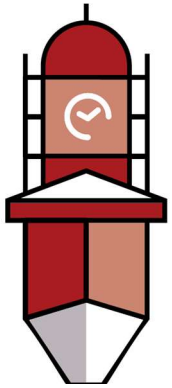


The University of Faisalabad

Data Structure lab Manual

Submitted to

Mam Irsha

Submitted by

Javed Mehmood

Reg No

BS-AI-029

	Table of Content	
Lab no		**Page**
01	Array	2
02	2D array	5
03	Vector	10
04	List	13
05	Stack	18
06	Queue	23
07	Dequeue	26
08	Tree	31

Data Structure
Lab Task

LAB NO 1

Array

Definition of Array :

In C++, an array is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow you to store multiple values of the same type in a single variable, which makes it easier to manage large amounts of data.

Syntax of Array :

The syntax for declaring an array in C++ is:

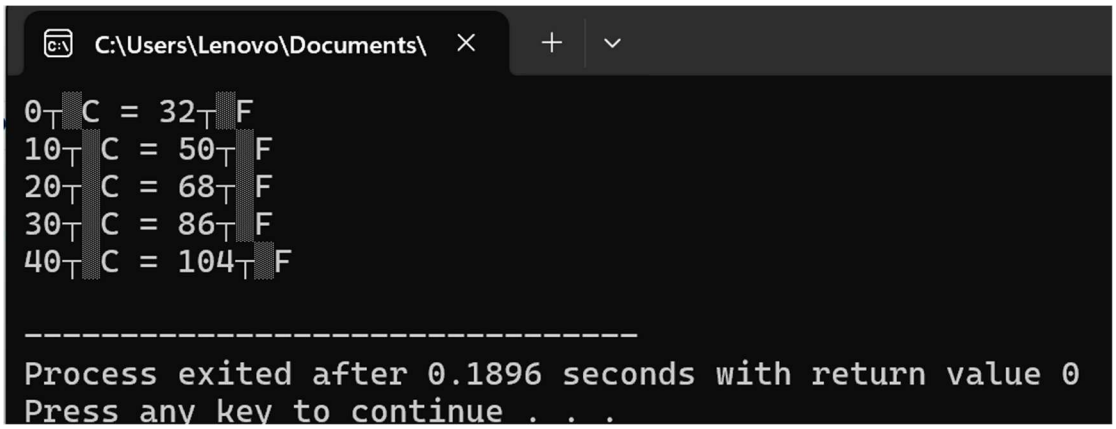
data_type array_name[array_size];

Where:

- **data_type**: Specifies the type of the elements (e.g., int, float, char).
- **array_name**: The name you choose to refer to the array.
- **array_size**: The number of elements the array will hold. This size must be a constant expression.

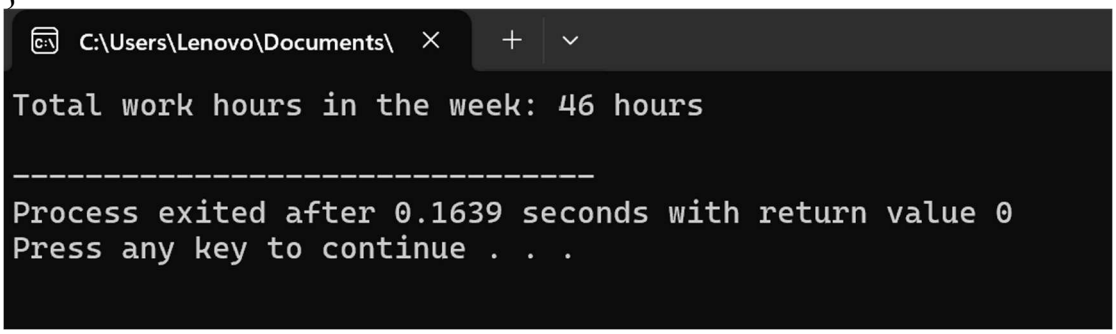
Examples:

```
#include <iostream>
using namespace std;
int main() {
   float celsius[5] = {0, 10, 20, 30, 40};
  float fahrenheit[5];
  for (int i = 0; i < 5; i++) {
      fahrenheit[i] = (celsius[i] * 9 / 5) + 32;
      cout << celsius[i] << "°C = " << fahrenheit[i] << "°F" << endl;
}
return 0;
  }
```



