STL C++ Components Explained

September 4, 2024

Sure, I'll provide detailed explanations and code snippets for each of the components and functions you mentioned in the Standard Template Library (STL) in C++. Let's go step by step.

1. STL Overview

The Standard Template Library (STL) in C++ is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, stacks, queues, and more. The STL has four components:

- **Containers**: These store data.
- **Algorithms**: These process the data.
- **Iterators**: These are used to traverse the data.
- **Functors**: These are function objects.

2. Containers

Containers are objects that store data. The STL provides several container types:

- Sequence Containers: `vector`, `deque`, `list`, `array`
- **Associative Containers**: **`set`**, **`multiset`**, **`map`**, **`multimap`**
- **Unordered Associative Containers**: `unordered_set` `unordered_multiset` `unordered_map`, `unordered_multimap`
- Container Adapters: `stack`, `queue`, `priority_queue`

Example: Vector

cpp

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

int main() {
    // Creating a vector of integers
    vector<int> vec = {1, 2, 3, 4, 5};

    // Adding an element to the vector
```

```
        vec.push_back(6);

        // Accessing elements using an iterator
        vector<int>::iterator it;
        for (it = vec.begin(); it != vec.end(); ++it) {
        // Output: 1 2 3 4 5 6  cout << *it << " ";
        }

        // Removing the last element
        vec.pop_back();       // vec is now {1, 2, 3, 4, 5}

        // Accessing the first and last element
        cout << "\nFirst element: " << vec.front() << endl; // Output: 1
        cout << "Last element: " << vec.back() << endl;                  // Output: 5

        // Getting the size of the vector
        cout << "Size of vector: " << vec.size() << endl;                // Output: 5

        return 0;
    }
```

Comments are included in the code to explain each function's use.

Common Functions of Vector:

- `` `push_back(value)` ``: Adds an element to the end of the vector.
- `` `pop_back()` ``: Removes the last element of the vector.
- `` `front()` ``: Returns the first element.
- `` `back()` ``: Returns the last element.
- `` `size()` ``: Returns the number of elements in the vector.
- `` `empty()` ``: Checks whether the vector is empty.
- `` `clear()` ``: Removes all elements from the vector.
- `` `insert(position, value)` ``: Inserts an element at the given position.
- `` `erase(position)` ``: Removes an element at the given position.
- `` `begin()` ``, `` `end()` ``: Returns an iterator to the beginning and end of the vector.

3. Iterators

Iterators are used to point to the elements of a container. They are similar to pointers in C++.

Example: Using Iterators

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

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

        // Iterator to traverse the vector
        vector<int>::iterator it;
        for (it = vec.begin(); it != vec.end(); ++it) {
        // Output: 10 20 30 40     cout << *it << " ";
        }

        // Reverse iterator to traverse the vector in reverse
        vector<int>::reverse_iterator rit;
        for (rit = vec.rbegin(); rit != vec.rend(); ++rit) {
            cout << *rit << " ";             // Output: 40 30 20 10
```

```
    }

    return 0;
}
```

Comments are included in the code to explain the iterator usage.

Common Functions of Iterators:

- **`begin()`**, **`end()`**: Returns an iterator to the beginning and end.
- **`rbegin()`**, **`rend()`**: Returns a reverse iterator to the beginning and end.
- **`advance(iterator, n)`**: Advances the iterator by **`n`** positions.
- **`distance(start, end)`**: Returns the number of elements between two iterators.

4. Stack

A stack is a container adapter that operates in a last-in, first-out (LIFO) manner.

Example: Stack

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

int main() {
    stack<int> s;

    // Pushing elements onto the stack
    s.push(1); s.push(2); s.push(3);



    // Top element
    cout << "Top element: " << s.top() << endl;                // Output: 3

    // Popping elements
    s.pop();     // Removes 3
    cout << "Top element after pop: " << s.top() << endl;              // Output: 2

    // Checking if stack is empty
    if (s.empty()) {
        cout << "Stack is empty" << endl;
    } else {
        cout << "Stack is not empty" << endl;
    }

