Bangladesh Army University of Science & Technology, Saidpur

Lab Manual

Course Code: CSE 1204

Course Title: Object Oriented Programming Language -I Sessional.

Topic: STL (Standard Template Library) Vector & Set.

Prepared By:

Md. Manirujjaman

Lecturer, BAUST

STL (Standard Template library)

The Standard Template Library (STL) in C++ is a powerful library that provides a set of generic classes and functions to perform common data operations. It simplifies the implementation of data structures and algorithms, allowing programmers to write efficient and reusable code. The STL is composed of four main components:

Containers:

These are data structures used to store collections of data. Examples include:

i) vector: A dynamic array.
ii) list: A doubly-linked list.
iii) set: A collection of unique elements.
iv) map: A collection of key-value pairs.
v) queue and stack: For FIFO (first-in, first-out) and LIFO (last-in, first-out) operations, respectively.

Algorithms:

STL provides a variety of algorithms for sorting, searching, modifying, and manipulating container elements. Examples include:

i) sort(): Sorts the elements.
ii) find(): Searches for an element.
iii) reverse(): Reverses the order of elements.

Iterators:

These are used to iterate over the elements in containers. They work similarly to pointers and are used to access elements in a container while abstracting the underlying data structure.

Function Objects (Functors):

These are objects that can be called as functions. They are often used to customize algorithms, allowing more flexibility and control over how operations are performed on container elements.

Vector

Introduction to vector in C++

The vector in C++ is a sequence container from the Standard Template Library (STL) that functions like a dynamic array. Unlike a regular array, a vector can grow and shrink dynamically, which makes it highly flexible and efficient when the size of the array is not known at compile time.

Key features of vector:

i) Dynamic Size: A vector can resize itself when new elements are added or removed.
ii) Element Access: Allows access to elements using an index, iterators, or member functions.
iii) Performance: Elements are stored in contiguous memory, meaning access is fast (constant time). The operations like insertions and deletions at the end of the vector are amortized constant time, while operations at the middle or front are linear in time complexity.

Important Member Functions of vector

Element Access Functions:

i) at(): Access an element at a given position with bounds checking.
ii) []: Access an element at a given position without bounds checking.
iii) front(): Access the first element.
iv) back(): Access the last element.
v) data(): Returns a pointer to the underlying array.

Modifiers:

i) insert(): Inserts elements at a specific position.
ii) emplace(): Inserts elements by constructing them in-place.
iii) push_back(): Adds an element at the end of the vector.
iv) emplace_back(): Adds an element at the end by constructing it in place.
v) pop_back(): Removes the last element.
vi) resize(): Changes the size of the vector.
vii) swap(): Swaps the contents of two vectors.
viii) erase(): Removes elements at specific positions.
ix) clear(): Clears all elements in the vector.

Example -01: Basic Operations of Vector.

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

int main() {
    // Declarations of vectors
    vector<int> arr1; // Empty vector
    vector<int> arr2(5, 20); // Vector with 5 elements, all
initialized to 20
    vector<int> arr3 = {1, 2, 3, 4, 5}; // Vector initialized with
list
    vector<int> arr4 {1, 2, 3, 4, 5}; // Uniform initialization

    // Printing elements of arr2
    cout << "arr2:" << endl;
    for(int i = 0; i < arr2.size(); i++) {
        cout << " " << arr2[i] << " ";
    }
    cout << endl;

    // Printing elements of arr3 using at() method
    cout << "arr3:" << endl;
    for(int i = 0; i < arr3.size(); i++) {
        cout << " " << arr3.at(i) << " ";
    }
    cout << endl;

    // Printing elements of arr3 using range-based for loop
    cout << "arr3 again:" << endl;
    for(int i : arr3) {
        cout << i << " ";
    }
    cout << endl;

    // Printing elements of arr4 using data() method
    cout << "arr4:" << endl;
    int* p = arr4.data(); // Get pointer to the underlying array
    for(int i = 0; i < arr4.size(); i++) {
        cout << *(p + i) << " "; // Using pointer to access elements
    }
    cout << endl;

