**Experiment No- 12 Date-**

**Aim – To study standard template library (STL)**

# Theory –

# Standard Template Library (STL)

The Standard Template Library (STL) in C++ is a powerful set of classes and functions designed for common data structures and algorithms. It provides flexible and efficient ways to manage and manipulate collections of data. STL includes components like containers, iterators, algorithms, and function objects, making it easier to write generic, reusable, and efficient code.

**Key Components of STL**

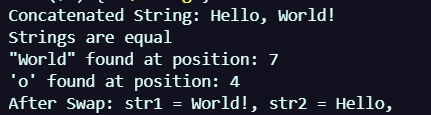
1. **Containers**: Collections of data that can hold multiple elements, such as arrays, linked lists, stacks, queues, and maps.
2. **Algorithms**: Functions that perform operations on data stored in containers, such as sorting, searching, counting, and transforming.
3. **Iterators**: Objects that provide a way to access elements in a container sequentially without exposing the underlying structure.
4. **Function Objects (Functors)**: Objects that can be used like functions and passed to algorithms to customize their behavior.

**Common STL Header Files**

Here is a list of commonly used STL header files and a brief description of the features they offer:

| **Header File** | **Description** |
| --- | --- |
| <vector> | Provides the vector container, a dynamic array that can resize automatically. |
| <list> | Implements a doubly linked list (list) for efficient insertion and deletion. |
| <deque> | Implements a double-ended queue (deque) that allows insertion and deletion at both ends. |
| <array> | Provides the array container, a fixed-size array with more functionality than a C array. |
| <stack> | Implements the stack container adapter for LIFO operations. |
| <queue> | Implements the queue container adapter for FIFO operations. |
| <priority\_queue> | Implements a priority\_queue, a queue where elements are arranged by priority. |
| <map> | Provides the map associative container, which stores key-value pairs sorted by keys. |
| <unordered\_map> | Implements unordered\_map, an associative container that stores key-value pairs with hash tables for fast access. |
| <set> | Provides the set associative container, which stores unique elements in sorted order. |
| <unordered\_set> | Provides unordered\_set, an associative container that stores unique elements in no particular order, using hash tables. |
| <algorithm> | Defines various algorithms such as sort, find, count, and accumulate. |
| <iterator> | Provides iterators, iterator traits, and iterator functions for working with STL containers. |
| <numeric> | Contains numeric algorithms such as accumulate, adjacent\_difference, and inner\_product. |
| <functional> | Contains function objects (functors) and other utilities for functional programming. |
| <utility> | Provides utility functions like pair, make\_pair, and other helper classes. |
| <memory> | Contains memory management utilities like unique\_ptr, shared\_ptr, and weak\_ptr. |
| <string> | Defines the string class for handling sequences of characters and string operations. |

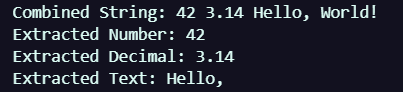
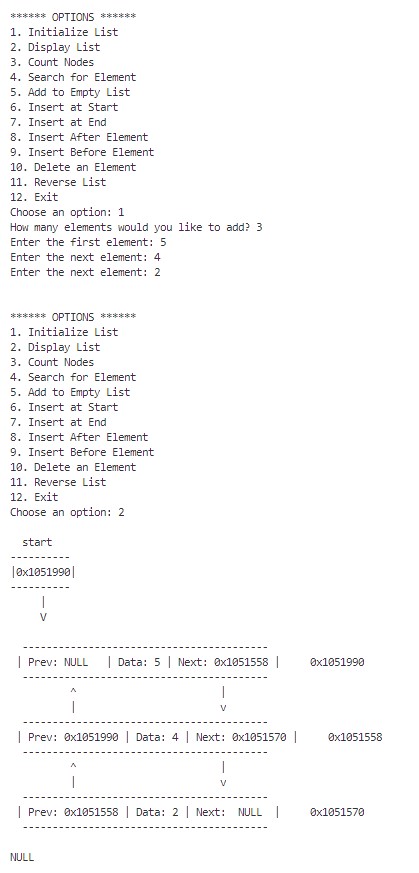
[A] Write a C++ program to implement exceptional handling concept (Divide by zero) using

