**STL – Standard Template Library**

Used to take advantage of pre – build commonly used features in C++. Very good to speed up your programming and get the results in less time.

These are of three types

1. Containers
2. Iterators
3. Algorithms

**Why to use them?** Efficient, reusable, generic (Work with different types), standardized.

**Containers:** They are structures to store data.

Types:

1. Sequence Containers – stores in linear arrangement
   1. Vectors
   2. Deque
   3. List
2. Associative Containers – work in key – value pairs
   1. Set
   2. Map
3. Container Adopters – special containers like
   1. Stack
   2. Queue
   3. Priority queue.

**Vectors:**

Vectors are dynamic arrays. Means there size can be changed.

Show example of vector with some functions like size, capacity, pop\_back, push\_back etc.

#include <iostream>

#include <vector>

using namespace std;

int main(){

    // how to initialize vectors

    vector <int> vec1;

    vector <int> vec2 = {1,2,3};

    vector <int> vec3(3,5); // 3 is the size and 5 is the value filled

    vector <int> vec4(vec2); // vector 4 initialized with values of vector 2.

    vector<int> vec = {10, 20, 30};

    vec.push\_back(40); // Add an element at the end

    cout << "Elements in vector: ";

    for (int val : vec) { // Range-based for loop

        cout << val << " ";

    }

    cout << "\nSize: " << vec.size(); // Outputs 4

    cout << "\nCapacity: " << vec.capacity(); // capacity doubles as soon as we add a new member.

    vec.pop\_back();

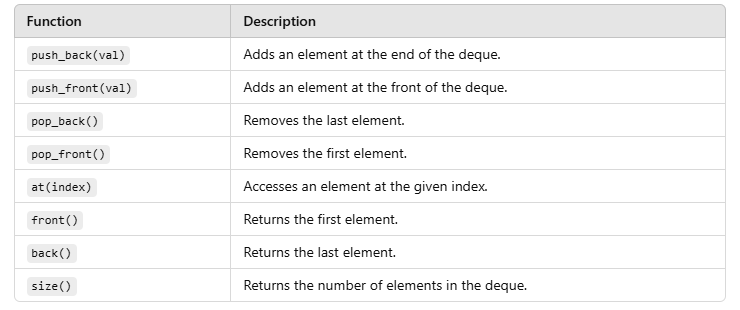
    cout << "\nSize: " << vec.size(); // Outputs 3 (size changes)

    cout << "\nCapacity: " << vec.capacity(); // capacity remains same.

    return 0;

}

**Deque** – dynamic array like linear container which allows insertions at both ends.



Code implementation of deque

#include <iostream>

#include <deque>

using namespace std;

int main() {

    deque<int> dq;

    // Adding elements to both ends

    dq.push\_back(10);

    dq.push\_front(20);

    dq.push\_back(30);

    // Displaying the deque

    cout << "Deque elements: ";

    for (int val : dq) {

        cout << val << " ";

    }

    cout << endl;

    // Accessing elements

    cout << "First element: " << dq.front() << endl;

    cout << "Last element: " << dq.back() << endl;

    cout << "Element at index 1: " << dq.at(1) << endl;

    // Removing elements

    dq.pop\_front();

    dq.pop\_back();

    cout << "Deque after pop operations: ";

    for (int val : dq) {

        cout << val << " ";

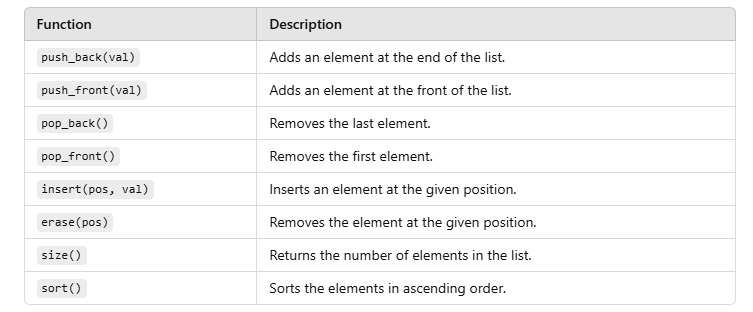
    }

    return 0;

}

LIST – Doubly linked list (Not now)

A list is a doubly linked list, allowing efficient insertion and deletion anywhere in the container.



Code implementation

#include <iostream>

#include <list>

using namespace std;

int main() {

    list<int> lst;

    // Adding elements

    lst.push\_back(10);

    lst.push\_front(20);

    lst.push\_back(30);

    // Displaying the list

    cout << "List elements: ";

    for (int val : lst) {

        cout << val << " ";

    }

    cout << endl;

    // Inserting and erasing

    auto it = lst.begin();

    advance(it, 1); // Move iterator to the second position

    lst.insert(it, 15); // Insert 15 at the second position

    cout << "List after insertion: ";

    for (int val : lst) {

        cout << val << " ";

    }

    cout << endl;

    lst.erase(it); // Erase the second element

    cout << "List after erasing second element: ";

    for (int val : lst) {

        cout << val << " ";

    }

    return 0;

}

MAP

A map stores key-value pairs.

