C++ STL

10 November 2024 16:01

Introduction to C++ STL
The C++ Standard Template Library (STL) is a comprehensive collection of template classes and functions that provide solutions to common programming tasks. It is designed to offer high-performance, reusable components that can be easily integrated into C++ programs. By utilizing the STL, developers can focus on the logic of their applications without needing to implement data structures and algorithms from scratch.

The STL is a cornerstone of modern C++ programming, enabling efficient code development and reducing the potential for errors. It includes powerful features for handling data, performing operations, and managing resources, making it an essential tool for any C++ developer.

Example of STL Usage

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

int main() {
   vector numbers = {1, 2, 3, 4, 5};
   sort(numbers.begin(), numbers.end());

   for (int num : numbers) {
      cout << num << " ";
   }
   return 0;
}
```

In the above example, the **vector** container stores a list of integers, and the **sort** algorithm is used to arrange them in ascending order. The STL simplifies the process by providing these ready-made components.

Key Components of STL

Containers

Containers in STL are used to store collections of objects. They provide various data structures like vector, list, set, and map to efficiently manage and access data.

Algorithms

Algorithms are a collection of functions that perform tasks such as searching, sorting, and manipulating data stored in containers. They are designed to work with a wide variety of data types.

Functions

Functions in STL include a range of utility functions that perform specific operations on data, such as for_each and transform, making it easier to process collections of data.

Iterators

Iterators are objects that point to elements within a container and provide a way to traverse through the container. They are essential for accessing and modifying elements in STL containers.

STL - Pair

The pair in C++ STL is a utility that allows the storage of two heterogeneous objects as a single unit. This is particularly useful when there is a need to associate two values together, such as when storing key-value pairs. The pair can store values of any data type, and the values can be accessed directly using first and second members.

Example Code

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

int main() {
    pair student;
    student.first = 1;
    student.second = "Alice";

    cout << "Roll No: " << student.first << ", Name: " << student.second << endl;
    return 0;
}
```

Expected Output

```
Roll No: 1, Name: Alice
```

Example Code

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

int main() {
    pair student;
    student.first = 1;
    student.second = "Alice";

    cout << "Roll No: " << student.first << ", Name: " << student.second << endl;

    pair, int> complexPair = {{2, 'B'}, 20};
    cout << "Roll No: " << complexPair.first.first << ", Section: " << complexPair.first.second;
    cout << ", Marks: " << complexPair.second << endl;

    return 0;
}
```

Expected Output

```
Roll No: 1, Name: Alice
```

Roll No: 2, Section: B, Marks: 20

C++ STL Vector Operations

1. Initialisation

vector is a dynamic array that can automatically resize itself when elements are added or removed. It provides the same functionalities as an array but with additional features like dynamic resizing.

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

    int main() {
      vector<int> v = {1, 2, 3, 4, 5};
      for (int i : v) {
        cout << i << " ";
      }
      return 0;
    }
```

Expected Output: 1 2 3 4 5

2. push_back

The push_back function adds a new element at the end of the vector. This is often used to append elements in a loop or based on certain conditions.

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

    int main() {
      vector<int> v = {1, 2, 3};
      v.push_back(4);
      v.push_back(5);
      for (int i : v) {
        cout << i << " ";
      }
      return 0;
    }
```

Expected Output: 1 2 3 4 5

3. front

The front function returns a reference to the first element in the vector. This is useful when you need to access the starting element without modifying the vector.

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

    int main() {
      vector<int> v = {10, 20, 30};
      cout << "Front element: " << v.front();
      return 0;
    }
```

Expected Output: Front element: 10

4. back

The back function returns a reference to the last element in the vector. It's typically used when the last inserted element needs to be accessed or modified.

