**Containers**

In programming, a **container** refers to a data structure that holds a collection of elements or objects of a certain type. It provides operations to add, remove, and access the elements stored within it.

Containers can be found in many programming languages and are used in a wide range of applications. They are particularly useful when dealing with large amounts of data that need to be organized and manipulated efficiently.

In C++, the **Standard Template Library** (STL) provides a collection of container classes that are widely used. These include **vectors**, **arrays**, **lists**, **maps**, and **sets**, among others. Each container class provides a specific set of operations and is designed to be used in different situations depending on the specific requirements of the program.

**Vector**

In C++, a **vector** is a dynamic array that can resize itself automatically when elements are added or removed. It is a **container** that allows you to store and manipulate a sequence of elements of the same data type.

Vectors provide the following benefits over traditional arrays:

1. They can grow or shrink in size dynamically as elements are added or removed.

2. They provide bounds checking to ensure that you don't access elements that are out of range.

3. They can be easily resized and copied.

**NOTE**

* **vector** can be found in <vector> header file
* Key methods of vector are:

push\_back(), pop\_back(), size(), front()**,** back()

**Declaration of a vector container**

**1-Dimensional Vector**

E.g vector<int> myvector;

E.g vector<int> myvector(size, val);

// val : value supplied to all cells, size: Fixed size of vector

E.g vector<int> myvector{1,2,3,4,5};

E.g vector<string> myvector{“Apple”, ”Banana”, ”Orange”, ”Grapes”, ”Strawberry”};

**2-Dimensional Vector**

E.g. vector<vector<int>> myvector;

E.g. vector<vector<int>> myvector {

{1,2,3}, {4,5,6}, {7,8,9}

};

E.g. vector<vector<int>> matrix(m, vector<int>(n, val));

// val: value supplied to all cells, m: Rows in vector, n: Columns in vector

**Traversing in a vector E.g 1**

for(int i=0; i<myvector.size();i++){

cout<<myvector[i];

}

**Traversing in a vector E.g 2**

for(auto x : myvector){

cout<<x;

}

**Traverse over a vector using an iterator**

**Sample Source Code:**

#include <iostream>

#include <vector>

int main() {

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Create an iterator for the vector

std::**vector<int>::iterator it;**

// Iterate over the vector using the iterator

for (it = **numbers.begin();** it != **numbers.end();** ++it) {

std::cout << **\*it** << " ";

}

return 0;

}

In this example, we have a vector called **numbers** containing integers. We declare an iterator called it using the std::vector<int>::iterator type.

Then, in the for loop, we initialize the iterator it with numbers.begin(), which points to the first element of the vector.

The loop continues until the iterator it reaches numbers.end(), which points to the element after the last element of the vector.

In each iteration, we print the value pointed by the iterator using \*it. Finally, we output a newline character after the loop.

**OUTPUT**

1 2 3 4 5

**Useful Methods related to vector**

**size()** - Returns the Number of Elements in vector

**For 2D Vectors**

myvector.size() - Returns No. of Rows

myvector[0].size() - Returns No. of Columns

**max\_size()** - Returns the Number of Elements a vector can hold

**capacity()** - Returns the size of the vector which is currently allocated to it, Expressed as a number of Elements

**push\_back()** - Push New Supplied Element into the vector/**string** from the back

**pop\_back()** - Remove Element from the vector/**string** from the back

**front() -** Get the first Element from the vector/**string**

**back() -** Get the End Element from the vector/**string**

**begin()** - Returns an Iterator Pointing to the first Element of the vector

**end()**  - Returns an Iterator Pointing to The Theoretical Element Present Next to the Last Element of the Vector

**NOTE**

* end() Iterator does not point to any Element Present in the vector

**find()** - Finds the Element in the given range. Returns Iterator to the First Element if Found, Otherwise Returns the last.

*Sample*:

vector<int> v{1,2,3,4,5};

vector<int>:: **iterator** it; // An Iterator stores addresses of vector/**string** elements and does not work with primitive types

int search = 4; // Element to be searched in vector

it = find(v.begin(), v.end(), search);

**it == v.end()** - If element is not found in vector/string

*Output:*

*\*it - 4 //* On Successful Search, find will return the address of the first Element Found which actual value can be seen using \* before the iterator just like a pointer.

it - v.begin() - 3 // Difference between the Pointers

**Find Method Key Findings**

* We can **subtract (-)** iterators to find the distance between 2 iterators produced by the find method.