exception rethrow mechanism

|  |  |
| --- | --- |
| **Program-**  #include <iostream>  #include <vector> *// Include the vector library*  using namespace std;  int *main*() {  *// Create a vector to store integer values*      vector<int> numbers;  *// Add elements to the vector using push\_back*      numbers.*push\_back*(10);      numbers.*push\_back*(20);      numbers.*push\_back*(30);      numbers.*push\_back*(40);    *// Display initial vector elements*      cout *<<* "Vector elements after push\_back operations: ";      for (int i = 0; i < numbers.*size*(); i++) {          cout *<<* numbers*[*i*]* *<<* " ";      }      cout *<<* *endl*;  *// Insert an element at a specific position*      numbers.*insert*(numbers.*begin*() *+* 2, 25); *// Insert 25 at index 2*  *// Display vector elements after insertion*      cout *<<* "Vector elements after insertion at index 2: ";      for (int i = 0; i < numbers.*size*(); i++) {          cout *<<* numbers*[*i*]* *<<* " ";      }      cout *<<* *endl*;  *// Access and modify an element*      numbers*[*3*]* = 35; *// Modify the element at index 3*  *// Display vector elements after modification*      cout *<<* "Vector elements after modifying index 3: ";      for (int i = 0; i < numbers.*size*(); i++) {          cout *<<* numbers*[*i*]* *<<* " ";      }      cout *<<* *endl*;  *// Remove the last element using pop\_back*      numbers.*pop\_back*();  *// Display vector elements after removing the last element*      cout *<<* "Vector elements after pop\_back: ";      for (int i = 0; i < numbers.*size*(); i++) {          cout *<<* numbers*[*i*]* *<<* " ";      }      cout *<<* *endl*;  *// Display the size and capacity of the vector* | OUTPUT –        cout *<<* "Current size of the vector: " *<<* numbers.*size*() *<<* *endl*;      cout *<<* "Current capacity of the vector: " *<<* numbers.*capacity*() *<<* *endl*;  *// Use range-based for loop to print vector elements*      cout *<<* "Vector elements using range-based for loop: ";      for (int num : numbers) {          cout *<<* num *<<* " ";      }      cout *<<* *endl*;  *// Clear all elements in the vector*      numbers.*clear*();      cout *<<* "Vector size after clear: " *<<* numbers.*size*() *<<* *endl*;      return 0;  } |

[B] Write a C++ program to implement a multi catch exception handling mechanism

|  |  |
| --- | --- |
| **Program –**    **#include <iostream>**  **#include <stdexcept>**  **#include <vector>**  **void *performOperations*(int index) {**  **std::vector<int> numbers = {1, 2, 3};**  ***// Throwing an exception if the index is invalid***  **if (index < 0) {**  **throw std::*invalid\_argument*("Index cannot be negative.");**  **}**  **if (index >= numbers.*size*()) {**  **throw std::*out\_of\_range*("Index is out of range.");**  **}**  ***// Access the element at the specified index***  **std::cout *<<* "Value at index " *<<* index *<<* ": " *<<* numbers*[*index*]* *<<* std::*endl*;**  **}**  **int *main*() {**  **int index;**  ***// Ask the user for an index***  **std::cout *<<* "Enter an index to access the numbers vector: ";**  **std::cin *>>* index;**  **try {**  ***performOperations*(index);**  **} catch (const std::invalid\_argument& e) {**  **std::cout *<<* "Caught invalid\_argument exception: " *<<* e.*what*() *<<* std::*endl*;**  **} catch (const std::out\_of\_range& e) {**  **std::cout *<<* "Caught out\_of\_range exception: " *<<* e.*what*() *<<* std::*endl*;**  **} catch (...) { *// Catch-all for any other exceptions***  **std::cout *<<* "Caught an unknown exception." *<<* std::*endl*;**  **}**  **return 0;**  **}** | **Output –** |