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

    int main() {
      vector<int> v = {10, 20, 30};
      cout << "Back element: " << v.back();
      return 0;
    }
```

Expected Output: Back element: 30

5. iterator

Vectors support iterators, which allow traversal through the elements in the vector. Iterators behave like pointers and can be used for accessing elements or performing operations on the vector.

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

    int main() {
      vector<int> v = {10, 20, 30};
      for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";
      }
      return 0;
    }
```

Expected Output: 10 20 30

6. for each

The for_each loop in C++11 allows for a simpler way to iterate over vectors. It simplifies the syntax and enhances code readability.

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

    int main() {
```

```
    vector<int> v = {10, 20, 30};
    for (int i : v) {
        cout << i << " ";
    }
    return 0;
}
```

Expected Output: 10 20 30

7. erase

The erase function removes an element from the vector by a position causing all subsequent elements to be shifted one position to the left.This helps reduces the size of the vector by one.

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

int main() {
    vector<int> v = {10, 20, 30};
    v.erase(v.begin() + 1);
    for (int i : v) {
        cout << i << " ";
    }
    return 0;
}
```

Expected Output: 10 30

8. insert

The insert function inserts elements at a specified position in the vector. It shifts all elements at and after the insertion point to the right, increasing the vector's size.

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

int main() {
    vector<int> v = {10, 30};
    v.insert(v.begin() + 1, 20);
    for (int i : v) {
        cout << i << " ";
    }
    return 0;
}
```

Expected Output: 10 20 30

9. size

The size function returns the number of elements in the vector. This is helpful in loops or conditions where the vector’s length is a determining factor.

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

int main() {
    vector<int> v = {10, 20, 30};
    cout << "Size of vector: " << v.size();
    return 0;
}
```

Expected Output: Size of vector: 3

10. Access Element

Elements in a vector can be accessed using the [] operator or the at() method.

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

    int main() {
        vector<int> v = {10, 20, 30};
        cout << "Element at index 1: " << v[1];
        return 0;
    }
```

Expected Output: Element at index 1: 20

11. swap

The swap function swaps the contents of two vectors. This is an efficient way to swap large vectors since it only swaps the internal pointers rather than copying elements.

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

    int main() {
        vector<int> v1 = {1, 2, 3};
        vector<int> v2 = {4, 5, 6};
        swap(v1, v2);
        for (int i : v1) {
            cout << i << " ";
        }
        return 0;
    }
```

Expected Output: 4 5 6

C++ STL List and Deque Operations

What is a List?

Lists are sequence containers that support non-contiguous memory allocation. Unlike vectors, which offer faster traversal, lists have slower traversal times. However, they excel in insertion and deletion operations, which are performed in constant time once the desired position is identified. A list in C++ STL is a doubly-linked list, which allows elements to be efficiently inserted or deleted from both ends as well as from the middle. Unlike arrays or vectors, lists do not provide fast random access but are highly efficient in insertion and deletion operations.

1. push_front / emplace_front (List)

The push_front function inserts a new element at the beginning of the list, while **emplace_front** constructs an element in place at the beginning. These operations are efficient since they only involve adjusting a few pointers.

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

    int main() {
        list<int> lst;
        lst.push_front(20);
        lst.emplace_front(10);  // Constructs 10 at the beginning
        for (int i : lst) {
            cout << i << " ";
        }
        return 0;
    }
```

Expected Output: 10 20

2. front (List)

The front function returns a reference to the first element in the list. This is useful for accessing or modifying the element at the front without affecting the rest of the list.

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

    int main() {
        list<int> lst = {10, 20, 30};
        cout << "Front element: " << lst.front();
        return 0;
    }
```

3. Deque

A deque is a sequence container with the ability to expand and contract on both ends efficiently. It supports fast insertions and deletions at both the beginning and the end, making it more flexible than a vector or list in scenarios where elements are frequently added or removed from both ends.