Emplooyes homework:

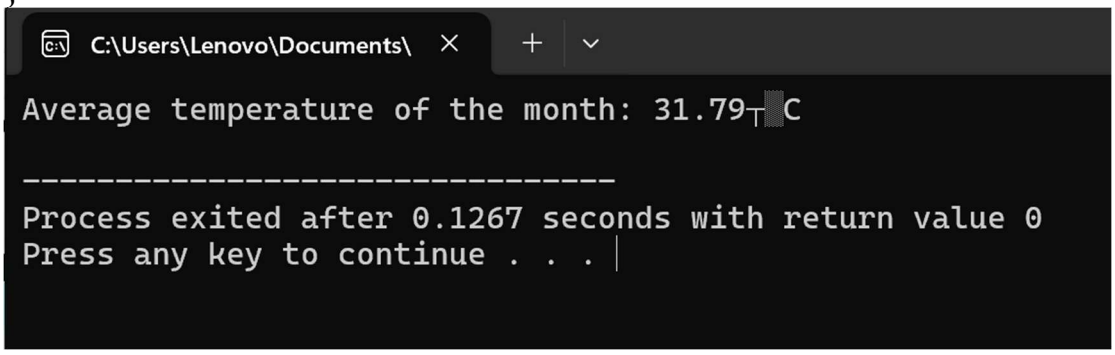
```
#include <iostream>
using namespace std;
int main() {
   int workHours[7] = {8, 8, 8, 8, 8, 6, 0}; // Work hours for 7 days
   int totalHours = 0;
   for (int i = 0; i < 7; i++) {
      totalHours += workHours[i];
   }
   cout << "Total work hours in the week: " << totalHours << " hours" <<
endl;
   return 0;
}
```



Temperature Record for a Month:

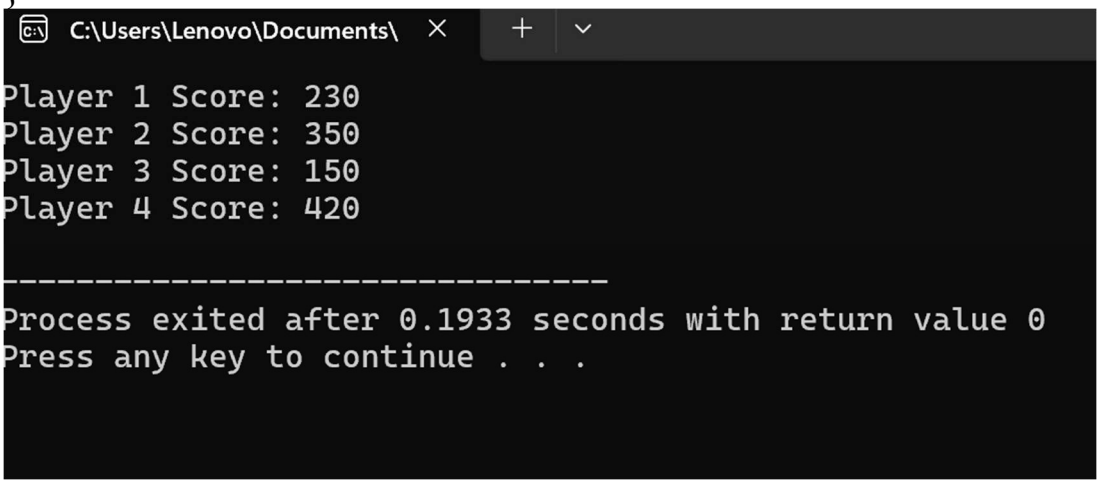
```
#include <iostream>
using namespace std;
int main() {
   float temperatures[30] = {30.5, 32.0, 31.0, 33.2, 34.1, 31.5, 32.6, 33.0,
31.9, 30.0, 31.2, 32.8, 33.5, 34.0, 35.1,
                        36.0, 34.2, 33.8, 32.7, 31.8, 30.1, 32.5, 33.2, 34.3, 35.0,
32.9, 33.1, 34.0, 33.7, 32.8, 30.9};

   float total = 0;
   for (int i = 0; i < 30; i++) {
```

```
        total += temperatures[i];
    }

    float averageTemperature = total / 30;
    cout << "Average temperature of the month: " << averageTemperature
<< "°C" << endl;
    return 0;
}
```



