cout << "There is no element greater than " << target << " in the vector." << std::endl;
    }

    return 0;
}
```

Use Cases of Binary Search in STL C++

- **Searching Elements in Sorted Containers**

The primary use case of binary search is locating specific elements in sorted containers. This proves particularly valuable when dealing with extensive datasets where linear search would be highly inefficient.

- **Finding Frequency of Elements**

Binary search can also be used to find the frequency of a particular element in a sorted container. By utilizing **lower_bound()** and **upper_bound(),** we can determine the count of occurrences efficiently.

Pair in STL C++

std::pair is a template class provided by the STL that allows us to bundle two different data elements together into a single unit. This powerful feature is incredibly versatile and finds its applications in various programming scenarios. Whether you need to associate two

variables, represent key-value pairs, or return multiple values from a function, std::pair **comes to the rescue.**

Creating and Initializing std::pair
To create a std::pair, we must include the &ltutility&gt header in our C++ code. Initializing a pair can be achieved using different methods, such as the `make_pair` function or the brace initialization syntax.

Accessing Elements in a std::pair
Once we have a pair, we need to access its elements. This can be accomplished using the **first** and **second** member functions. Understanding how to access these elements is crucial for effectively utilizing **std::pair**.

Syntax of Pair :

```
pair< data_type, data_type> pair_name;
```

Pair is a container that is given by the "utility" library and is declared in C++ by using the keyword "pair". In essence, a pair is used to combine two components or values into one, allowing for the storage of things with various data types or two heterogeneous objects as a single entity. Only two components can be stored in the pair container, the first of which can only be referred by "first" and the second of which can only be stored in "second".

Functions in Pair :

Function	Working
make_pair()	store two values in the pair.
swap()	swap the components of pair 1 and pair 2.
tie()	unpacks the pair values into independent variables

Example of Pair in STL :

```
#include<iostream
#include<bits/stdc++.h>
using namespace std;
int main() {
   pair<int, int> p1;
   pair<string, int> p2("PrepInsta",1);
   p1.first = 2;
   p1.second = 3;

   cout<<"The elements in pair p1 : ";
   cout<< p1.first<<" "<< p1.second<< endl;
```

```
    cout<<"The elements in pair p2: ";
    cout<< p2.first<<" "<< p2.second;

    return 0;
}
```

Vector in STL C++

A vector in C++ is a sequence container that represents a dynamic array. Unlike traditional arrays with fixed sizes, vectors can change their size during runtime. This feature makes vectors a valuable asset for managing collections of data whose size may vary.

Vector in STL

In C++, vectors are not ordered. Iterators make it simple to retrieve and navigate through vector items that are stored in nearby storage.Data is added at the end of vectors when the push back() method is used. As there may occasionally be a need to expand the vector, adding an element to the end of the vector requires differential time, but adding an element to the beginning or centre of the vector just requires linear time. Due to the absence of any scaling, removing the last piece just requires constant time.

Initializing a Vector

To use a vector, you must include the header file in your C++ program. Then, you can declare a vector of a specific data type.

For example:

```
#include <vector>

std::vector myVector; // Creates an empty vector of integers
```

Adding Elements to a Vector

One of the primary reasons for using vectors is to store and manipulate a collection of elements. You can add elements to a vector using the **push_back()** function, which appends elements at the end of the vector:

```
myVector.push_back(10); // Adds the integer 10 to the vector
myVector.push_back(20); // Adds the integer 20 to the vector
```

Accessing Elements in a Vector

You can access elements in a vector using the subscript operator **[]**. Remember that the first element is at index 0:

```
int firstElement = myVector[0]; // Accesses the first element
int secondElement = myVector[1]; // Accesses the second element
```

Vector Size and Capacity

Vectors maintain two essential properties: **size** and **capacity**.

The size represents the number of elements currently stored in the vector, while the capacity represents the total number of elements the vector can hold before resizing is required.

```
size_t size = myVector.size(); // Retrieves the current size of the vector
size_t capacity = myVector.capacity(); // Retrieves the current capacity of the vector
```

Resizing a Vector

If you want to change the size of the vector manually, you can use the **resize()** function:

```
myVector.resize(5); // Resizes the vector to hold 5 elements
```

Removing Elements from a Vector

You can remove elements from a vector using the **pop_back()** function, which eliminates the last element:

```
myVector.pop_back(); // Removes the last element from the vector
```

Vector Iterators

To traverse through a vector, you can use iterators, which act as pointers to elements within the vector:

```
std::vector::iterator it;
for (it = myVector.begin(); it != myVector.end(); ++it) {
   // Access and manipulate the elements using the iterator
}
```

Sorting a Vector

Sorting a vector in C++ is straightforward using the sort() function from the &ltalgorithm&gt header:

```
#include <algorithm>
std::sort(myVector.begin(), myVector.end()); // Sorts the elements in ascending order
```

Vector Capacity Management

Vectors automatically manage their capacity, but you can explicitly request a capacity change using the **reserve()** function:

```
myVector.reserve(100); // Reserves space for 100 elements
```

Functions in Vector :

Function	Working
begin()	gives back an iterator pointing to the vector’s first element.

Function	Working
end()	gives back an iterator pointing to the vector’s last element.
size()	gives the number of elements in vector
push_back()	Add an element in the last of the vector
empty()	return whether the vector is empty or not

Example of Vector in STL :

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

int main() {
   vector< int >vect;
   // adding elements to the vector
   vect.push_back(2);
   vect.push_back(5);
   vect.push_back(10);
   vect.push_back(12);

   //size of vector
   cout<< "size of vector "<< vect.size()<< endl;

   //printing elements of vector
   cout<<"The elements of vector : ";
   for(int i=0;i< vect.size();i++){
       cout<< vect[i]<<" ";
   }

    return 0;
}
```

List in STL C++

List in STL is defined as the container used to store elements in the contiguous manner and allow insertion and deletion at any position.

About List STL :

In STL, List store the elements on a contiguous memory, but vector stores on a non-contiguous memory. List is a contiguous container, whereas vector is a non-contiguous container. As it takes a long time to move all the elements, insertion and deletion in the midst of the vector are quite expensive. This issue is solved by linklist, which is implemented using a list container. The list offers an effective method for insertion and deletion operations and facilitates bidirectional communication. List traversal is longer because elements are read sequentially, whereas a vector allows for random access.