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

    int main() {
        deque<int> dq = {10, 20, 30};
        dq.push_front(0);  // Insert at the beginning
        dq.push_back(40);  // Insert at the end
        for (int i : dq) {
            cout << i << " ";
        }
        return 0;
    }
```

Expected Output: 0 10 20 30 40

Stack (LIFO)

In C++ Standard Template Library (STL), the stack is a powerful container adapter that follows the Last In, First Out (LIFO) principle. It provides a streamlined interface for adding and removing elements from one end, making it ideal for scenarios requiring dynamic data management and nested operations.

1. Push / Emplace

The push operation adds an element to the top of the stack. The emplace operation is similar, but it constructs the element in place, which can be more efficient.

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

   int main() {
      stack<int> s;
      s.push(10);
      s.emplace(20); // Emplace is similar to push

      cout << "Top element: " << s.top() << endl;
      return 0;
   }
```

Expected Output: Top element: 20

2. Top

The top operation returns the element at the top of the stack without removing it.

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

int main() {
   stack<int> s;
   s.push(10);
   s.push(20);

   cout << "Top element: " << s.top() << endl;
   return 0;
}
```

Expected Output:

```
Top element: 20
```

3. Pop

The pop operation removes the top element from the stack.

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

int main() {
   stack<int> s;
   s.push(10);
   s.push(20);
   s.pop();

   cout << "Top element after pop: " << s.top() << endl;
```

```
    return 0;
}
```

Expected Output: Top element after pop: 10

4. Size

The size operation returns the number of elements in the stack.

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

int main() {
    stack<int> s;
    s.push(10);
    s.push(20);

    cout << "Size of stack: " << s.size() << endl;
    return 0;
}
```

Expected Output: Size of stack: 2

5. Swap

The swap operation swaps the contents of two stacks.

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

int main() {
    stack<int> s1, s2;
    s1.push(10);
    s2.push(20);
    s1.swap(s2);

    cout << "Top element of s1 after swap: " << s1.top() << endl;
    cout << "Top element of s2 after swap: " << s2.top() << endl;
    return 0;
}
```

Expected Output: Top element of s1 after swap: 20 Top element of s2 after swap: 10

6. Empty

The empty operation checks if the stack is empty. It returns true if the stack is empty, otherwise false.

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

int main() {
    stack<int> s;

    cout << "Is stack empty? " << (s.empty() ? "Yes" : "No") << endl;
    s.push(10);

    cout << "Is stack empty after push? " << (s.empty() ? "Yes" : "No") << endl;
    return 0;
}
```

Expected Output: Is stack empty? Yes Is stack empty after push? No

Queue (FIFO)

In C++ Standard Template Library (STL), the queue is a versatile container adapter that operates on a First In, First Out (FIFO) basis. It facilitates efficient data handling by allowing elements to be inserted at one end and removed from the other making it particularly useful for managing tasks and scheduling operations in a sequential manner.

1. Push / Emplace

The push operation adds an element to the end of the queue. The emplace operation constructs the element in place, similar to push, but more efficient.

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

int main() {
    queue<int> q;
    q.push(10);
    q.emplace(20); // Emplace is similar to push

    cout << "Front element: " << q.front() << endl;
    return 0;
}
```

Expected Output: Front element: 10

2. Front

The front operation returns the element at the front of the queue without removing it.

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

int main() {
    queue<int> q;
    q.push(10);
    q.push(20);

    cout << "Front element: " << q.front() << endl;
    return 0;
}
```

Expected Output: Front element: 10

3. Pop

The pop operation removes the front element from the queue.

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

int main() {
   queue<int> q;
   q.push(10);
   q.push(20);
   q.pop();

   cout << "Front element after pop: " << q.front() << endl;
   return 0;
}
```

Expected Output: Front element after pop: 20

4. Size

The size operation returns the number of elements in the queue.