Game Player Scores:

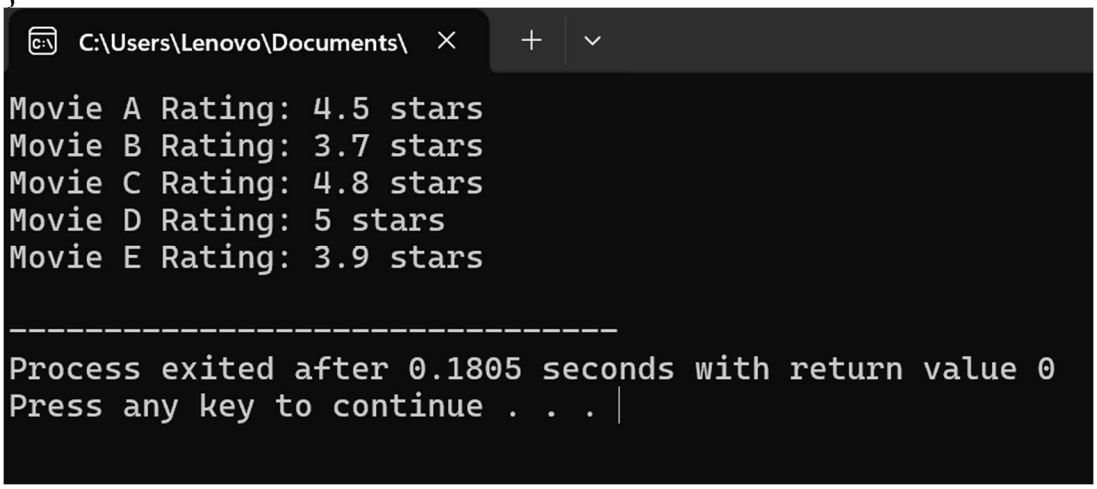
```
#include <iostream>
using namespace std;
int main() {
    int playerScores[4] = {230, 350, 150, 420}; // Scores of 4 players
    for (int i = 0; i < 4; i++) {
        cout << "Player " << i + 1 << " Score: " << playerScores[i] << endl;
    }
    return 0;
}
```



Movie rating:

```
#include <iostream>
using namespace std;
int main() {
```

```
    float movieRatings[5] = {4.5, 3.7, 4.8, 5.0, 3.9}; // Ratings for 5
movies
    string movies[5] = {"Movie A", "Movie B", "Movie C", "Movie D",
"Movie E"};

    for (int i = 0; i < 5; i++) {
        cout << movies[i] << " Rating: " << movieRatings[i] << " stars" <<
endl;
    }
    return 0;
}
```



LAB NO 2

2D Array

Definition:

A **2D array** in C++ is a collection of data elements organized in rows and columns, forming a matrix-like structure. It's essentially an array of arrays, where each element is identified by two indices: one for the row and one for the column.

Syntax:

The general syntax for declaring a 2D array is:

data_type array_name[row_size][column_size];

- **data_type**: The type of data stored in the array (e.g., int, float, char).
- **array_name**: The name of the array.
- **row_size**: The number of rows in the array.

- **column_size**: The number of columns in the array.

Examples:

Temperature Data:

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

int main() {
   float temperatures[3][7] = {{30.5, 32.1, 33.0, 31.8, 30.2, 29.9, 30.0},  // City 1
                               {28.5, 29.0, 28.8, 29.1, 30.5, 31.0, 32.1},  // City 2
                               {22.5, 23.0, 22.9, 23.5, 24.0, 25.0, 26.1}}; // City 3

   for (int i = 0; i < 3; i++) {
      cout << "City " << i + 1 << " temperatures: ";
      for (int j = 0; j < 7; j++) {
         cout << temperatures[i][j] << " ";
      }
      cout << endl;
   }
   return 0;
}
```

```
City 1 temperatures: 30.5 32.1 33 31.8 30.2 29.9 30
City 2 temperatures: 28.5 29 28.8 29.1 30.5 31 32.1
City 3 temperatures: 22.5 23 22.9 23.5 24 25 26.1

