Problem 1: List Operations

Description:

Write a program that uses the std::list container to manage a collection of integers. Your program should perform the following operations:

Insert elements at the front and back of the list.

Remove elements from the front and back of the list.

Sort the list in ascending and descending order.

Reverse the list.

Display the elements of the list.

#include <iostream>

#include <list>

#include <algorithm>

class ListManager {

public:

void insertFront(int value) {

myList.push\_front(value);

}

void insertBack(int value) {

myList.push\_back(value);

}

void removeFront() {

if (!myList.empty()) {

myList.pop\_front();

}

}

void removeBack() {

if (!myList.empty()) {

myList.pop\_back();

}

}

void sortAscending() {

myList.sort();

}

void sortDescending() {

myList.sort(std::greater<int>());

}

void reverseList() {

myList.reverse();

}

void displayList() const {

for (const int& elem : myList) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

private:

std::list<int> myList;

};

int main() {

ListManager listManager;

listManager.insertFront(120);

listManager.insertBack(270);

listManager.insertFront(54);

listManager.insertBack(256);

std::cout << "List after inserting elements at front and back:" << std::endl;

listManager.displayList();

listManager.removeFront();

listManager.removeBack();

std::cout << "List after removing elements from front and back:" << std::endl;

listManager.displayList();

listManager.sortAscending();

std::cout << "List after sorting in ascending order:" << std::endl;

listManager.displayList();

listManager.sortDescending();

std::cout << "List after sorting in descending order:" << std::endl;

listManager.displayList();

listManager.reverseList();

std::cout << "List after reversing:" << std::endl;

listManager.displayList();

return 0;

}



Problem 2: Vector Manipulation

Description:

Create a program that uses the std::vector container to store a collection of floating-point numbers. The program should:

Add elements to the vector.

Remove elements from a specified position.

Find the maximum and minimum elements in the vector.

Calculate the average of the elements.

Display the elements of the vector.

#include <iostream>

#include <vector>

#include <algorithm>

#include <numeric>

void displayVector(const std::vector<float>& vec) {

for (const float& elem : vec) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

int main() {

std::vector<float> myVector;

myVector.push\_back(1.1f);

myVector.push\_back(2.2f);

myVector.push\_back(3.3f);

myVector.push\_back(4.4f);

std::cout << "Vector after adding elements:" << std::endl;

displayVector(myVector);

size\_t positionToRemove = 8;

if (positionToRemove < myVector.size()) {

myVector.erase(myVector.begin() + positionToRemove);

}

std::cout << "Vector after removing element at position " << positionToRemove << ":" << std::endl;

displayVector(myVector);

if (!myVector.empty()) {

float maxElement = \*std::max\_element(myVector.begin(), myVector.end());

std::cout << "Maximum element: " << maxElement << std::endl;

}

if (!myVector.empty()) {

float minElement = \*std::min\_element(myVector.begin(), myVector.end());

std::cout << "Minimum element: " << minElement << std::endl;

}

if (!myVector.empty()) {

float sum = std::accumulate(myVector.begin(), myVector.end(), 0.0f);