Functions in List :

Function	Working
push_back()	insert an element to the back end of the list.
push_front()	insert an element to the front end of the list.
size()	give the number of elements present in the list.
empty()	tells whether thelist is empty or not.
pop_front()	remove the element from front of the list.

Basic Operations with List

Creating a List

To use the C++ list STL, include the &ltlist&gt header file and declare a list object. For example:

```
#include <list>
std::list<int> myList;
```

Adding Elements

Elements can be added to the list using the push_back() and push_front() methods for adding to the end and beginning, respectively.

```
myList.push_back(42);
myList.push_front(10);
```

Removing Elements

Use the pop_back() and pop_front() methods to remove elements from the list.

```
myList.pop_back();
myList.pop_front();
```

Accessing Elements

Access elements using iterators. Here’s an example of iterating through the list:

```
for (std::list < int >::iterator it = myList.begin(); it != myList.end(); ++it) {
    std::cout << *it << " ";
}
```

Advanced Operations with List

Sorting

The list can be sorted using the sort() method from the algorithm header.

```
 myList.sort();
```

Merging Lists

Merging combines two sorted lists into a single sorted list.

```
std::list < int > newList;
myList.merge(newList);
```

Reversing

Reverse the elements in the list using the reverse() method.

```
myList.reverse();
```

Example 2: Merging Sorted Lists

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

int main() {
    std::list list1 = {1, 3, 5};
    std::list list2 = {2, 4, 6};

    list1.sort();
    list2.sort();

    list1.merge(list2);

    for (const auto& num : list1) {
        std::cout << num << " ";
    }

    return 0;
}
```

Deque in STL C++

Deque in STL is defined as the container used to insert and remove elements at both ends.

They are implementing using queue data structure.

Introduction to Deque in C++

Deque, short for “Double-Ended Queue,” is an essential data structure in C++ provided by the Standard Template Library (STL). It shares similarities with vectors and lists but offers additional benefits due to its versatile nature. A Deque allows elements to be inserted or removed from both ends efficiently, making it an excellent choice for various applications.

Advantages of using Deque:

Dynamic Size: Unlike arrays, Deque can change its size dynamically, making it more flexible in handling varying amounts of data.

Fast Insertions and Deletions: Deque enables fast insertions and deletions at both ends, providing efficient data manipulation.

Random Access: Like vectors, Deque allows for direct access to elements using indexing, ensuring quick element retrieval.

Memory Efficiency: Deque optimizes memory utilization, making it an ideal choice for large-scale applications.

Functions in Deque :

Function	Working
push_back()	insert an element to the back end of the deque.
push_front()	insert an element to the front end of the deque.
size()	give the number of elements present in the deque.
empty()	tells whether the deque is empty or not.
max_size()	return the maximum number of elements can deque hold.

Basic Operations on Deque

Initializing a Deque:

To use a Deque, we need to include the header file in our C++ program. We can then create a Deque object using its constructor.

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

int main() {
    std::deque<int> myDeque; // Initializing an empty integer Deque
    return 0;
}
```

Insertion and Deletion of Elements:

Deque supports several methods for inserting and removing elements. Some commonly used functions include **push_back()**, **push_front()**, **pop_back()**, and **pop_front()**. Insertion and Deletion of Elements:

```
#include <iostream>
#include <deque>
int main() {
    std::dequemyDeque;

    myDeque.push_back(10);    // Insert 10 at the end
    myDeque.push_front(5);    // Insert 5 at the beginning

    myDeque.pop_back();       // Remove the last element
    myDeque.pop_front();      // Remove the first element

    return 0;
}
```

Accessing Elements in Deque:

Elements in a Deque can be accessed using the subscript operator **[]** or the **at()** function.

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

int main() {
    std::deque<int>myDeque;

    myDeque.push_back(10);
    myDeque.push_back(20);
    myDeque.push_back(30);

    int elementAtIndex1 = myDeque[1]; // Access the element at index 1 (20)
    int elementAtIndex2 = myDeque.at(2); // Access the element at index 2 (30)

    return 0;
}
```

Implementation of Deque in STL

The C++ Standard Template Library (STL) provides a rich set of classes and functions to work with data structures like Deque.

Code example of Deque implementation:

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

```
int main() {
    std::deque<int>myDeque;

    myDeque.push_back(10);
    myDeque.push_back(20);
    myDeque.push_front(5);

    std::cout << "Front element: " << myDeque.front() << std::endl; // Output: Front element: 5
    std::cout << "Back element: " << myDeque.back() << std::endl;   // Output: Back element: 20

    return 0;
}
```

Use Cases of Deque in C++

Deque finds applications in various scenarios due to its unique characteristics. Real-world scenarios for Deque:

Task Scheduling: Deque can be used in task scheduling algorithms, where new tasks can be added or removed efficiently from both ends.

Sliding Window Technique: Deque is commonly used to solve problems involving sliding window algorithms, where elements move in a fixed-size window.

Undo/Redo Operations: Deque's ability to insert and remove elements from both ends makes it suitable for implementing undo and redo functionalities in applications.

Queue in STL C++

Queue in STL is defined as the container used first in first out (FIFO) principle to inserted elements at the back end and deleted from the front.

Syntax of Queue :

```
queue< data_type> queue_name;
```

In the STL, queues are dynamic storage containers that are implemented as memory-resident queue data structures. They operate according to the FIFO (first in first out) principle, which states that the last element will be the final one to be removed. In a queue, elements are kept together in a continuous way. The head of the queue is used for insertion, while the back of the queue is always used for deletion.

Functions in Queue :

Function	Working
push()	insert an element at back end of the queue.
front()	give the first element of the queue.
size()	give the number of elements present in the queue
empty()	tells whether the queue is empty or not.
pop()	remove the elements from the front of queue.

Example of Queue in STL :

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
int main() {
    queue qq;
    qq.push(1);
    qq.push(2);
    qq.push(3);
    qq.push(5);

    //diplaying queue elements
    cout<<"The elements in the queue : ";
    while(!qq.empty()){
        cout<< qq.front()<<" ";
        qq.pop();
    }

    return 0;
}
```

Priority Queue in STL C++

Priority Queue in STL is defined as the adapter container used to store elements in the particular order either in increasing or decreasing depend on the implementations.