--------------------------------
Process exited after 0.1593 seconds with return value 0
Press any key to continue . . .
```

Game Scoreboard:

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

int main() {
   string teams[3] = {"Team A", "Team B", "Team C"};
   int scores[3][2] = {{5, 3}, {7, 2}, {6, 6}}; // Scores for 3 teams in 2 rounds

   for (int i = 0; i < 3; i++) {
      cout << teams[i] << " - Round 1: " << scores[i][0] << " Round 2: " << scores[i][1] << endl;
```

```
    }
    return 0;
}
```

```
Team A - Round 1: 5 Round 2: 3
Team B - Round 1: 7 Round 2: 2
Team C - Round 1: 6 Round 2: 6

--------------------------------
Process exited after 0.1563 seconds with return value 0
Press any key to continue . . .
```

2D Tic tac toe board:#include <iostream>

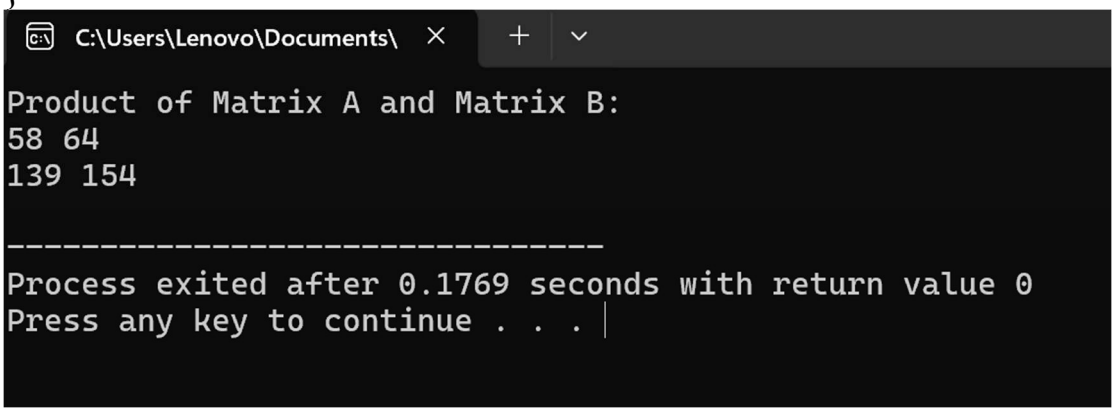
```
using namespace std;

int main() {
    char board[3][3] = {{'X', 'O', 'X'}, {' ', 'X', 'O'}, {'O', ' ', 'X'}};

    cout << "Tic-Tac-Toe Board:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

```
C:\Users\Lenovo\Documents\
Tic-Tac-Toe Board:
X O X
  X O
O   X

--------------------------------
Process exited after 0.1723 seconds with return value 0
Press any key to continue . . .
```

Matrix Multiplication :

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

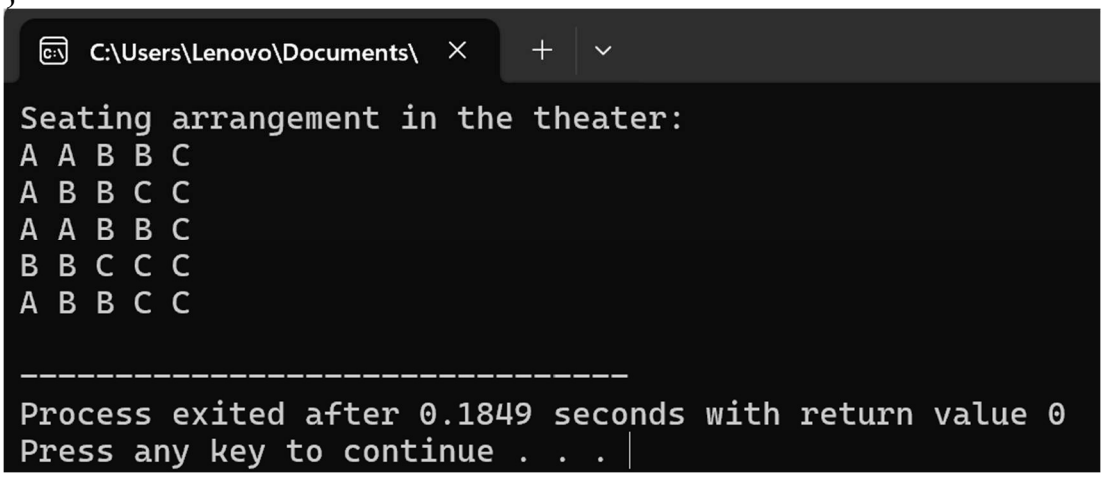
```
int main() {
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int B[3][2] = {{7, 8}, {9, 10}, {11, 12}};
    int result[2][2] = {0};  // Initialize result matrix with 0s