    // Stack size
    cout << "Stack size: " << s.size() << endl;                // Output: 2

    return 0;
}
```

Comments are included in the code to explain each function's use.

Common Functions of Stack:

- **`push(value)`**: Adds an element to the top of the stack.

- `pop()` : Removes the top element of the stack.
- `top()` : Returns the top element of the stack. :
- `empty()`Checks whether the stack is empty.
- `size()` : Returns the number of elements in the stack.

5. Queue

A queue is a container adapter that operates in a first-in, first-out (FIFO) manner.

Example: Queue

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

int main() {
    queue<int> q;

    // Adding elements to the queue
    q.push(10); q.push(20); q.push(30);



    // Accessing the front and back element
    cout << "Front element: " << q.front() << endl;             // Output: 10 //
    cout << "Back element: " << q.back() << endl;               Output: 30

    // Removing elements from the queue
    q.pop();     // Removes 10

    // Checking the front element after pop
    cout << "Front element after pop: " << q.front() << endl;             // Output: 20

    // Queue size
    cout << "Queue size: " << q.size() << endl;             // Output: 2

    return 0;
}
```

Comments are included in the code to explain each function's use.

Common Functions of Queue:

- **`push(value)`**: Adds an element to the back of the queue.
- **`pop()`**: Removes the front element of the queue.
- **`front()`**: Returns the front element of the queue.
- **`back()`**: Returns the back element of the queue.
- **`empty()`**: Checks whether the queue is empty.
- **`size()`**: Returns the number of elements in the queue.

6. Priority Queue

A priority queue is a container adapter that operates like a regular queue but orders its elements according to a priority. By default, it is a max-heap, but it can be customized to a min-heap using a comparator.

Example: Priority Queue (Max-Heap)

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

int main() {
    priority_queue<int> pq;

    // Pushing elements onto the priority queue
    pq.push(5);
    pq.push(10);
    pq.push(1);

    //  Top  element  (maximum  element  in  the  queue)
    cout << "Top element: " << pq.top() << endl;            // Output: 10

    // Removing elements
    pq.pop();     // Removes 10
    cout << "Top element after pop: " << pq.top() << endl;            // Output: 5

    return 0;
}
```

Comments are included in the code to explain each function's use.

Min-Heap with Comparator

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int main() {
    // Custom comparator for min-heap
    priority_queue<int, vector<int>, greater<int>> minHeap;

    minHeap.push(5);
    minHeap.push(10);
    minHeap.push(1);

    // Top element (minimum element in the queue)
    cout << "Top element in min-heap: " << minHeap.top() << endl;            // Output: 1

    minHeap.pop();     // Removes 1
    cout << "Top element after pop: " << minHeap.top() << endl;            // Output: 5

    return 0;
}
```

Comments are included in the code to explain how to implement a min-heap using a comparator.

Common Functions of Priority Queue:

- **`push(value)`**: Adds an element to the priority queue.
- **`pop()`**: Removes the top element of the priority queue.
- **`top()`**: Returns the top element of the priority queue.

- `empty()`: Checks whether the priority queue is empty.
- `size()` : Returns the number of elements in the priority queue.

7. Set and Multiset

A **`set`** is an associative container that contains a sorted set of unique objects. A `multiset` allows duplicate elements.

Example: Set

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

int main() {
        set<int> s;

        //    Inserting    elements
        s.insert(3);        s.insert(1);
        s.insert(2); s.insert(2);

                                // Duplicate elements are ignored

        // Traversing the set
        for (int x : s) {
                cout << x << " ";            // Output: 1 2 3
        }

        // Finding an element
        if (s.find(2) != s.end()) {
                cout << "\n2 found in set" << endl;
        } else {
                cout << "2 not found in set" << endl;
        }

        return 0;
}
```

Comments are included in the code to explain the set's usage.

Example: Multiset

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

int main() {
        multiset<int> ms;

        //    Inserting    elements
        ms.insert(3); ms.insert(1);
        ms.insert(2);
        ms.insert(2);
                                // Duplicate elements are allowed

        // Traversing the multiset
        for (int x : ms) {
                cout << x << " ";            // Output: 1 2 2 3
```

```
        }

        return 0;
    }
```

Comments are included in the code to explain the multiset's usage.

Common Functions of Set/Multiset:

- **`insert(value)`**: Adds an element to the set/multiset.
- **`find(value)`**: Returns an iterator to the element if found.
- **`erase(value)`**: Removes an element from the set/multiset.
- **`size()`**: Returns the number of elements in the set/multiset.
- **`empty()`**: Checks whether the set/multiset is empty.
- **`count(value)`**: Returns the number of occurrences of an element (useful in multiset).

8. Lower Bound and Upper Bound

- **Lower Bound**: Returns an iterator to the first element that is not less than the given value.
- **Upper Bound**: Returns an iterator to the first element that is greater than the given value.

