**Sets**

In C++ STL (Standard Template Library), sets are a container that stores unique, ordered elements. A set is a collection of elements that are sorted in a specific order (usually ascending order) and have no repeated elements. Sets are implemented as balanced binary search trees (usually **Red-Black trees**) and have a logarithmic time complexity for insertion, deletion, and searching.

Sets in C++ STL are defined in the <**set**> header file. The most commonly used set in C++ is std::set, which is a template class. The type of elements that can be stored in a set is defined by the template parameter.

Here's an example of how to declare and use a set in C++ STL:

#include <iostream>

#include <set>

int main() {

std::**set<int>** mySet; // declare a set of integers

mySet.**insert**(10);

mySet.**insert**(20);

mySet.**insert**(30); // insert elements into the set

std::cout << "Size of mySet: " << mySet.**size**() << std::endl; // output: 3

if (mySet.**find**(20) != mySet.**end**()) {

std::cout << "20 is in the set." << std::endl; // output: 20 is in the set.

}

// Erase an element from the set

mySet.**erase**(20);

// Print all elements in the set

for (const auto& e : mySet) {

std::cout << e << std::endl;

}

// Accessing set using a for loop

set<int> s = {1, 2, 3, 4};

for (auto it = s.begin(); it != s.end(); it++) {

cout << \*it << " ";

}

// Output: 1 2 3 4

return 0;

}

**NOTE**

* The find() function for sets takes one argument, which is the value to be searched in the set. It returns an iterator to the element if it is found in the set, otherwise it returns an iterator to the end() of the set.

For instance, let s be a set.

Then, **s.end() == s.find(val)** if **val** is not present in the set.

* The find() method of set is not purely same as method being used with vectors that requires 3 arguments i.e start iterator, end iterator and value to be searched
* **set** does not provide subscript operator [ ] to access elements like vector, string. To access its elements **iterator** can be used
* **size()** method can be used to check size of a set like a vector, map.
* set\_name.**clear()** erases all elements from the set

**Unordered\_Set**

std::unordered\_set is an unordered associative container implemented as a hash table. It stores unique elements in no particular order, and provides constant O(1) average-case time complexity for most operations, including insertions, deletions, and searches.

Here are some key notes on std::unordered\_set:

1. Elements are stored based on their hash values, which are calculated using a hash function provided by the user or by the default hash function of the type of the element.

2. Since the order of elements is not fixed, iteration through the elements of an unordered set can be in any order and is not guaranteed to be sorted.

3. std::unordered\_set allows for duplicate detection of elements, but only stores unique elements. If you attempt to insert a duplicate element, it will not be added to the set.

4. The load\_factor of an unordered set is the average number of elements per bucket. A low load factor means that the set has many buckets with few elements, while a high load factor means that the set has fewer buckets with more elements. The default load factor is 1.0.

5. std::unordered\_set provides the same set of functions as std::set, including insert(), erase(), and find(), but with constant time complexity on average rather than log(n). However, the worst-case time complexity of some operations can be O(n) due to hash collisions.

6. std::unordered\_set can be useful for applications such as fast lookups, **removing duplicates**, and implementing set operations such as intersection and union. However, it may not be suitable for cases where the order of elements matters, or when you need to iterate through the elements in a specific order.

**C++ code to demonstrate the implementation of std::unordered\_set:**

#include <iostream>

#include <unordered\_set>

#include <string>

int main() {

// Declare an unordered\_set with string elements

std::**unordered\_set**<std::string> mySet;

// Insert elements into the set

mySet.**insert**("apple");

mySet.**insert**("banana");

mySet.**insert**("orange");

// Check if an element is present in the set

if (mySet.**find**("pear") != mySet.end()) {

std::cout << "Pear is in the set." << std::endl;

} else {

std::cout << "Pear is not in the set." << std::endl;

}

// Erase an element from the set

mySet.**erase**("banana");

// Print all elements in the set

for (const auto& e : mySet) {

std::cout << e << std::endl;

}

return 0;

}

**Unordered\_set method to remove Duplicates from a vector/string without affecting the relative Ordering**

vector<int> **removeDuplicates**(vector<int>& nums) {

**unordered\_set**<int> seen;

vector<int> result;

for (int num : nums) {

if (seen.**insert**(num).**second**) {

result.push\_back(num);

}

}

return result;

}

So, the line if (seen.**insert**(num).**second**) actually does two things at once:

1. It inserts the element num into the seen set (if it doesn't already exist) using the **insert()** function.

2. It accesses the **second** member of the std::pair returned by insert() to determine whether the insertion was successful or not.

This allows us to check whether the current element is a **duplicate** or not, and add it to the result vector only if it's a new unique element.

**NOTE**

* In the context of the seen.insert(num) statement, **first** would be an **iterator** to the element in the seen set corresponding to the value of num, and **second** would be a **boolean** value indicating whether the insertion was successful or not.