#include <iostream>

#include <map>

using namespace std;

int main() {

    map<string, int> scores;

    scores["Alice"] = 90;

    scores["Bob"] = 85;

    cout << "Bob's score: " << scores["Bob"] << endl;

    for (const auto& pair : scores) {

        cout << pair.first << ": " << pair.second << endl;

    }

    return 0;

}

STACK - A stack follows the Last In, First Out (LIFO) principle.

#include <iostream>

#include <stack>

using namespace std;

int main() {

    stack<int> s;

    s.push(10); s.push(20); s.push(30);

    while (!s.empty()) {

        cout << "Top: " << s.top() << endl;

        s.pop();

    }

    return 0;

}

ITERATORS - Iterators are objects in C++ Standard Library (STL) that allow you to traverse (or iterate over) the elements in a container, such as arrays, maps, deques, and others.

Show example of one code first

**Input Iterator -** Reading and printing elements from a vector.

#include <iostream>

#include <vector>

using namespace std;

int main() {

    vector<int> vec = {10, 20, 30, 40, 50};

    vector<int>::iterator it;

    cout << "Reading elements using Input Iterator: ";

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

        cout << \*it << " "; // Read the element

        // \*it = 100; // Error: Input iterators do not allow modification

    }

    return 0;

}

**Output Iterator** - An output iterator allows **writing** values to a container

#include <iostream>

#include <vector>

#include <iterator>

using namespace std;

int main() {

    vector<int> vec(5); // Vector with 5 default-initialized elements

    vector<int>::iterator it = vec.begin();

    cout << "Writing elements using Output Iterator: ";

    for (int i = 1; it != vec.end(); ++it, ++i) {

        \*it = i \* 10; // Write a value

        cout << \*it << " "; // Display the value just written

    }

    return 0;

}

Common example to show all types of iterators

#include <iostream>

#include <vector>

#include <iterator>

using namespace std; // Removes the need for std:: prefixes

int main() {

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

    // Input Iterator Example - Reading elements

    cout << "Input Iterator: ";

    for (vector<int>::iterator it = myVector.begin(); it != myVector.end(); ++it) {

        cout << \*it << " "; // Reading elements

    }

    cout << endl;

    // Output Iterator Example - Writing elements

    cout << "Output Iterator: ";

    vector<int> outputVector(5); // Vector of size 5

    fill(outputVector.begin(), outputVector.end(), 0); // Initialize with 0

    int value = 1;

    for (vector<int>::iterator it = outputVector.begin(); it != outputVector.end(); ++it) {

        \*it = value++; // Writing elements

    }

    for (vector<int>::iterator it = outputVector.begin(); it != outputVector.end(); ++it) {

        cout << \*it << " "; // Reading elements to display

    }

    cout << endl;

    // Forward Iterator Example - Traversing forward

    cout << "Forward Iterator: ";

    for (vector<int>::iterator it = myVector.begin(); it != myVector.end(); ++it) {

        cout << \*it << " ";

    }

    cout << endl;

    // Bidirectional Iterator Example - Traversing backward

    cout << "Bidirectional Iterator: ";

    for (vector<int>::reverse\_iterator rit = myVector.rbegin(); rit != myVector.rend(); ++rit) {

        cout << \*rit << " ";

    }

    cout << endl;

    // Random Access Iterator Example - Direct access

    cout << "Random Access Iterator: ";

    for (size\_t i = 0; i < myVector.size(); ++i) {

        cout << myVector[i] << " "; // Direct access using offset

    }

    cout << endl;

    // Another way using iterator arithmetic

    vector<int>::iterator it = myVector.begin();

    cout << "Element at index 3 using iterator: " << \*(it + 3) << endl;

    return 0;

}

**STL Algorithms**

STL provides powerful algorithms for common operations like sorting, searching, and transformations. These algorithms work on iterators, making them compatible with all containers.

Common Algorithms

1. Sort
2. Find
3. Count
4. Accumulate

**Sort example**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

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

    sort(vec.begin(), vec.end());

    cout << "Sorted elements: ";

    for (int val : vec) {

        cout << val << " ";

    }

    return 0;

}

Find example

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

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

    auto it = find(vec.begin(), vec.end(), 30);

    if (it != vec.end()) {

        cout << "Element 30 found at index: " << distance(vec.begin(), it);

    } else {

        cout << "Element 30 not found!";

    }

    return 0;

}

Accumulate

#include <iostream>

#include <vector>

#include <numeric>

using namespace std;

int main() {

    vector<int> vec = {1, 2, 3, 4};

    int sum = accumulate(vec.begin(), vec.end(), 0); // Start summing with 0

    cout << "Sum of elements: " << sum;

    return 0;

}