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

int main() {
   queue<int> q;
   q.push(10);
   q.push(20);

   cout << "Size of queue: " << q.size() << endl;
   return 0;
}
```

Expected Output::Size of queue: 2

5. Swap

The swap operation swaps the contents of two queues.

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

int main() {
   queue<int> q1, q2;
   q1.push(10);
   q2.push(20);
   q1.swap(q2);

   cout << "Front element of q1 after swap: " << q1.front() << endl;
   cout << "Front element of q2 after swap: " << q2.front() << endl;
   return 0;
}
```

Expected Output: Front element of q1 after swap: 20 Front element of q2 after swap: 10

6. Empty

The empty operation checks if the queue is empty. It returns true if the queue is empty, otherwise false.

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

int main() {
    queue<int> q;

    cout << "Is queue empty? " << (q.empty() ? "Yes" : "No") << endl;

    q.push(10);

    cout << "Is queue empty after push? " << (q.empty() ? "Yes" : "No") << endl;
    return 0;
}
```

Expected Output: Is queue empty? Yes Is queue empty after push? No

Set Container

The `set` container in C++ stores unique elements in a sorted manner. It automatically removes duplicate entries and provides efficient search operations.

Insert / Emplace

Elements can be added to a set using two methods: insert and emplace. The insert function adds a new element if it's not already present, returning a pair indicating success and an iterator. The emplace function constructs the element in place, potentially improving efficiency by avoiding unnecessary copies or moves.

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

int main() {
    set<int> s;
    s.insert(10);
    s.insert(20);
    s.emplace(15); // Emplaces an element
    for(int x : s) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: 10 15 20

Find

The find function searches for a specified element within the set. It returns an iterator pointing to the element if found, or to the end() of the set if the element is not present. This allows for efficient lookups and can be used in combination with other operations to check for the existence of an element or to perform actions on it.

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

   int main() {
      set<int> s = {10, 20, 30};
      auto it = s.find(20);
      if(it != s.end()) cout << "Found: " << *it;
      else cout << "Not Found";
      return 0;
   }
```

Expected Output: Found: 20

Erase

The erase function removes elements from the set by specifying either an iterator pointing to the element or the element’s value directly. If an iterator is provided, it deletes the element at that position, while specifying a value will remove all occurrences of that value

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

   int main() {
      set<int> s = {10, 20, 30};
      s.erase(20);
      for(int x : s) {
         cout << x << " ";
      }
      return 0;
   }
```

Expected Output: 10 30

Lower Bound

Get the iterator to the first element that is not less than the given value.

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

int main() {
   set<int> s = {10, 20, 30};
   auto it = s.lower_bound(20);
   cout << "Lower Bound: " << *it;
   return 0;
}
```

Expected Output: Lower Bound: 20

Upper Bound

Get the iterator to the first element that is greater than the given value.

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

int main() {
    set<int> s = {10, 20, 30};
    auto it = s.upper_bound(20);
    cout << "Upper Bound: " << *it;
    return 0;
}
```

Expected Output: Upper Bound: 30

Multiset

The `multiset` container allows storing multiple occurrences of the same element. It is useful when you need to keep duplicates and still maintain sorted order.

Insert

Inserting elements into a multiset is similar to a set, but it allows multiple occurrences of the same element. The insert function can be used to add elements, and unlike a set, it does not enforce uniqueness. This makes it ideal for scenarios where duplicate values are needed while still maintaining the sorted order of elements.

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

int main() {
    multiset<int> ms;
    ms.insert(10);
    ms.insert(20);
    ms.insert(20);  // Duplicate element
    for(int x : ms) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: 10 20 20

Erase

Remove elements from the multiset. All occurrences of the element will be removed.

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

int main() {
   multiset<int> ms = {10, 20, 20};
   ms.erase(20);  // Erase all occurrences of 20
   for(int x : ms) {
      cout << x << " ";
   }
   return 0;
}
```

Expected Output: 10

Unordered Set

The `unordered_set` container stores unique elements in an unsorted manner. It provides faster search operations compared to a set due to hashing.