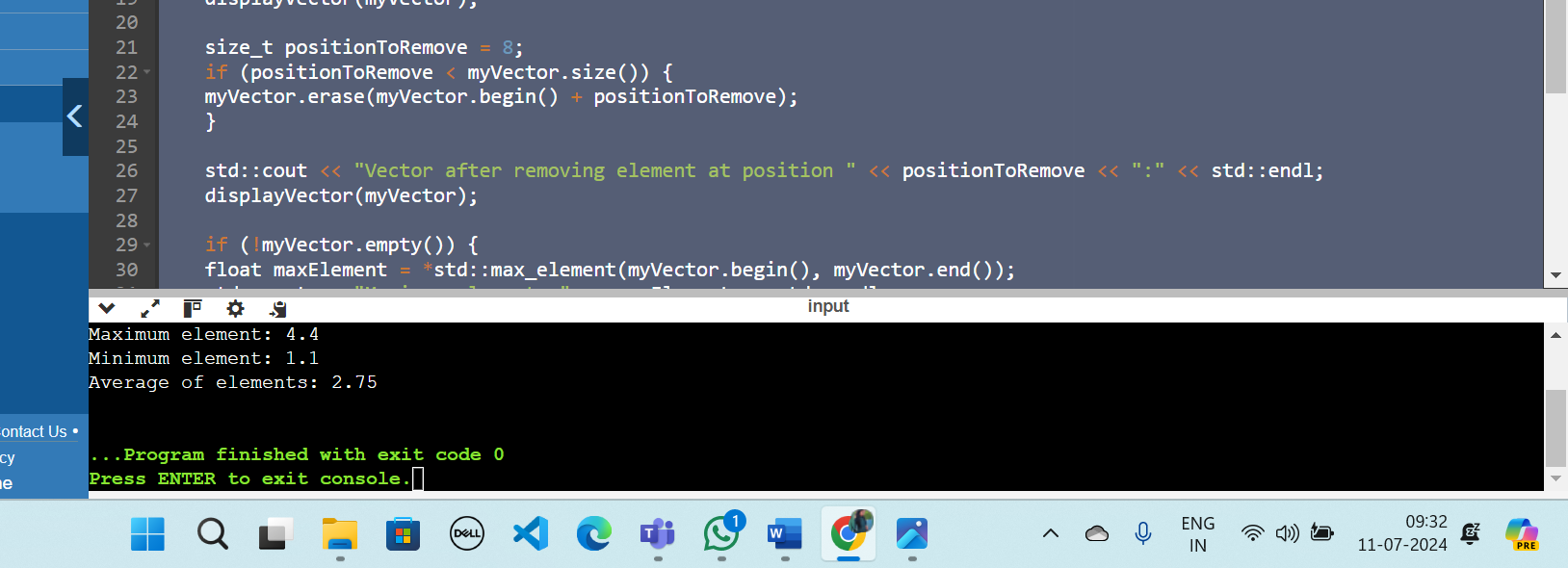
float average = sum / myVector.size();

std::cout << "Average of elements: " << average << std::endl;

}

return 0;

}



Problem 3: Queue Simulation

Description:

Implement a program using the std::queue container to simulate a ticketing system. The program should:

Add customers to the queue.

Serve customers (remove from front of the queue).

Display the current queue.

Display the number of customers served.

#include <iostream>

#include <queue>

class TicketingSystem {

public:

void addCustomer(const std::string& customer) {

customerQueue.push(customer);

std::cout << "Add customer: " << customer << std::endl;

}

void serveCustomer() {

if (!customerQueue.empty()) {

std::cout << "Serving customer: " << customerQueue.front() << std::endl;

customerQueue.pop();

customersServed++;

}

else {

std::cout << "No customers to serve." << std::endl;

}

}

void displayQueue() const {

std::queue<std::string> tempQueue = customerQueue;

std::cout << "Current queue: ";

while (!tempQueue.empty()) {

std::cout << tempQueue.front() << " ";

tempQueue.pop();

}

std::cout << std::endl;

}

void displayServedCount() const {

std::cout << "Number of customers served: " << customersServed << std::endl;

}

private:

std::queue<std::string> customerQueue;

int customersServed = 0;

};

int main() {

TicketingSystem ticketingSystem;

ticketingSystem.addCustomer("Customer 1");

ticketingSystem.addCustomer("Customer 2");

ticketingSystem.addCustomer("Customer 3");

std::cout << "Current state of the queue:" << std::endl;

ticketingSystem.displayQueue();

ticketingSystem.serveCustomer();

ticketingSystem.serveCustomer();