Example: Lower Bound and Upper Bound in Set

cpp

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

    int main() {
        set<int> s = {1, 2, 4, 5, 6};

        // Finding lower bound of 4
        auto it = s.lower_bound(4); if (it
        != s.end()) {
            cout << "Lower bound of 4: " << *it << endl;          // Output: 4
        }

        // Finding upper bound of 4
        it = s.upper_bound(4);
        if (it != s.end()) {
            cout << "Upper bound of 4: " << *it << endl;          // Output: 5
        }

        return 0;
    }
```

Comments are included in the code to explain lower bound and upper bound.

9. Ordered Set vs Unordered Set

- **Ordered Set**: Elements are stored in sorted order. (e.g., **`set`**)
- **Unordered Set**: Elements are stored without any specific order. (e.g., `unordered_set`)

Example: Ordered Set vs Unordered Set

cpp

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

int main() {
        set<int> orderedSet = {3, 1, 4, 2};
        unordered_set<int> unorderedSet = {3, 1, 4, 2};

        // Ordered set
        cout << "Ordered set: ";
        for (int x : orderedSet) {
                cout << x << " ";           // Output: 1 2 3 4
        }

        //    Unordered    set    cout    <<
        "\nUnordered  set:  ";  for  (int  x  :
        unorderedSet) {
                cout << x << " ";           // Output may vary (no specific order)
        }

        return 0;
}
```

Comments are included in the code to explain the difference between ordered and unordered sets.

10. Map and Unordered Map

A **`map`** is an associative container that contains key-value pairs in sorted order based on the key. An **`unordered_map`** is similar, but the elements are not sorted.

Example: Map

cpp

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

int main() {
        map<int, string> m;

        // Inserting key-value pairs m[1] =
        "one";   m[2]    =    "two";    m[3]    =
        "three";


        // Traversing the map
        for (auto it = m.begin(); it != m.end(); ++it) {
                cout << it->first << " -> " << it->second << endl; // Output: // 1
                -> one // 2 -> two // 3 -> three



        }

        // Finding a key
        if (m.find(2) != m.end()) {
                cout << "Key 2 found with value: " << m[2] << endl;                // Output: two
        }
```

```cpp
    return 0;
}
```

Comments are included in the code to explain the map's usage.

Unordered Map

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

int main() {
    unordered_map<int, string> um;

    // Inserting key-value pairs um[1]
    = "one"; um[2] = "two"; um[3] =
    "three";


    // Traversing the unordered_map
    for (auto it = um.begin(); it != um.end(); ++it) {
        cout << it->first << " -> " << it->second << endl;
        // Output order may vary (no specific order)
    }

    return 0;
}
```

Comments are included in the code to explain the unordered_map's usage.

Common Functions of Map/Unordered Map:

- **`insert(pair<key, value>)`**: Adds a key-value pair to the map/unordered_map.
- **`find(key)`**: Returns an iterator to the key if found.
- **`erase(key)`**: Removes the key-value pair from the map/unordered_map.
- **`size()`**: Returns the number of elements in the map/unordered_map.
- **`empty()`**: Checks whether the map/unordered_map is empty.
- **`at(key)`**: Returns the value associated with the key.

11. Multimap

A **`multimap`** is an associative container that allows multiple elements with the same key.

Example: Multimap

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

int main() {
    multimap<int, string> mm;

    // Inserting elements
```

```
        mm.insert({1,  "one"});  mm.insert({2,
        "two"});     mm.insert({2,     "another
        two"});

        // Traversing the multimap
        for (auto it = mm.begin(); it != mm.end(); ++it) {
            cout << it->first << " -> " << it->second << endl; // Output: // 1
            -> one // 2 -> two // 2 -> another two



        }

        return 0;
    }
```

Comments are included in the code to explain the multimap's usage.

12. Sorting in Vector

You can sort a `vector` using the

You said:

ChatGPT said:

Sure, let's continue from where we left off.

12. Sorting in Vector

You can sort a **`vector`** using the **`sort()`** function from the **`<algorithm>`** header. By default, the **`sort()`** function sorts the elements in ascending order, but you can also specify a custom comparator to sort them in descending order or based on a custom condition.

Example: Sorting in Vector

```cpp
    #include <iostream>
    #include <vector>
    #include <algorithm>      // For sort()
    using namespace std;

    int main() {
        vector<int> vec = {3, 1, 4, 1, 5, 9};