Insert / Emplace

Inserting elements into an unordered_set can be done using the **insert** or **emplace** methods. The **insert** function adds elements to the container, ensuring that each element is unique, and it relies on hashing for efficient insertion. The **emplace** method constructs elements in place, which can be more efficient by avoiding unnecessary copies or moves.

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

   int main() {
      unordered_set<int> us;
      us.insert(10);
      us.insert(20);
      us.emplace(30); // Emplaces an element
      for(int x : us) {
         cout << x << " ";
      }
      return 0;
   }
```

Expected Output: Elements printed in any order, e.g., 10 20 30

Find

Search for an element in the unordered_set.

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

int main() {
   unordered_set<int> us = {10, 20, 30};
   auto it = us.find(20);
   if(it != us.end()) cout << "Found: " << *it;
   else cout << "Not Found";
   return 0;
}
```

Expected Output: Found: 20

Erase

The **erase** function removes elements from an unordered_set by specifying either an iterator pointing to the element or the element's value directly.

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

int main() {
    unordered_set<int> us = {10, 20, 30};
    us.erase(20);
    for(int x : us) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: Elements printed in any order, e.g., 10 30

Map

The `map` container stores key-value pairs in a sorted order based on the keys. It provides efficient lookup and insertion operations.

Insert / Emplace / Assignment

Key-value pairs can be added to a **map** using insert, **emplace**, or assignment. The **insert** function adds a new pair without modifying existing entries if the key is present. The **emplace** method constructs the pair in place avoiding unnecessary copies or moves. Assignment updates the value for an existing key or adds a new key-value pair if the key is absent. All methods maintain the map's sorted order and ensure efficient operations.

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

    int main() {
        map<int, string> m;
        m.insert({1, "one"});
        m.emplace(2, "two");  // Emplaces a key-value pair
        m[3] = "three";       // Assignment
        for(auto &p : m) {
            cout << p.first << ": " << p.second << "\n";
        }
        return 0;
    }
```

Expected Output:

```
1: one
2: two
3: three
```

Find

The find function searches for a key in the **map** and returns an iterator to the key-value pair if the key is found. If the key is not present, it returns an iterator to the **end()** of the map, allowing efficient lookup with average logarithmic time complexity.

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

```
   using namespace std;

   int main() {
      map<int, string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
      auto it = m.find(2);
      if(it != m.end()) cout << "Found: " << it->second;
      else cout << "Not Found";
      return 0;
   }
```

Expected Output: Found: two

Empty

The empty function checks whether the map is empty by returning a boolean value. It returns true if the map contains no elements and false otherwise, providing a simple way to determine if any elements are present.

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

   int main() {
      map<int, string> m;
      cout << (m.empty() ? "Empty" : "Not Empty");
      return 0;
   }
```

Expected Output: Empty

Size

The size function retrieves the number of key-value pairs currently stored in the map. It returns the count of elements, allowing you to gauge the map's capacity and manage resources accordingly.

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

   int main() {
      map<int, string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
      cout << "Size: " << m.size();
      return 0;
   }
```

Expected Output: Size: 3

Lower Bound

Get an iterator to the first element that is not less than the given key.

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

int main() {
   map<int, string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
   auto it = m.lower_bound(2);
   cout << "Lower Bound: " << it->first << ": " << it->second;
```

```
    return 0;
}
```

Expected Output: Lower Bound: 2: two

Upper Bound

Get an iterator to the first element that is greater than the given key.

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

int main() {
    map<int, string> m = {{1, "one"}, {2, "two"}, {3, "three"}};
    auto it = m.upper_bound(2);
    cout << "Upper Bound: " << it->first << ": " << it->second;
    return 0;
}
```

Expected Output: Upper Bound: 3: three

Multimap

The `multimap` container allows storing multiple values for the same key in a sorted order.

Equal Range

The equal_range function provides a range of elements with a specific key in a multimap. It returns a pair of iterators: the first iterator points to the first element with the specified key, and the second iterator points just past the last element with that key.

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

int main() {
    multimap<int, string> mm;
    mm.insert({1, "one"});
    mm.insert({1, "uno"});
    auto range = mm.equal_range(1);
    for(auto it = range.first; it != range.second; ++it) {
        cout << it->first << ": " << it->second << "\n";
    }
    return 0;
}
```

Expected Output:

```
1: one
1: uno
```

Functions and Algorithms

Sort

The `sort` function in C++ sorts elements in a given range [first, last] into ascending order by default. It can also take a custom comparator for sorting.