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    cout << "Product of Matrix A and Matrix B:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```



Seating Arrangement in a Movie Theater:

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

int main() {
    char seats[5][5] = {{'A', 'A', 'B', 'B', 'C'},
                        {'A', 'B', 'B', 'C', 'C'},
                        {'A', 'A', 'B', 'B', 'C'},
                        {'B', 'B', 'C', 'C', 'C'},
                        {'A', 'B', 'B', 'C', 'C'}};
```

```
    cout << "Seating arrangement in the theater:" << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            cout << seats[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```



LAB NO 3
Vectors

Definition:
A **vector** in C++ is a dynamic array-like data structure from the C++ Standard Template Library (STL) that can automatically resize when elements are added or removed. Vectors are more flexible than arrays because their size can change at runtime, and they provide various member functions for manipulating the stored data.

Syntax:
To use vectors in C++, you need to include the <vector> header and then define a vector as follows:

```
#include <vector>
vector<data_type> vector_name;
```

Where:

- **data_type** is the type of elements the vector will store (e.g., int, float, string).
- **vector_name** is the name you want to assign to the vector.

You can also initialize a vector with predefined values:

```
vector<data_type> vector_name = {value1, value2, value3};
```

Basic Operations on Vectors:

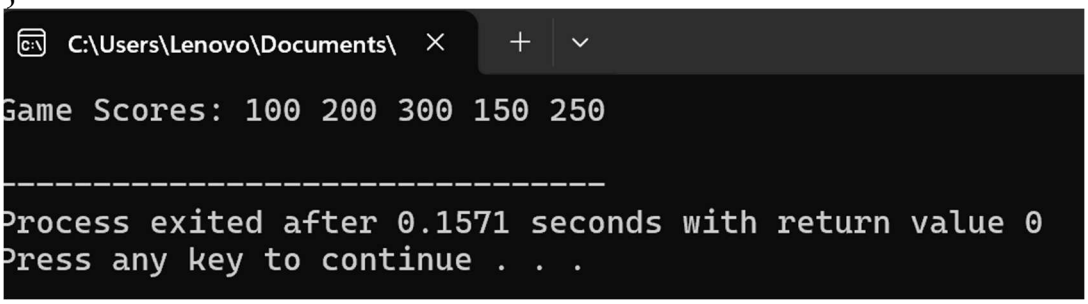
1. **Adding elements**: push_back(value)
2. **Accessing elements**: vector_name[index]
3. **Getting the size**: vector_name.size()
4. **Removing last element**: pop_back()

Examples:

. Dynamic List of Scores in a Game

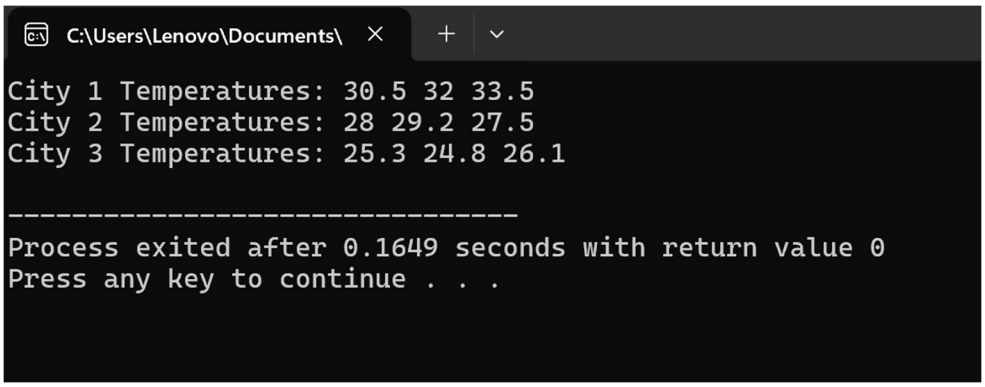
```
#include <iostream>
#include <vector>
using namespace std;
int main() {
   vector<int> scores;

   // Simulating scores for 5 rounds
   scores.push_back(100);
   scores.push_back(200);
   scores.push_back(300);
   scores.push_back(150);
   scores.push_back(250);
```

```
    cout << "Game Scores: ";
    for (const int score : scores) {
        cout << score << " ";
    }
    cout << endl;

    return 0;
}
```