    // Taking input from the user and using push_back to add elements
    int n, read;
    cout << "Enter size of vector:" << endl;
    cin >> n;
    cout << "Enter the elements of the vector" << endl;
    for(int i = 0; i < n; i++) {
        cin >> read;
```

```
        arr1.push_back(read); // Add element to the end of arr1
    }

    // Printing arr1
    cout << "arr1:" << endl;
    for(int i = 0; i < arr1.size(); i++) {
        cout << arr1[i] << " ";
    }
    cout << endl;

    // Using pop_back to remove last element and print the size
    cout << "Size of array after one pop_back of arr1:" << endl;
    arr1.pop_back(); // Remove last element
    cout << arr1.size() << endl;

    // Using resize to adjust the size of arr1
    cout << "Resizing of vector arr1" << endl;
    arr1.resize(10, 20); // Resize to 10 elements, new elements
initialized to 20
    cout << arr1.size() << endl;
    cout << "Print arr1:" << endl;
    for(int i = 0; i < arr1.size(); i++) {
        cout << arr1[i] << " ";
    }
    cout << endl;

    // Using insert to add an element in the middle of arr1
    cout << "Printing arr1 elements after inserting an element" <<
endl;
    arr1.insert(arr1.end() - 3, 30); // Insert 30 before last 3
elements
    for(int i = 0; i < arr1.size(); i++) {
        cout << arr1[i] << " ";
    }
    cout << endl;

    // Iterating through arr1 using iterator
    cout << "Iterating arr1 through iterator" << endl;
    vector<int>::iterator it;
    for(it = arr1.begin(); it != arr1.end(); it++) {
        cout << *it << " ";
    }

    // Erasing elements using erase()
    cout << "Erase elements" << endl;
    arr1.erase(arr1.begin(), arr1.begin() + 2); // Erase first 2
elements
    for(it = arr1.begin(); it != arr1.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
```

```
    // Clearing all elements from arr1
    cout << "Clear arr1" << endl;
    arr1.clear(); // Remove all elements from arr1
    cout << "Size of arr1: " << arr1.size() << endl;

    // Using swap to exchange contents of two vectors
    cout << "Swap operation" << endl;
    swap(arr1, arr3); // Swap contents of arr1 and arr3
    cout << "Size of arr1: " << arr1.size() << endl;
    cout << "Size of arr3: " << arr3.size() << endl;

    return 0;
}
```

Example -02: Vector in class

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

// Define the 'student' class with 'id' and 'name' attributes
class student
{
public:
    int id;      // Student ID
    string name; // Student name

    // Constructor to initialize student with an ID and a name
    student(int i, string n)
    {
        id = i;
        name = n;
    }
};

int main()
{
    // Declare a vector of 'student' objects
    vector<student> s =
    {
        {2, "Manirujjaman"},
        {3, "Kamrul"},
        {4, "Amirul"}
    };

    // Print the size of the vector
    cout << "Size of the vector: " << s.size() << endl;

    // Iterate through the vector using a range-based for loop
```

```
    cout << "Student List:" << endl;
    for (student i : s)
    {
        cout << i.id << " " << i.name << endl;
    }

    // Add a new student to the vector using push_back()
    s.push_back({5, "Anite"});

    // Print the updated size of the vector
    cout << "Size of the vector after push_back: " << s.size()
<< endl;

    // Iterate through the vector using an iterator
    cout << "Updated Student List (Using Iterator):" << endl;
    for (vector<student>::iterator i = s.begin(); i != s.end();
i++)
    {
        cout << i->id << " " << i->name << endl; // Accessing
elements via the iterator
    }

    // Remove the last student from the vector using pop_back()
    cout << "Applying pop_back to remove the last element" <<
endl;
    s.pop_back();

    // Print the size of the vector after pop_back
    cout << "Size of the vector after pop_back: " << s.size() <<
endl;

    // Iterate through the vector using a traditional for loop
    cout << "Student List after pop_back:" << endl;
    for (int i = 0; i < s.size(); i++)
    {
        cout << s[i].id << " " << s[i].name << endl; //
Accessing elements by index
    }

    return 0;
}
```

STL Algorithm (sort, find, reverse)

Example-03:

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

int main()
{
    // Initialize a vector with some elements
    vector<int> vec{1, 4, 5, 6, 8};

    // Use find() to check if the element '8' is present in the
vector
    auto it = find(vec.begin(), vec.end(), 8);
    if (it != vec.end())  // Check if the element is found
    {
        cout << "Element is found" << endl;
    }
    else
    {
        cout << "Element is not found" << endl;
    }

    // Sort the vector in ascending order
    sort(vec.begin(), vec.end());

    // Reverse the vector to display elements in descending
order
    reverse(vec.begin(), vec.end());

    // Print the elements of the vector
    cout << "Elements in descending order: ";
    for (int i : vec)
    {
        cout << i << " ";
    }

    return 0;
}
```

Set in STL

i) A set is an associative container that stores a sorted set of unique elements.
ii) The operations like insert, erase, and search in a set have logarithmic time complexity, making them efficient for large datasets.
iii) Elements are sorted by default according to their values (in ascending order for most basic types like integers).
iv) No duplicate elements are allowed in a set. If you attempt to insert a duplicate, it will simply be ignored.
v) Iteration can be done using iterators.
vi) [] operator is not supported for sets, hence attempting to access elements like Set[i] would result in a compilation error.

Example -04:

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

int main()
{
    // Initialize the set with duplicate values, but they
will be removed automatically
    set<int> Set = {1, 2, 3, 1, 1, 4, 5, 6, 7, 7, 8, 1,
20};

    // Inserting new elements
    Set.insert(11);  // Adds 11 to the set
    Set.insert(21);  // Adds 21 to the set

    // Erasing an element
    Set.erase(3);    // Removes element 3 from the set

    // Display elements using a range-based for loop
    for (int i : Set) {
        cout << i << " ";
    }
    cout << endl;

    // Using an iterator to display elements
    set<int>::iterator i;
    for (i = Set.begin(); i != Set.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;

    // Displaying size of the set
    cout << "Size of the set: " << Set.size() << endl;
```

```
    return 0;
}

//another example for descending order
#include<iostream>
#include<set>
using namespace std;

int main()
{
    // Using greater<int> to sort the set in descending
order
    set<int, greater<int>> Set = {1, 2, 3, 1, 1, 4, 5, 6,
7, 7, 8, 1, 20};

    // Inserting new elements
    Set.insert(11);
    Set.insert(21);

    // Erasing an element
    Set.erase(3);

    // Displaying elements in descending order
    for (int i : Set) {
        cout << i << " ";
    }
    cout << endl;

    // Displaying size of the set
    cout << "Size of the set: " << Set.size() << endl;

    return 0;
}
```

Example -05:

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

class student
{
public:
    int id;
    string name;

    student(int i, string n) : id(i), name(n) {}
```

```
    // Overloading the < operator to compare students by id
    bool operator<(const student &x) const {
        return id < x.id; // sorting by student ID
    }
};

int main()
{
    set<student> s = {{2, "Manirujjaman"}, {5, "Anite"},
{3, "Sagor"}};

    // Inserting a new student
    s.insert({4, "Kamrujjaman"});

    // Erasing the student with id 4, exactly matching the
object
    s.erase(student(4, "Kamrujjaman"));

    // Iterating and printing students using an iterator
    set<student>::iterator i;
    cout << "Using iterator:" << endl;
    for(i = s.begin(); i != s.end(); i++) {
        cout << i->id << " " << i->name << endl;
    }

    cout << endl;

    // Iterating using range-based for loop
    cout << "Using range-based for loop:" << endl;
    for(student i : s) {
        cout << i.id << " " << i.name << endl;
    }

    return 0;
}
```

Conclusion:

The Standard Template Library (STL) in C++ provides powerful, generic containers and algorithms that simplify data management. Containers like set and vector offer efficient ways to store and manipulate data. A set stores unique elements in a sorted order, ensuring no duplicates and providing efficient insertion, deletion, and search operations. In contrast, a vector is a dynamic array that allows efficient random access and dynamic resizing, making it ideal for scenarios where elements need to be frequently added or removed. Overall, STL enhances code efficiency, reusability, and maintainability by providing ready-to-use solutions for common data structure and algorithm tasks.