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

int main() {
   vector<int> v = {3, 1, 4, 1, 5, 9};
   sort(v.begin(), v.end());
   for(int x : v) {
      cout << x << " ";
   }
   return 0;
}
```

Expected Output: 1 1 3 4 5 9

Accumulate

The `accumulate` function calculates the sum of a range of elements [first, last]. It is part of the <numeric> library.

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

int main() {
   vector<int> v = {1, 2, 3, 4, 5};
   int sum = accumulate(v.begin(), v.end(), 0);
   cout << "Sum: " << sum;
   return 0;
}
```

Expected Output: Sum: 15

Count

The `count` function returns the number of occurrences of a value in a range [first, last].

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

int main() {
   vector<int> v = {1, 2, 3, 1, 2, 1};
   int cnt = count(v.begin(), v.end(), 1);
   cout << "Count of 1: " << cnt;
   return 0;
}
```

Expected Output: Count of 1: 3

Find

The `find` function searches for the first occurrence of a value in a range [first, last] and returns an iterator to it.

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

int main() {
   vector<int> v = {1, 2, 3, 4, 5};
   auto it = find(v.begin(), v.end(), 3);
```

```
    if(it != v.end()) cout << "Found: " << *it;
    else cout << "Not Found";
    return 0;
}
```

Expected Output: Found: 3

Comparator

A comparator is a function or functor that determines the order of elements during sorting or other operations. It is often passed to functions like `sort`.

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

bool compare(int a, int b) {
    return a > b; // Sort in descending order
}

int main() {
    vector<int> v = {3, 1, 4, 1, 5, 9};
    sort(v.begin(), v.end(), compare);
    for(int x : v) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: 9 5 4 3 1 1

Next Permutation

The `next_permutation` function rearranges elements into the next lexicographically greater permutation. If the current permutation is the largest, it transforms it into the smallest permutation.

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

int main() {
    vector<int> v = {1, 2, 3};
    next_permutation(v.begin(), v.end());
    for(int x : v) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: 1 3 2

Max Element

The `max_element` function returns an iterator to the largest element in a range [first, last].

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

int main() {
    vector<int> v = {1, 5, 3, 9, 7};
    auto it = max_element(v.begin(), v.end());
    cout << "Max Element: " << *it;
    return 0;
}
```

Expected Output: Max Element: 9

Reverse

The `reverse` function reverses the order of elements in a range [first, last].

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

int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    reverse(v.begin(), v.end());
    for(int x : v) {
        cout << x << " ";
    }
    return 0;
}
```

Expected Output: 5 4 3 2 1

Pow

The `pow` function, found in the <cmath> library, raises a base number to the power of an exponent.

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

int main() {
    double base = 2.0, exp = 3.0;
    double result = pow(base, exp);
    cout << "2^3 = " << result;
    return 0;
}
```

Expected Output: 2^3 = 8

Conclusion

The C++ Standard Template Library (STL) is a crucial component of modern C++ programming, offering a suite of highly efficient and reusable template classes and functions. Key elements of the STL include containers, algorithms, functions, and iterators, each designed to streamline and enhance common programming tasks. With its rich set of features, such as dynamic arrays (vectors), linked lists (lists), deques, stacks, queues, and associative containers (sets, maps), the STL simplifies data management and operations, promotes code reusability, and minimizes errors. Mastery of the STL is essential for developing high-performance C++ applications.