****[C] Write a C++ program to understand string stream processing concept

|  |  |
| --- | --- |
| **Program –**  **#include <iostream>**  **#include <stdexcept>**  **using namespace std;**  ***// Template class for Queue***  **template <typename T>**  **class Queue {**  **private:**  **static const int capacity = 10; *// Fixed size for simplicity***  **T data[capacity]; *// Array to store queue elements***  **int front; *// Index of the front of the queue***  **int rear; *// Index of the rear of the queue***  **int count; *// Current number of elements in the queue***  **public:**  ***// Constructor to initialize the queue***  ***Queue*() : *front*(0), *rear*(0), *count*(0) {} *// Simple constructor***  ***// Enqueue operation***  **void *enqueue*(T item) {**  ***// Check if the queue is full***  **if (count >= capacity) {**  ***// Throw an exception if capacity is exceeded***  **throw *overflow\_error*("Queue capacity exceeded!");**  **}**  ***// Insert the item at the rear***  **data[rear] = item;**  ***// Update the rear index, wrapping around if necessary***  **rear = (rear + 1) % capacity;**  ***// Increment the count of items***  **count++;**  **}**  ***// Dequeue operation***  **T *dequeue*() {**  ***// Check if the queue is empty***  **if (count <= 0) {**  ***// Throw an exception if trying to dequeue from an empty queue***  **throw *underflow\_error*("Queue is empty!");**  **}**  ***// Retrieve the item from the front***  **T item = data[front];**  ***// Update the front index, wrapping around if necessary***  **front = (front + 1) % capacity;**  ***// Decrement the count of items***  **count--;**  ***// Return the dequeued item***  **return item;**  **}**  ***// Get the number of items in the queue***  **int *size*() const {**  **return count; *// Return the current count of items***  **}**  ***// Check if the queue is empty***  **bool *isEmpty*() const {**  **return count == 0; *// Return true if count is zero***  **}**  ***// Function to get the maximum capacity of the queue***  **int *getCapacity*() const {**  **return capacity; *// Return the maximum capacity***  **}**  ***// Display the elements in the queue***  **void *display*() const {**  **if (*isEmpty*()) {**  **cout *<<* "Queue is empty.\n";**  **return;**  **}**  **cout *<<* "Queue elements: ";**  **for (int i = 0; i < count; ++i) {**  **cout << data[(front + i) % capacity] << " ";**  **}**  **cout *<<* "\n";**  **}**  **};**  ***// Interactive main function***  **int *main*() {**  **int n; *// Number of values to enqueue***  **Queue<int> queue; *// Create a queue for integers***  **int choice, value; *// Variables for user choice and value input***  **do {**  ***// Menu for user choices***  **cout *<<* "\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nChoose an option: ";**  **cin *>>* choice;** | **Output –**          **try {**  **switch (choice) {**  **case 1: *// Enqueue***  **cout *<<* "Enter number of values to enqueue (max " *<<* queue.*getCapacity*() *<<* "): ";**  **cin *>>* n;**  **if (n > queue.*getCapacity*()) {**  **throw *overflow\_error*("Cannot enqueue more than queue capacity.");**  **}**  **for (int i = 0; i < n; i++) {**  **cout *<<* "Enter value to enqueue: ";**  **cin *>>* value; *// Use 'value' for input***  **queue.*enqueue*(value); *// Call enqueue method***  **}**  **queue.*display*(); *// Display the queue***  **break;**  **case 2: *// Dequeue***  **value = queue.*dequeue*(); *// Call dequeue method***  **cout *<<* value *<<* " dequeued.\n";**  **queue.*display*(); *// Display the queue after dequeue***  **break;**  **case 3: *// Display***  **queue.*display*(); *// Display current queue***  **break;**  **case 4: *// Exit***  **cout *<<* "Exiting...\n";**  **break;**  **default:**  **cout *<<* "Invalid option. Please choose again.\n";**  **}**  **} catch (const overflow\_error& e) {**  ***// Catch overflow error if the queue is full***  **cerr *<<* e.*what*() *<<* "\n"; *// Output error message***  **} catch (const underflow\_error& e) {**  ***// Catch underflow error if the queue is empty***  **cerr *<<* e.*what*() *<<* "\n"; *// Output error message***  **}**  **} while (choice != 4); *// Continue until user chooses to exit***  **return 0; *// End of the program***  **}** |

**Conclusion – All the codes were successfully executed using the concepts of Exce*ption Handling.***