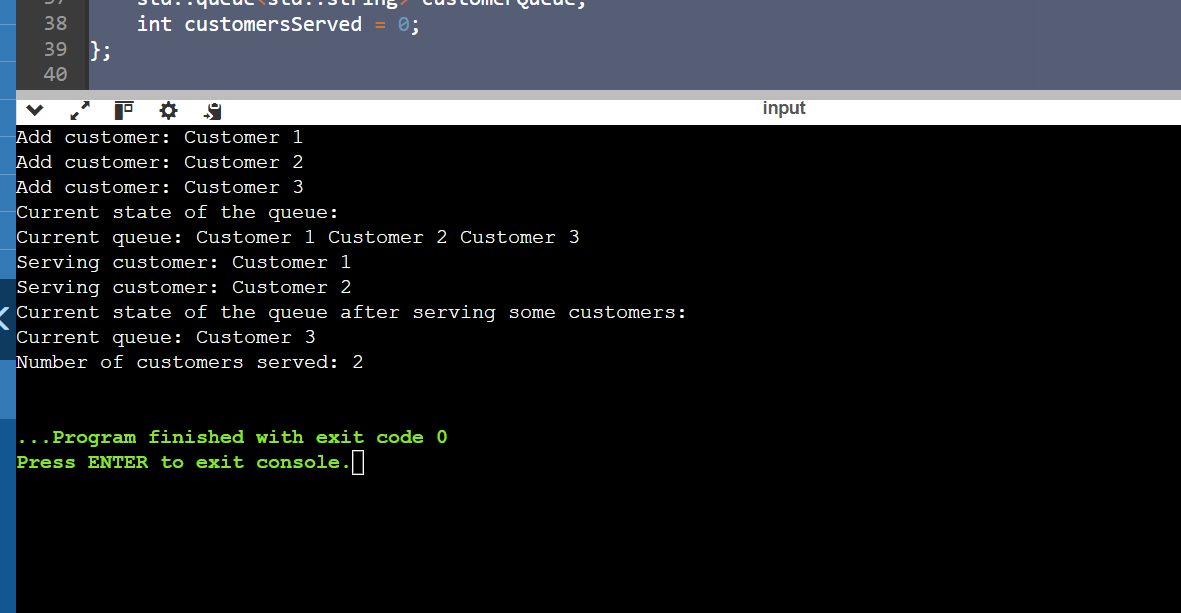
std::cout << "Current state of the queue after serving some customers:" << std::endl;

ticketingSystem.displayQueue();

ticketingSystem.displayServedCount();

return 0;

}



Problem 4: Stack Operations

Description:

Write a program using the std::stack container to evaluate a postfix expression. The program should:

Read a postfix expression.

Use a stack to evaluate the expression.

Display the result of the evaluation.

#include <iostream>

#include <stack>

#include <sstream>

#include <string>

bool isOperator(const char& token) {

return token == '+'||token == '-'||token == '\*'||token == '/';

}

float applyOperation(const char& operation, float operand1, float operand2) {

switch (operation) {

case '+': return operand1 + operand2;

case '-': return operand1 - operand2;

case '\*': return operand1 \* operand2;

case '/': return operand1 / operand2;

default: throw std::invalid\_argument("Invalid operation");

}

}

float evaluatePostfixExpression(const std::string& expression) {

std::stack<float> stack;

std::istringstream tokens(expression);

std::string token;

while (tokens >> token) {

if (isOperator(token[0]) && token.size() == 1) {

if (stack.size() < 2) {

throw std::invalid\_argument("Invalid postfix expression");

}

float operand2 = stack.top(); stack.pop();

float operand1 = stack.top(); stack.pop();

float result = applyOperation(token[0], operand1, operand2);

stack.push(result);

}

else {

stack.push(std::stof(token));

}

}

if (stack.size() != 1) {

throw std::invalid\_argument("Invalid postfix expression");

}

return stack.top();

}

void processPostfixExpression() {

std::string postfixExpression;

std::cout << "Enter a postfix expression: ";

std::getline(std::cin, postfixExpression);

try {

float result = evaluatePostfixExpression(postfixExpression);

std::cout << "The result of the evaluation is: " << result << std::endl;

} catch (const std::exception& e) {

std::cout << "Error: " << e.what() << std::endl;

}

}

int main() {

processPostfixExpression();

return 0;

}