E.g

vector<int> cont = {1, 2, 2, 3, 4, 5};

vector<int>::vector it1 = find(v.begin(), v.end(), 2);

vector<int>::vector it2 = find(v.begin(), v.end(), 3);

// Here 1st  occurrence of 2 is at index 1 and 3 is at index 4

cout << (it - it2); // -2

cout << (it2 - it1); // 2

cout << abs(it1 - it2); // 2

**sort()**  - Sorts the vector either in Ascending or Descending Order.

*Assume the Sample vector be v :*

*The syntax* for ***Ascending*** *order* sort: sort(v.begin(), v.end())

*The syntax* for ***Descending*** *order* sort: sort(v.begin(), v.end(), greater<T>())

**NOTE**

* greater<T>() is known as comparator, placed as 3rd argument (optional)
* Both vector and string can be sorted with sort() and in case of string, sorting is done on the basis of ASCII value. ie alphanumeric string such as “bC78ip” will be sorted to “78Cbip”

**Custom Comparator in Sorting**

* **Custom Comparator** can be used in sort method as 3rd argument to achieve custom sorting
* For Example

std::vector<int> numbers = {5, 2, 9, 1, 5, 6};

// Sort using custom comparator

std::sort(numbers.begin(), numbers.end(), **customComparator**);

bool customComparator(int a, int b) {

return a > b; // Sort in descending order

}

**NOTE 1 :** Internally Call will be made in this sequence

customComparator(5, 2) - Comparing the first two elements, 5 and 2.

customComparator(2, 9) - Comparing the next two elements, 2 and 9.

customComparator(9, 1) - Comparing the next two elements, 9 and 1.

…

**NOTE 2**

* The behavior of the std::sort algorithm in C++ depends on the return value of the comparator function you provide. Specifically, it relies on the relationship between the return value and the desired order of elements:
* Comparator Returns **True**: If the comparator function returns true, it signifies that the first argument should come before the second argument in the sorted order. In other words, it indicates that a should precede b in the sorted sequence.
* Comparator Returns **False**: If the comparator function returns false, it signifies that the second argument should come before the first argument in the sorted order. In other words, it indicates that b should precede a in the sorted sequence.

**Create a custom comparator in C++ that sorts a sequence of pairs primarily based on the first value and, in the case of a tie, sorts based on the second value**

bool customComparator(const std::pair<int, int>& a, const std::pair<int, int>& b) {

if (a.first != b.first) {

return a.first < b.first; // Sort primarily by the first value in ascending order

}

return a.second < b.second; // In case of a tie, sort by the second value in ascending order

}

std::vector<std::pair<int, int>> pairs = {{5, 2}, {2, 9}, {1, 9}, {5, 1}, {6, 6}};

// Sort using custom comparator

std::sort(pairs.begin(), pairs.end(), customComparator);

**insert()** : The insert() function is used to insert an element at a specific position (**iterator required**) in a vector/**string**. It shifts all the elements to the right of the position by one position to make room for the new element, and inserts the new element at the specified position. However, if you insert an element at the beginning of the vector or at the end, you do not need to shift any elements, since there are no elements on the left or right to be shifted.

E.g : Assume a vector/**string** v, **iterator** for position, **value** to insert

Syntax: v.insert(v.begin() + index, value)

**reverse()** : the reverse() function from the algorithm header is used to reverse the characters in the string s. The begin() and end() member functions of the string class are used to get the iterators to the start and end of the string. Once the reverse() function is called with these iterators as arguments, the string s is reversed in-place.

E.g : string s = "hello world";

**reverse**(s.begin(), s.end());

**erase()** : To erase an element from a vector/**string** in C++, you can use the erase method, which **takes an iterator to the element to be erased** and not an index.

Here's an example that erases an element from a vector :

vector<int> v = {1, 2, 3, 4, 5};

v.erase(v.begin() + 2); // 1 2 4 5

string str = “abcde”;

str.erase(v.begin() + 2); // a b d e

**NOTE**

* Avoid using it for large string/vector as memory runtime error may occur

**clear()**: The clear() function is used to remove all the elements of the vector container, thus making it size 0.

Syntax:

vector\_name.**clear**()

Parameters: No parameters are passed.