Syntax of Priority Queue :

```
priority_queue< data_type> priority_queue_name;
```

The first element of a priority queue is either the largest or the smallest of all the elements in the queue, and the elements are arranged in a non increasing order . Priority queues are a type of container adapter. However, the highest element is always the default in the C++ STL. We may alternatively put the smallest component at the top instead. Priority queues are constructed using an internal array or vector and are created on top of the heap to its maximum height.

Functions in Priority Queue :

Function	Working
push()	insert an element at back end of the priority queue.
top()	give the top element of the priority queue.
size()	give the number of elements present in the priority queue
empty()	tells whether the priority queue is empty or not.
pop()	remove the top elements from priority queue.

Example of Priority Queue in STL :

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

int main() {
    priority_queue< int>pq;
    pq.push(34);
    pq.push(20);
    pq.push(4);
    pq.push(1);
    pq.push(50);
    cout<<"The element at the top of the priority queue : "<< pq.top()<< endl;
    pq.pop();
    cout<<"The element at the top of the priority queue after remove one element : "<< pq.top();

    return 0;
}
```

Set in STL C++

Set in STL is defined as the container used to store unique elements in the particular order either increasing or decreasing order.

Syntax of Set :

```
set< data_type> set_name;
```

In STL, each value in the set serves as a key and the set doesn't enable indexing, each element of the set is distinct, meaning that no duplicate values may be placed in it. Since there can only be one index, the elements/values (keys) themselves are the indexes. Additionally, just the keys and values must be used to access the values in the set. The components of the Set are likewise kept in sorted order.

In logarithmic time complexity, the element in the set may be added, deleted, and searched. As soon as an element is added to a set, it becomes a constant and cannot be changed (its value cannot be altered). The binary search tree implements the set STL internally in C++ (Red-Black Tree).

Functions in Set :

Function	Working
insert()	insert an element to the set.
clear()	remove all the element of the set.
size()	give the number of elements present in the set.
empty()	tells whether the set is empty or not.
max_size()	return the maximum number of elements set can hold.

Example of Set in STL :

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

int main() {
   set< int> st;
   st.insert(5);
   st.insert(5);
   st.insert(1);
   st.insert(1);
   st.insert(5);
   cout<<"The elements in the set : ";
```

```
    for(auto it : st){
        cout<< it<<" ";
    }

    return 0;
}
```

Map in STL C++

Map in STL is defined as the container used to store the elements as the key-value pair. Keys will always be unique in the map

Syntax of Map :

```
map< data_type1, data_type2> map_name;
```

In STL, An associative container known as a Map is used to store objects in mapped form.
Every object on the map is made up of a key-value pair and an associated value.
The same key values cannot be shared by two mapped values.The key values work well for classifying and single-handedly recognising components.
The mapped values are used to store the key-related material. Although the types of the two may be different, the member type nonetheless combines them using a pair type that combines both.

Functions in Map :

Function	Working
insert({})	insert a pair of key – value to the map
count()	return the number of matches element from the map.
size()	give the number of keys present in the map.
empty()	tells whether the map is empty or not.
erase(const it)	remove the element at the position pointed by iterator.

Example of Map in STL :

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

int main() {

    map< int,int> mpp;
    mpp[1] = 10;
    mpp[2] = 20;
    mpp[3] = 30;
    mpp.insert({4,40});
    mpp.insert({5,50});
    cout<<"The key value pair in map are : "<< endl;
    for(auto it : mpp){
        cout<< it.first << " "<< it.second<< endl;
    }
    return 0;
}
```

Map vs Unordered Map in STL in C++

Map in STL :

DefinitionMap in stl is defined as the container which store the elements in key-value pair in sorted order.

Unordered Map in STL :

DefinitionUnordered Map in stl is defined as the container which store the elements in key-value pair in non-sorted order.

Important Note

- The items are stored in mapped way in associative containers called maps.
- There are key values and mapped values for every element.
- The key values for any two mapped values cannot be the same.
- An associated container called unordered map holds elements created by fusing a mapped value with a key value.
- The element is uniquely identified by the key value, and the mapped value is the content related to the key.

Difference in Map and Unordered Map :

Map	Unordered Map
Elements are ordered in increasing order	Elements are not ordered in the sorted order.
Implementation is done by BST	Implementation is done by Hashtable.
It is slow	It is fast.
Used when ordering is required	Used when no ordering is required.

Example of Map in STL :

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

int main() {
    map< int,int> mpp;

    mpp.insert({1,10});
    mpp.insert({2,20});
    mpp.insert({4,40});
    mpp.insert({8,80});
    mpp.insert({6,60});
    mpp.insert({3,30});

    cout<<"The key and value in the map : "<< endl;
    for(auto it : mpp){
        cout<< it.first<<" "<< it.second<< endl;
    }
    cout<< endl;
    cout<<"The size of map :"<< mpp.size();
    return 0;
}
```

Example of Unordered Map in STL:

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

int main() {
    unordered_map< int,int> mpp;

    mpp.insert({1,10});
    mpp.insert({2,20});
    mpp.insert({4,40});
    mpp.insert({8,80});
    mpp.insert({6,60});
    mpp.insert({3,30});

    cout<<"The key and value in the map : "<< endl;
    for(auto it : mpp){
        cout<< it.first<<" "<< it.second<< endl;
    }
    cout<< endl;
    cout<<"The size of map :"<< mpp.size();
    return 0;
}
```

Round in STL C++

Library Function round() in C++

In C++ programming language the round() function is included in cmath header file.

The round() function in C++ can be used to round any floating-point number within the range of the corresponding floating-point type (double, float, or long double) to the nearest integer. The value is rounded to the nearest integer, with half-integers being rounded to the nearest even integer.
For example, round(1.5) will return 2, while round(2.5) will return 2.

Syntax

```
double round(double x);
float round(float x);
long double round(long double x);
```

Parameters

The round() function takes a single parameter x.

Parameter	Description
x	It is a floating-point number which is to be rounded off.

Return value

The round() function in C++ returns the nearest integer to the floating-point number passed as an parameter as a double, float, or long double depending on the type of the function being called.

mplementation of round() function in C++

Example 1:

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

int main() {

  double num = 3.14;
  cout << "round(" << num  << ") = "<< round(num) << endl;

  num = 3.7;
  cout << "round(" << num  << ") = "<< round(num) << endl;

  return 0;
}
```

Note: The round() function returns an integer, so if you pass it a value that is already an integer (e.g. round(3)), it will simply return the original value without any changes.