        //      Sorting      in      ascending      order
        sort(vec.begin(),  vec.end());  cout  <<  "Sorted
        in ascending order: "; for (int x : vec) {

            cout << x << " ";          // Output: 1 1 3 4 5 9
        }

        // Sorting in descending order using a comparator
        sort(vec.begin(), vec.end(), greater<int>());
        cout << "\nSorted in descending order: ";
        for (int x : vec) {
            cout << x << " ";          // Output: 9 5 4 3 1 1
        }
```

```
        return 0;
}
```

Comments are included in the code to explain how to sort a vector.

Common Functions for Sorting:

- **`sort(begin, end)`**: Sorts the range **`[begin, end)`** in ascending order.
- **`sort(begin, end, comparator)`**: Sorts the range **`[begin, end)`** using a custom comparator.

13. Sorting in Array

Similar to sorting a **`vector`**, you can also sort an array using the `sort()` function.

Example: Sorting in Array

cpp

```cpp
#include <iostream>
#include <algorithm>      // For sort()
using namespace std;

int main() {
        int arr[] = {3, 1, 4, 1, 5, 9};
        int n = sizeof(arr) / sizeof(arr[0]);

        // Sorting the array in ascending order
        sort(arr, arr + n);
        cout << "Sorted array in ascending order: ";
        for (int i = 0; i < n; i++) {
                cout << arr[i] << " ";                // Output: 1 1 3 4 5 9
        }

        // Sorting the array in descending order using a comparator
        sort(arr, arr + n, greater<int>());
        cout << "\nSorted array in descending order: ";
        for (int i = 0; i < n; i++) {
                cout << arr[i] << " ";                // Output: 9 5 4 3 1 1
        }

        return 0;
}
```

Comments are included in the code to explain how to sort an array.

14. Next Permutation

The **`next_permutation`** function rearranges the elements in the range **`[first, last)`** into the next lexicographically greater permutation. If no such permutation exists, it rearranges the elements as the lowest possible order (i.e., sorted in ascending order).

Example: next_permutation

cpp

```cpp
#include <iostream>
#include <algorithm>      // For next_permutation
#include <vector>
using namespace std;

int main() {
        vector<int> vec = {1, 2, 3};

        // Generating the next permutation
        next_permutation(vec.begin(), vec.end());
        cout << "Next permutation: ";
        for (int x : vec) {
                cout << x << " ";          // Output: 1 3 2
        }

        // Generating the next permutation again
        next_permutation(vec.begin(), vec.end());
        cout << "\nNext permutation: ";
        for (int x : vec) {
                cout << x << " ";          // Output: 2 1 3
        }

        return 0;
}
```

Comments are included in the code to explain the use of `next_permutation`.

15. Built-in Functions

C++ provides several built-in functions for bit manipulation. Two of the most commonly used ones are `__builtin_popcount` and `__builtin_popcountll`.

__builtin_popcount

This function returns the number of set bits (1s) in the binary representation of an unsigned integer.

Example: __builtin_popcount

cpp

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

int main() {
        unsigned int x = 29;          // Binary: 11101

        // Counting the number of set bits int count = __builtin_popcount(x); cout <<
        "Number of set bits in " << x << " is " << count << endl;
                                                                                  // Output: 4

        return 0;
}
```

Comments are included in the code to explain the use of `__builtin_popcount`.

__builtin_popcountll

This function is similar to `__builtin_popcount` but works with `unsigned long long` integers.

Example: __builtin_popcountll

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

int main() {
    unsigned long long x = 1234567891234567890ULL;

    // Counting the number of set bits
    int count = __builtin_popcountll(x);
    cout << "Number of set bits in " << x << " is " << count << endl;