Result: All the elements of the vector are removed (or **destroyed**).

**NOTE**

* The clear() method removes all the elements from a vector, but does not change its capacity. This means that the memory allocated for the vector remains the same, but the **size** of the vector becomes **zero**.

**resize():** resize() method can be used to resize the vector even after It is initialized or completely empty. **resize** method takes int value as an argument and fills that size of vector with default value: 0

Here is an example Implementation of 1D and 2D vectors resize:

// Resize vector using constructor

#include <iostream>

#include <vector>

using namespace std;

class Node {

vector<int> v1;

vector<vector<int>> v2;

public:

Node() {}

Node(int row) {

v1.resize(row);

}

Node(int row, int cols) {

v2.resize(row, vector<int>(cols));

}

void print1D() {

cout << "\nPrinting 1d vector\n";

for(auto x : v1) {

cout << x << " ";

}

}

void print2D() {

cout << "\nPrinting 2d vector\n";

for(auto x : v2) {

for(auto y : x) {

cout << y << " ";

}

cout << endl;

}

}

};

int main() {

Node obj(5); // 1d vector

//After resize (Default value: 0)

obj.print1D();

Node obj2(5, 5); //2d vector

//After resize (Default value: 0)

obj2.print2D();

return 0;

}

**Output:**

Printing 1d vector

0 0 0 0 0

Printing 2d vector

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

**numeric\_limits<T>::max()** : This is a template function in the C++ standard library that returns the maximum value of a given numeric type T.

This function is defined in the limits header file and can be used for various data types, including integers, floating-point numbers, and even some user-defined types.

For instance, if we call numeric\_limits<int>::max(), it will return the largest value that can be represented by an int on the current platform. Similarly, calling numeric\_limits<double>::max() will return the maximum finite value that a double can represent.

The max() function is particularly useful when we need to compare values of a numeric type, or when we want to initialize a variable with the largest possible value. By using numeric\_limits<T>::max() we can avoid hardcoding the maximum value for a given type, which can make our code more portable and less error-prone.

**count():** In C++ STL, count() is a function that is used to count the number of occurrences of a specific element in a given range of elements.

Here are some key points to keep in mind about the count() method:

1. count() is defined in the <**algorithm**> header file in C++ STL.

2. It takes two iterators as its arguments: the first iterator represents the start of the range, and the second iterator represents the end of the range.

3. It also takes the value of the element to be counted as a third argument.

4. The function returns the number of occurrences of the element in the given range.

5. The range of elements must be sorted in order for count() to work properly. If the range is not sorted, the behavior of count() is undefined.

6. The type of the value being counted must be comparable using the == operator, or an equivalent custom comparison function must be provided.

Here's an example usage of count():

#include <algorithm>

#include <iostream>

#include <vector>

#include <string>

using namespace std;

int main() {

// Usage of count() on Containers : Vector, Arrays

vector<int> v = {1, 2, 3, 4, 2, 2, 5};

int count = count(v.begin(), v.end(), 2);

cout << "Number of occurrences of 2: " << count << endl; // 3

// Usage of count() on string

string str = "the quick brown fox jumps over the lazy dog";

char ch = ‘t’;

int count = count(str.begin(), str.end(), ch);

cout << "Number of occurrences of \"" << ch << "\": "

<< count << endl; // 2

return 0;

}

**Count Method Key Findings:**

* We can only use the count method to count **characters** and **digits**, and not **string** to find substrings.
* The Following is the wrong use-case of **count** method:

string str = "the quick brown fox jumps over the lazy dog";

string substr = "the";

int count = count(str.begin(), str.end(), substr); // Error

**Find Minimum and Maximum Element in a vector**

* The Following Methods are defined in <**algorithm**> Header File
* **max\_element(iterator, iterator)**
  + Takes two iterator of a vector and return the maximum element value iterator present in the vector
  + For example,

vector<int> :: iterator it = max\_element(arr.begin(), arr.end());

* The above statement can be dereferenced to get actual value instead of an iterator, using

int maxElement = **\*max\_elemen**t(arr.begin(), arr.end());

* **min\_element(iterator, iterator)**
  + Same follow for min\_element() method, For example

int minElement = **\*min\_elemen**t(arr.begin(), arr.end());