Multiset in STL C++

Multiset in STL is defined as the container used to store the elements just like set but they can store them in sorted order or duplicate element can also exist in it.

Working of Multiset :
In STL, Set-like containers called multisets exist. Multisets, as contrast to sets, may organise the storage of duplicate elements.Once they are added to the multiset, the items cannot be modified; they can only be added or removed.The header file for #include<set> contains a multiset. Iterators can be used to access the multiset's elements.

Functions in Multiset :

Function	Working
insert()	insert an element to the multiset.
clear()	remove all the element of the multiset.
size()	give the number of elements present in the multiset.
empty()	tells whether the set is empty or not.
count()	return the frequency of particular character in the multiset.

Example of Multiset in STL :

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

int main() {
   multiset< int> mset;
   mset.insert(1);
   mset.insert(2);
   mset.insert(10);
   mset.insert(15);
   mset.insert(1);
   mset.insert(4);
   cout<<"The elements in the multiset : ";
   for(auto it : mset){
       cout<< it<<" ";
   }
```

```
    return 0;
}
```

Heap in STL C++

Heap in STL is defined as the container used to store the elements in the order such that the maximum elements always remains on the top.

Working of Heap :
In STL, A heap is a type of data structure that makes access to the topmost (or lowest) member possible in O(1) time. The order of each additional component is dependent on how it is implemented, but it stays constant throughout.The header “algorithm” has a definition of this function.

Functions in Map :

Function	Working
make_heap()	convert container in to heap
front()	return the front element of the heap.
push_heap()	insert element into the heap.
sort_heap()	sort the elements of the heap.
pop_heap()	remove the element from the heap.

Example of Heap in STL :

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

int main() {

    vector vect = {2,3,1,30,20,50,5,10};
    // converting vect to heap
    make_heap(vect.begin(), vect.end());
    cout<<"The elements in the heap :";
    for(auto it : vect){
        cout<< it<<" ";
    }
    cout<< endl;
    cout<< "The maximum element in the heap : "<< vect.front();
    return 0;
}
```

Sort in STL C++

Sort in STL in C++

- In C++ programming language, the sort() function is included in algorithm header file.
- Sorting is defined as arranging the data in a particular order, which can be increasing, decreasing or in any specific way.

Syntax

```
sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

where **first** and **last** are the beginning and ending iterators, respectively, of the range to be sorted and **comp** must be a function that should take two arguments of the same type as the elements in the range being sorted, and it should return a bool indicating whether the first argument should be considered "less than" the second argument.

Parameters

The sort() function in C++ has the following parameters:

Parameter	Description
first	It is an iterator pointing to the first element in the range to be sorted.
last	It is an iterator pointing to the element one past the last element in the range to be sorted.
comp	It is an optional parameter that specifies a custom comparison function to be used to compare the elements in the range being sorted.

Note: The sort function doesn't produce a result. Just the items/elements from the first to the last iterations are updated. If you don't specify the comp parameter, the sort function will use the default comparison function, which sorts the elements in ascending order.

Sorting in Descending Order:

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

int main() {
    vector< int > arr;
    arr.push_back(45);
    arr.push_back(80);
    arr.push_back(35);
    arr.push_back(69);
    arr.push_back(3);

     cout<<"The element of array before sorting : ";
    for(int i=0;i< arr.size();i++){
         cout<< arr[i]<<" ";
    }
    cout<< endl;

    // sort function for sorting the elements
    sort(arr.begin(), arr.end(), greater());
    cout<<"The element of array after sorting : ";
    for(int i=0;i< arr.size();i++){
         cout<< arr[i]<<" ";
    }

    return 0;
}
```

Transform in C++ STL

Sort in STL in C++

- In C++ programming language, the transform() function is included in algorithm header file.
- Transform function in C++ is used to perform operations on all the elements which could be of any type.
- Suppose, we need to multiply every element of an array by a particular number, then we can make use of Transform function.

Syntax

For Unary operations :

```
transform(Iterator inputBegin, Iterator inputEnd,Iterator OutputBegin, unary_operation);
```

For Binary operations :

```
transform(Iterator inputBegin1, Iterator inputEnd1,Iterator inputBegin2,
            Iterator OutputBegin, binary_operation);
```

where inputBegin and inputEnd are the beginning and ending iterators, respectively, of the range to be transformed, OutputBegin specifies the location where the transformed values need to be stored and operation specifies what transformation operation is to be performed and then records the resulting value at the desired location provided by the user.

Parameters

The transform() function in C++ has the following parameters:

Parameter	Description
inputBegin	It is an iterator pointing to the first element in the range to be transformed.
inputEnd	It is an iterator pointing to the element one past the last element in the range to be transformed.
OutputBegin	It is an iterator pointing to the first index of the ouput where the transformed values will be stored.
operation	It specifies about what transformation is to be performed.

Note: The transform function in C++ using the above mentioned syntax can perform unary operations as well as binary operations depending on the user's requirement and provided suitable parameters.

Binary Search Functions in C++ STL

In C++ programming language, there are mainly 3 types of operations which are performed using binary search :

1. Finding an element
2. Lower bound
3. Upper bound

1. Finding an element :

While finding an element using binary search, the function will return true if the element is present.Otherwise, it will return false.