    return 0;
}
```

Comments are included in the code to explain the use of `__builtin_popcountll`.

Summary

- **STL Components**: Containers (vector, stack, queue, priority queue, set, multiset, map, unordered_map, multimap), Algorithms (sort, next_permutation), Iterators, Functors.
- **Vector**: Common container with various functions like **`push_back`**, **`pop_back`**, etc.
- **Stack**: LIFO structure.
- **Queue**: FIFO structure.
- **Priority Queue**: Allows custom ordering.
- **Set/Multiset**: Stores unique/duplicate elements.
- **Map/Unordered Map/Multimap**: Key-value pairs, with or without ordering.
- **Sorting**: Using **`sort()`** for vectors and arrays.
- **next_permutation**: Rearranges elements to the next lexicographic order.
- **Built-in Functions**: Bit manipulation like **`__builtin_popcount`** and **`__builtin_popcountll`**.

Feel free to ask if you need any more detailed explanations or have specific questions on any of these topics!

You said:

ChatGPT said:

Practicing STL (Standard Template Library) in C++ is a great way to strengthen your understanding of data structures and algorithms. Here are some practice problems that cover various components of STL:

1. Vector Problems

- **Problem 1**: Given a vector of integers, remove all even numbers and print the remaining elements.
- **Problem 2**: Find the second largest element in a vector without sorting it.

- **Problem 3**: Rotate a vector to the right by `k` positions.

2. Set/Multiset Problems

- **Problem 1**: Given a set of integers, find the sum of all unique elements.
- **Problem 2**: Given a multiset of integers, count the number of distinct elements.
- **Problem 3**: Check if two sets are disjoint (i.e., they have no common elements).

3. Map/Unordered Map Problems

- **Problem 1**: Given a list of words, count the frequency of each word using a map.
- **Problem 2**: Implement a phonebook using a map where the key is the name and the value is the phone number.
- **Problem 3**: Given an array of integers, find the number of pairs with a given sum using an unordered_map.

4. Stack/Queue Problems

- **Problem 1**: Implement a function to check if a given string of parentheses is balanced using a stack.
- **Problem 2**: Reverse the first **`k`** elements of a queue.
- **Problem 3**: Implement a stack using two queues.

5. Priority Queue Problems

- **Problem 1**: Find the **`k`** largest elements in an array using a priority queue.
- **Problem 2**: Implement a min-heap and max-heap using a priority queue.
- **Problem 3**: Merge **`k`** sorted arrays into a single sorted array using a priority queue.

6. Algorithm Problems

- **Problem 1**: Given a vector of integers, sort them in descending order using **`sort()`** and a custom comparator.
- **Problem 2**: Given a vector of integers, generate all possible permutations using `next_permutation()`.
- **Problem 3**: Find the longest subsequence of consecutive elements in an unsorted array using STL functions.

7. Miscellaneous Problems

- **Problem 1**: Given a string, find the most frequent character using an unordered_map.
- **Problem 2**: Implement a custom comparator for a priority queue to sort integers based on their number of set bits.
- **Problem 3**: Use **`lower_bound`** and **`upper_bound`** to find the range of elements equal to a given value in a sorted array.

Detailed Problem Examples:

Example 1: Balanced Parentheses

- **Problem**: Given a string consisting of the characters `(`, `)`, `{`, `}`, `[`, and `]`, determine if the input string is valid. An input string is valid if:

    1. Open brackets must be closed by the same type of brackets.
    2. Open brackets must be closed in the correct order.
- **Solution**: Use a stack to push open brackets and pop when a matching closing bracket is found.

```cpp
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isValid(string s) {
    stack<char> st;
    for (char c : s) {
        if (c == '(' || c == '{' || c == '[') {
            st.push(c);
        } else {
            if (st.empty()) return false;
            if ((c == ')' && st.top() != '(') ||
            (c == '}' && st.top() != '{') ||
            (c == ']' && st.top() != '[')) {
            return false;
            }
            st.pop();
        }
    }
    return st.empty();
}

int main() {
    string s = "{[()]}";
    cout << (isValid(s) ? "Valid" : "Invalid") << endl;                  // Output: Valid
    return 0;
}
```

Example 2: K Largest Elements in an Array

- **Problem**: Given an array of integers, find the `k` largest elements.
- **Solution**: Use a min-heap (priority queue) to keep track of the `k` largest elements.

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

vector<int> kLargest(vector<int>& arr, int k) {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    for (int i : arr) {
        minHeap.push(i);
        if (minHeap.size() > k) {
            minHeap.pop();
        }
    }
    vector<int> result;
    while (!minHeap.empty()) {
        result.push_back(minHeap.top());
        minHeap.pop();
    }
    return result;
}

int main() {
    vector<int> arr = {3, 1, 4, 1, 5, 9, 2, 6, 5};
```

```
    int k = 3;
    vector<int> result = kLargest(arr, k);
    for (int x : result) {
    cout << x << " ";                  // Output: 5 6 9
    }
    return 0;
}
```

These problems will help you practice various aspects of STL in C++. Try solving them and exploring different STL components and their functions!