Example

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
        vector arr = { 10, 15, 20, 25, 30, 35 };
        if (binary_search(arr.begin(), arr.end(), 15))
                cout << "15 exists in vector";
        else
                cout << "15 does not exist";
        cout << endl;
        if (binary_search(arr.begin(), arr.end(), 23))
                cout << "23 exists in vector";
        else
                cout << "23 does not exist";

        cout << endl;
}
```

2. Lower Bound :

The lower bound function returns pointer to the position of num if the container contains only one occurrence of num. It returns a pointer to the first position of num if the container contains multiple occurrences of num. It returns pointer to the position of a number just higher than num, if the container does not contain an occurrence of num which is the position of the number when inserted in the already sorted array and sorted again. Subtracting the first position i.e vect.begin() from the pointer, returns the actual index.

Example

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
        vector arr1 = { 10, 15, 20, 25, 30, 35 };
        vector arr2 = { 10, 15, 20, 20, 25, 30, 35 };
        vector arr3 = { 10, 15, 25, 30, 35 };

        cout << "The position of 20 using lower_bound "
                        " (in single occurrence case) : ";
        cout << lower_bound(arr1.begin(), arr1.end(), 20)
                                - arr1.begin();
        cout << endl;

        cout << "The position of 20 using lower_bound "
                        "(in multiple occurrence case) : ";
        cout << lower_bound(arr2.begin(), arr2.end(), 20)
                                - arr2.begin();
        cout << endl;

        cout << "The position of 20 using lower_bound "
                        "(in no occurrence case) : ";
        cout << lower_bound(arr3.begin(), arr3.end(), 20)
                                - arr3.begin();
        cout << endl;
}
```

3. Upper Bound:

The upper bound function returns pointer to the position of next higher number than num if the container contains one occurrence of num. It returns pointer to the first position of the next higher number than the last occurrence of num if the container contains multiple occurrences of num. It returns pointer to position of next higher number than num if the container does not contain an occurrence of num. Subtracting the first position i.e vect.begin() from the pointer, returns the actual index.

Example

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
        vector arr1 = { 10, 15, 20, 25, 30, 35 };
        vector arr2 = { 10, 15, 20, 20, 25, 30, 35 };
        vector arr3 = { 10, 15, 25, 30, 35 };

        cout << "The position of 20 using upper_bound"
                        " (in single occurrence case) : ";
        cout << upper_bound(arr1.begin(), arr1.end(), 20)
                                - arr1.begin();
        cout << endl;

        cout << "The position of 20 using upper_bound "
                        "(in multiple occurrence case) : ";
        cout << upper_bound(arr2.begin(), arr2.end(), 20)
                                - arr2.begin();
        cout << endl;

        cout << "The position of 20 using upper_bound"
                        " (in no occurrence case) : ";
        cout << upper_bound(arr3.begin(), arr3.end(), 20)
                                - arr3.begin();

        cout << endl;
}
```

About bits/stdc++.h in C++

About bits/stdc++.h in C++

This header file is not part of the C++ standard library, but it is widely used in competitive programming because it saves the time and effort of including individual header files for each library that is needed.

Using bits/stdc++.h may make your code less portable because it includes a lot of implementation-specific header files. It is generally a good idea to include only the specific header files that you need in your code.

Syntax

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

Advantages of bits/stdc++.h

- It saves time and effort by including a lot of standard C++ libraries in a single file.
- It can be especially useful in competitive programming, where time is often a critical factor.
- It can reduce the number of #include statements in your code, which can make your code more concise and easier to read.

Disadvantages of bits/stdc++.h

- It includes a lot of header files that you may not actually need in your code. This can increase the compile time of your code and make your compiled binaries larger.
- It is not part of the C++ standard library, and it may not be available on all systems or compilers.
- It is generally a good idea to avoid using non-standard library extensions in your code to ensure maximum portability.

Implementation of bits/stdc++.h in C++

Example :

The following code shows the use of header file bits/stdc++.h :

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

int main() {

   // Use standard library functions and objects
   vector< int>v = {1, 2, 3, 4, 5};
   sort(v.begin(), v.end());

   for (int x : v) {
       cout << x << " ";
   }
```

```
        cout << endl;

    return 0;
}
```

Note: The header file bits/stdc++.h can be a useful shortcut in some situations, it is generally a good idea to include only the specific header files that you need in your code to ensure maximum portability and efficiency.

Unordered Set in STL C++

Unordered Set in STL is defined as a container that stores unique elements in no particular order, and allows for fast retrieval of the elements.

Syntax of Unordered Set :

```
unordered_set< data_type> unordered_set_name;
```

In STL, each value in the set serves as a key and the set doesn’t enable indexing, each element of the set is distinct, meaning that no duplicate values may be placed in it.
Unordered sets are implemented using a hash table, which means that the elements can be quickly inserted and retrieved, but the order in which the elements are inserted is not preserved.

Unordered sets are useful when you need to store a collection of unique elements and you don’t need to preserve the order of the elements. They offer fast insertion, removal, and search operations, and use less memory than sets because they do not store the elements in a specific order.

Functions in Unordered Set :

Function	Working
insert()	It inserts an element to the set.
clear()	It removes all the element of the set.
size()	It gives the number of elements present in the set.
empty()	It tells whether the set is empty or not.
max_size()	It returns the maximum number of elements set can hold.

Implementation of Unordered Set in STL C++

Example 1 :

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

```
int main() {
    unordered_set< int> ust;
    ust.insert(8);
    ust.insert(8);
    ust.insert(3);
    ust.insert(3);
    ust.insert(8);
    cout<<"The elements in the set : ";
    for(auto it : ust){
        cout<< it<<" ";
    }

    return 0;
}
```

Range-based for loop in C++

Range-based For loop

In C++ programming language, we will see the syntax of range based for loop and along with it example code for the same. Range based for loops are used to implement for loops within a specified range.

Syntax:

```
for( range_declaration : range_expression )
    loop_statement
```

Range Declaration :

a declaration of a named variable, whose type is the type of the element of the sequence represented by
range_expression

Range Expression :

any expression that represents a suitable sequence or a braced-init-list.

Loop statement :

any statement, typically a compound statement, which is the body of the loop.

Implementation of range based for loops in C++

Example:

The following code shows the use of range-based for loops.

```
#include<iostream>
```

```
#include<vector>
using namespace std;
int main()
{
		vector v = { 0, 1, 2, 3, 4, 5 };
		for (auto i : v)
			cout << i << ' ';

		cout << '\n';
		for (int n : { 0, 1, 2, 3, 4, 5 })
			cout << n << ' ';

		cout << '\n';
		int a[] = { 0, 1, 2, 3, 4, 5 };
		for (int n : a)
			cout << n << ' ';

		cout << '\n';
		for (int n : a)
			cout << "Prime" << ' ';

		cout << '\n';
		string str = "Prep";
		for (char c : str)
			cout << c << ' ';

		cout << '\n';
}
```

Math in STL

sinh() function in C++ STL

The sinh() function uses a mathematical formula to calculate the hyperbolic sine of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function sinh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function sinh() in C++

In C++ programming language, the sinh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of sinh function

```
sinh(data_type x)
```

Parameters of sinh function

The sinh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	Description
Hyperbolic angle	This value of the parameter can be of double / float or long double type.

Return value of sinh function

The sinh function returns the hyperbolic sine value of the argument passed.

Parameter	Return Value
Hyperbolic sine of the argument passed	It returns the value in double / float or long double type.

Note: If the return value of the sinh function is too large to be displayed as output, then the function returns 'inf' as output.

Implementation of STL Function sinh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    double x = 4.1;
    double result = sinh(x);
    cout << "sinh(4.1) = " << result << endl;
    double xDegrees = 90;
    x = xDegrees * 3.14159 / 180;
    result = sinh(x);
    cout << "sinh(90 degrees) = " << result << endl;
    return 0;
}
```

cosh() function in C++ STL

The cosh() function uses a mathematical formula to calculate the hyperbolic cosine of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function cosh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function cosh() in C++

In C++ programming language, the cosh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of cosh function

```
cosh(data_type x)
```

Parameters of cosh function

The cosh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	Description
Hyperbolic angle	This value of the parameter can be of double / float or long double type.

Return value of cosh function

The cosh function returns the hyperbolic cosine value of the argument passed.

Parameter	Return Value
Hyperbolic cosine of the argument passed	It returns the value in double / float or long double type.

Note: If the return value of the cosh function is too large to be displayed as output, then the function returns 'inf' as output.

Implementation of STL Function cosh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    double x = 4.1, result;
    result = cosh(x);
    cout << "cosh(4.1) = " << result << endl;
    double xDegrees = 90;
    x = xDegrees * 3.14159 / 180;
    result = cosh(x);
    cout << "cosh(90 degrees) = " << result << endl;
    return 0;
}
```

tanh() function in C++ STL

The tanh() function uses a mathematical formula to calculate the hyperbolic tangent of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function tanh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function tanh() in C++

In C++ programming language, the tanh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of tanh function

```
tanh(data_type x)
```

Parameters of tanh function

The tanh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	Description
Hyperbolic angle	This value of the parameter can be of double / float or long double type.

Return value of tanh function

The tanh function returns the hyperbolic tangent value of the argument passed.

Parameter	Return Value
Hyperbolic tangent of the argument passed	It returns the value in double / float or long double type.

Implementation of STL Function tanh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
   double x = 4.1, result;
   result = tanh(x);
   cout << "tanh(4.1) = " << result << endl;
   double xDegrees = 90;
   x = xDegrees * 3.14159 / 180;
   result = tanh(x);
   cout << "tanh(90 degrees) = " << result << endl;
   return 0;
}
```

Note: The function returns no matching function for call to error when a string or character is passed as an argument.

asinh() function in C++ STL

The asinh() function uses a mathematical formula to calculate the inverse hyperbolic sine of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function asinh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function asinh() in C++

In C++ programming language, the asinh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of asinh function

```
asinh(data_type x)
```

Parameters of asinh function

The asinh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	Description
Hyperbolic angle	This value of the parameter can be of double / float /int or long double type.

Return value of asinh function

The asinh function returns the inverse hyperbolic sine value of the argument passed.

Parameter	Return Value
Inverse hyperbolic sine of the argument passed	It returns the value in double / float / int or long double type.

Implementation of STL Function asinh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
   int x = -50.0;
   double result = asinh(x);
   cout << "asinh(-50.0) = " << result << " radians"<< endl;
   cout << "asinh(-50.0) = " << result * 180 / 3.141592<< " degrees" << endl;
```

```
    return 0;
}
```

Note: The function returns no matching function for call to error when a string or character is passed as an argument.

acosh() function in C++ STL

The acosh() function uses a mathematical formula to calculate the inverse hyperbolic cosine of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function acosh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function acosh() in C++

In C++ programming language, the acosh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of acosh function

```
acosh(data_type x)
```

Parameters of acosh function

The acosh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	Description
Hyperbolic angle	This value of the parameter can be of double / float /int or long double type.

Return value of acosh function

The acosh function returns the inverse hyperbolic cosine value of the argument passed.

Parameter	Return Value
Inverse hyperbolic cosine of the argument passed	It returns the value in double / float / int or long double type

Implementation of STL Function acosh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    double x = 50.0;
    double result = acosh(x);
    cout << "acosh(50.0) = " << result << " radians"<< endl;
    cout << "acosh(50.0) = " << result * 180 / 3.141592<< " degrees" << endl;
    return 0;
}
```

Note: The function returns no matching function for call to error when a string or character is passed as an argument.

Note: The function returns NaN (not a number) if a negative value is passed as an argument.

atanh() function in C++ STL

The atanh() function uses a mathematical formula to calculate the inverse hyperbolic tangent of a number. This formula involves the use of exponential functions and involves the ratio of the sides of a right triangle in a hyperbolic coordinate system.In this section, we will discuss about library function atanh() in STL which is used in C++.The Standard Template Library(STL) is a set of complete data structures and functions which can be used in C++.

Library Function atanh() in C++

In C++ programming language, the atanh function is included in standard template library.

The range of input argument which is passed to floor function is not limited here as it can be any value such as float or long double also.

Declaration of atanh function

```
atanh(data_type x)
```

Parameters of atanh function

The atanh function accepts a single input argument which is the hyperbolic angle and can be a double/float or long double value.

Parameter	**Description**
Hyperbolic angle	This value of the parameter can be of double / float /int or long double type.

Return value of atanh function

The atanh function returns the inverse hyperbolic tangent value of the argument passed.

Parameter	**Return Value**
-1 < x < 1	It returns a finite value.
x > 1 or x < -1	It returns NaN (Not a number).

Parameter	Return Value
x = -1	It returns (-inf).
x = 1	It returns (+inf).

Implementation of STL Function atanh() in C++

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int x = 0;
    double result = atanh(x);
    cout << "atanh(0) = " << result << " radians\n";
    cout << "atanh(0) = " << result * 180 / 3.141592 << " degrees\n";

    x = -1;
    result = atanh(x);
    cout << "\natanh(-1) = " << result << " radians\n";
    cout << "atanh(-1) = " << result * 180 / 3.141592 << " degrees\n";

    x = 1;
    result = atanh(x);
    cout << "\natanh(1) = " << result << " radians\n";
    cout << "atanh(1) = " << result * 180 / 3.141592 << " degrees\n";

    x = -2;
    result = atanh(x);
    cout << "\natanh(-2) = " << result << " radians\n";
    cout << "atanh(-2) = " << result * 180 / 3.141592 << " degrees\n";

    return 0;
}
```

Exception Handling

Stack Unwinding in C++

About Stack Unwinding in C++

Stack unwinding in C++ refers to the process of unwinding the call stack, which is the memory structure that stores the return addresses of function calls.

This process occurs when an exception is thrown and propagates up the call stack, looking for a catch block that can handle the exception. As the stack unwinds, the destructors of local objects are called, and any memory allocated on the stack is freed. This process ensures that resources are properly released and memory is not leaked in the event of an exception.

Note: If there are more functions in the call stack, then the stack unwinding process goes up the call stack to find the catch block.

Example of Stack Unwinding in C++

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

class MyException:public exception
{
public:
  MyException():exception() {}
  const char *what() const throw()
  {
    return "My Exception";
  }
};

class MyResource
{
public:
  MyResource()
  {
    cout << "MyResource acquired" << endl;
  }
   ~MyResource()
  {
    cout << "MyResource released" << endl;
  }
};

void function1()
{
  MyResource res;
  throw MyException();
}

void function2()
{
  function1();
}

void function3()
```

```
{
  function2();
}

int main()
{
  try
  {
    function3();
  } catch (const MyException& e)
  {
    cout << "Caught exception: " << e.what () << endl;
  }
  return 0;
}
```

This program defines a custom exception class MyException and a class MyResource that has a constructor that prints “MyResource acquired” and a destructor that prints “MyResource released” when it’s called.

In this example:

- The function3() calls function2() and function2() calls function1() and in function1() the class object is created and it throws a custom exception of type MyException.
- The exception propagates up the call stack, looking for a catch block that can handle it. In this case, the catch block is located in the main function.
- As the stack unwinds, the destructors of any local objects will be called in the reverse order of their construction.

As you can see in the output, first the resource is acquired, then the exception is thrown and goes up the call stack, and finally caught in the catch block, the destructor of the resource is called. This process of going up the call stack to find the catch block and calling the destructor of the objects is what is known as stack unwinding.

Catching Base and Derived Classes as Exceptions in C++

Exception is defined as an unwanted error during compilation that a program throw. Exception handling in C++ is a mechanism for dealing with errors or unexpected conditions that may occur during the execution of a program. It allows you to separate the error-handling code from the normal code and handle errors in a structured and organized way.

Catching Base and Derived Classes as Exceptions

The basic structure of exception handling in C++ involves using the “try” block to enclose the code that may throw an exception. The “catch” block is used to handle the exception that was thrown. The catch block is associated with the try block by following the try block immediately.

Syntax

```
try {
   // code that may throw an exception
} catch (exceptionType e) {
   // code to handle the exception
}
```

When an exception is thrown inside the try block, the program will immediately exit the current function and search for the nearest enclosing try block. If a matching catch block is found, the program will transfer control to that catch block and execute the code inside it. If no matching catch block is found, the program will terminate.

Here, we will examine the concept of exception handling and the ways to catch base and derived classes in C++.

- If the base and derived classes that we have defined are caught as exceptions,then the catch block for the derived class will be displayed before the base class in the output terminal.
- The above statement is correct, which can be verified by reversing the order of the base and derived classes, and in that case the catch block for the derived class will not be reached.

Note: It is also possible to use multiple catch blocks to handle different types of exceptions. The catch block that appears first in the sequence will be executed first.

Example 1:

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

class Base {};
class Derived : public Base {};

int main()
{
  try
  {
    throw Derived();
  }
  catch (Base& b)
  {
    cout << "Caught base exception" << endl;
  }
  catch (Derived& d)
  {
    cout << "Caught derived exception" << endl;
  }

  return 0;
}
```

Thus it is proved that if catch block of the base class is written before the catch block for the derived class then the derived class catch block will never be reached.

Note: The exception handling can have a performance impact on your program, as it requires additional memory and processing time. Therefore, it's important to use exception handling judiciously and only for exceptional conditions that cannot be handled by normal program flow.

Catch Block and Type Conversion in C++

Catch Block

The catch block specifies a type of exception that it can handle, and the code inside the catch block is executed when an exception of that type is thrown.

Syntax:

```
try {
    // code that may throw an exception
} catch (exceptionType e) {
    // code to handle the exception
}
```

Here, **"exceptionType"** is the type of exception that the catch block can handle, and **"e"** is a variable that holds the exception object. The exception object can be used to access information about the exception, such as the error message or stack trace.

You can also use catch block without any exception type which will catch any exception thrown in try block.

```
try {
    // code that may throw an exception
} catch (...) {
    // code to handle the exception
}
```

Note: You can also have multiple catch blocks to handle different types of exceptions. The order of catch block matters, the catch block for the most specific exception type should be written first and the catch block for the least specific exception type should be written last.

Type conversion in Exceptional handling

Type conversion in exceptional handling in C++ refers to the process of converting an exception object from one type to another. In C++, when an exception is thrown, the catch block that will handle the exception is determined by matching the type of the exception object to the type specified in the catch block. If the exception object is of a different type than the catch block, then the catch block will not be executed.

To handle this scenario, C++ allows for type conversion of exception objects so that a catch block can handle an exception object of a different type than what is specified in the catch block. This is done by using a conversion constructor, which is a constructor that takes a single argument of a different type and converts it to the type of the class.

Example 1:

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

int main()
{
    try
    {
        throw 'p';
    }
    catch(int p)
    {
        cout << "Integer value is caught :" << p;
    }
    catch(...)
    {
        cout << "Default catch block";
    }
}
```

The above program throws an exception of character type, ‘p’ and there is a catch block to catch an exception of int type. One could assume that the catch block for int should match with the ASCII value of ‘p’, however, catch blocks do not perform such type of conversion.

Note: When a derived type object is thrown and there is a catch block to catch a base type, then the derived object is automatically converted to the base type.

Exception Handling and Object Destruction in C++

Exception Handling in C++ :

DefinitionException handling is performed in C++ using try, catch and throw. These help in making sure that the whole program runs completely, even if some minute runtime errors that may occur due to logical, boundary and other problem in the program.

Object Destruction in C++:

DefinitionA destructor is a member function of a class with the same name as the class but is prefixed by the symbol ~(tilde). It is also invoked automatically after the code's scope has expired. Object destruction is the process of destroying or crushing the existing object memory.

Example 1 :

```
#include<iostream>
using namespace std;
class check {
public:
		check()
		{
				cout << "Constructing an object of class check "<< endl;
		}

		~check()
		{
				cout << "Destructing the object of class check "<< endl;
		}
};
int main()
{
		try {
				check t1;
				throw 5;
		}
		catch (int i) {
				cout << "Caught " << i << endl;
		}
}
```

In the above program, when an exception is thrown, destructors of the objects (whose scope ends with the try block) are automatically called before the catch block gets executed.

Library in C++

sqrt(), sqrtl() and sqrtf() in C++

C++ Library Function cmath sqrt()

In C++, the sqrt() function is included in cmath header file.

The sqrt() function is used to compute the square root of a number. Square root of a number is a value, which on multiplication by itself, gives the original number.

Syntax

```
double sqrt(double x);
```

sqrt() function return a double data type as result or as a square root of a number.,

Example:

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

int main() {

  double x = 9.0;
  double y = sqrt(x);

  cout << "The square root of " << x << " is " << y << endl;

  return 0;
}
```

C++ Library Function cmath sqrtl()

In C++, the sqrtl() function is included in cmath header file.

The sqrtl() function is used to compute the square root of a long double value. This function has output with more precision.

Syntax

```
long double sqrtl(long double x);
```

sqrtl() function returns a long double data type as result or as a square root of a number.

Example:

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

int main() {

  long double x = 1345.0;
  long double y = sqrtl(x);

  cout << "The square root of " << x << " is " << y << endl;

  return 0;
}
```

C++ Library Function cmath sqrtf()

In C++, the sqrtf() function is included in cmath header file.

The sqrtf() function is used to compute the square root of a float value.

Syntax

```
float sqrtf(float x);
```

sqrtf() function returns a float data type as result or as a square root of a number.

Example:

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

int main() {

  float x = 15.67;
  float y = sqrtf(x);

  cout << "The square root of " << x << " is " << y << endl;

  return 0;
}
```

Library Function rename() in C++

Library Function rename() in C++

In C++ programming language, the **rename()** function is included in cstdio header file.

The **rename()** function is used to change the name by passing as an argument of a file from old name to new name without changing the file's content. If new name is the same as the name of an existing file in the same folder, then the function may override the existing file, depending on the system and library implementation.

The **rename()** function can also be used to move a file to a different directory. Instead of copying the file to the new location, the original file is moved to the new directory location.

Declaration

```
int rename(const char* oldname, const char* newname);
```

Parameters

This function takes two input parameters.

Parameter	Description
oldname	A string containing the name of the file you want to rename.
newname	A string containing the new name for the file.

Return value

The return type of rename() function is an integer.

- This function returns zero on success i.e. if the file is renamed successfully.
- This function returns a non-zero value on error.

Implementation of Library Function rename() in C++

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

int main()
{
		const char* oldname = "old_file.txt";
	const char* newname = "new_file.txt";

		/*		Deletes the file if exists */
		if (rename(oldname, newname) != 0)
				perror("Error renaming file");
		else
				cout << "File renamed successfully";

		return 0;
}
```

Note: You can use the rename function to change the name of any file on your file system, as long as you have the necessary permissions to do so. It is important to note that the rename function will not work if the new name is already in use by another file. In this case, the rename function will return a non-zero value and the file will not be renamed.

Library Function strstr() in C++

Library Function strstr() in C++

In C++ programming language, the strstr function is included in cstring header file.

This function takes two arguments: str1 and str2. It searches for the first occurrence of a str2 within str1 and stops on ‘\0′(null terminating character).

Declaration

```
char *strstr(const char *str1, const char *str2);
```

Parameters

The strstr function in C++ takes two parameters:

Parameter	Description
str1	This is the string in which to search for the substring. It is a pointer to a null-terminated string.
str2	This is the string to search for within str1. It is also a pointer to a null-terminated string.

Return value

- If str2 is found within str1, a pointer to the first occurrence of str2 within str1 is returned.
- If str2 is not found within str1, a null pointer is returned.
- If an empty string is found in str2 then str1 is returned.

Implementation of Library Function strstr() :

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
   char str1[] = "PrepInsta Prime";
   char str2[] = "Prime";

   char *p = strstr(str1, str2);

   if (p)
   cout << " Found " << str2 << " in " << str1 << " at index " << p-str1;
   else
   cout << str2 << " Did not find " << str1 << " in ";

   return 0;
}
```

Tuples in C++

Properties of Tuples:

Some of the key properties of tuples in C++ include:

1. **Fixed size:** Tuples have a fixed number of elements, which are specified when the tuple is created. This means that the size of a tuple cannot be changed after it is created.
2. **Homogeneous elements:** The elements in a tuple can be of different types, but they must all be of the same type.
3. **Immutable elements:** The elements of a tuple are immutable, which means that they cannot be changed after the tuple is created.
4. **Efficient memory usage:** Tuples are designed to be efficient in terms of memory usage, since they store their elements in a contiguous block of memory.
5. **Type-safe:** Tuples are type-safe, which means that the compiler checks the types of the elements when the tuple is created. This helps to prevent type errors and ensures that the elements are used correctly.
6. **Easy to use:** Tuples are easy to use, as they provide a simple and convenient way to store and manipulate multiple values.

Declaration

```
tuple<int, float, string> myTuple;
```

Initialization

There are two ways to initialize tuples in C++.

1. The simplest way is to specify the values of the elements within parentheses, as shown below:

```
tuple<int, float, string> myTuple(1, 3.14, "hello");
```

2. We can also use the make_tuple function to create a tuple, as shown below:

```
tuple<int, float, string> myTuple
myTuple = make tuple(1, 3.14, "hello");
```

Functions used in Tuples:

- **get():** This function is used to access the elements of a tuple and for modifying them. It takes the tuple and an index as arguments, and returns the element at the specified position within the tuple.
- **tie():** This function is used to unpack the elements of a tuple into separate variables.
- **swap():** This function is used to swap the elements of the two different tuples.

- **tuple_size:** This function is used to determine the number of elements in a tuple.
- **tuple_cat:** This function is used to concatenate two or more tuples.