Storing Temperature Data for Multiple Cities

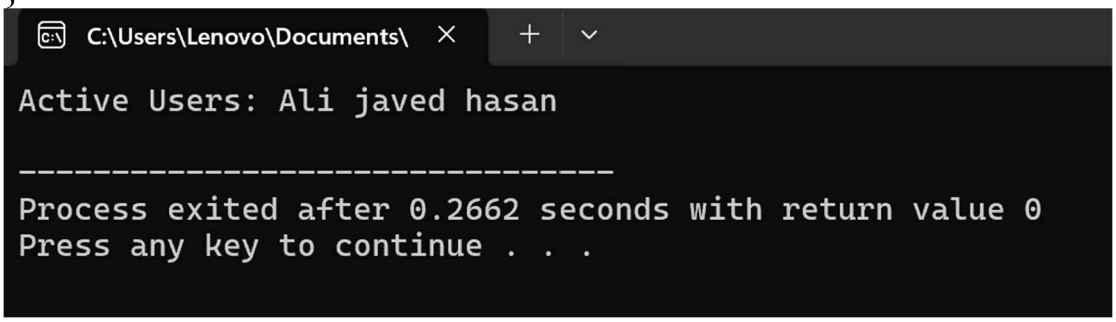
```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<vector<float>> temperatures = {
        {30.5, 32.0, 33.5},  // City 1
        {28.0, 29.2, 27.5},  // City 2
        {25.3, 24.8, 26.1}   // City 3
    };

    for (int i = 0; i < temperatures.size(); i++) {
        cout << "City " << i + 1 << " Temperatures: ";
        for (float temp : temperatures[i]) {
            cout << temp << " ";
        }
        cout << endl;
    }

    return 0;
```



}

Track Users in an Online Application:

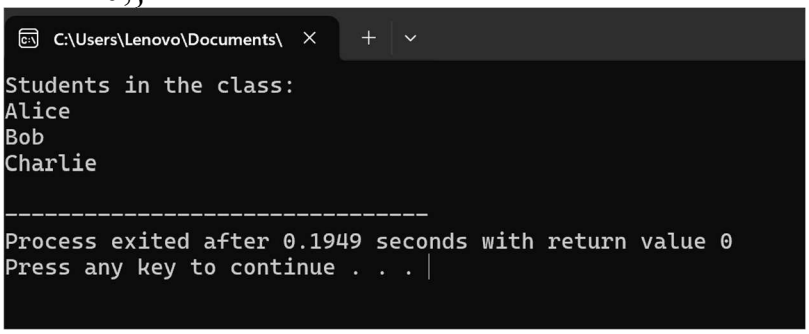
```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<string> activeUsers = {"Alice", "Bob", "Charlie"};

    cout << "Active Users: ";
    for (const string& user : activeUsers) {
        cout << user << " ";
    }
    cout << endl;

    return 0;
}
```



Dynamic List of Students in a Class

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<string> students;
    students.push_back("Alice");
    students.push_back("Bob");
```

```
students.push_back("Charlie");

cout << "Students in the class: " << endl;
for (const auto& student : students) {
    cout << student << endl;
}

return 0;}
```



LAB NO 4

List

Definition:
In C++, a **list** is a doubly linked list container provided by the **Standard Template Library (STL)**. It allows for efficient insertion and deletion of elements at both ends (front and back) or in the middle, unlike arrays or vectors which require shifting elements for insertion and deletion. A list stores elements in non-contiguous memory locations, and each element (node) contains a pointer to the next and previous elements.
Syntax:
To use a list in C++, you need to include the <list> header. The basic syntax for declaring a list is:
cpp
Copy code
#include <list>

list<data_type> list_name;
Where:

- **data_type** is the type of elements the list will store (e.g., int, string).
- **list_name** is the name of the list.

You can also initialize a list with predefined values:
list<data_type> list_name = {value1, value2, value3};

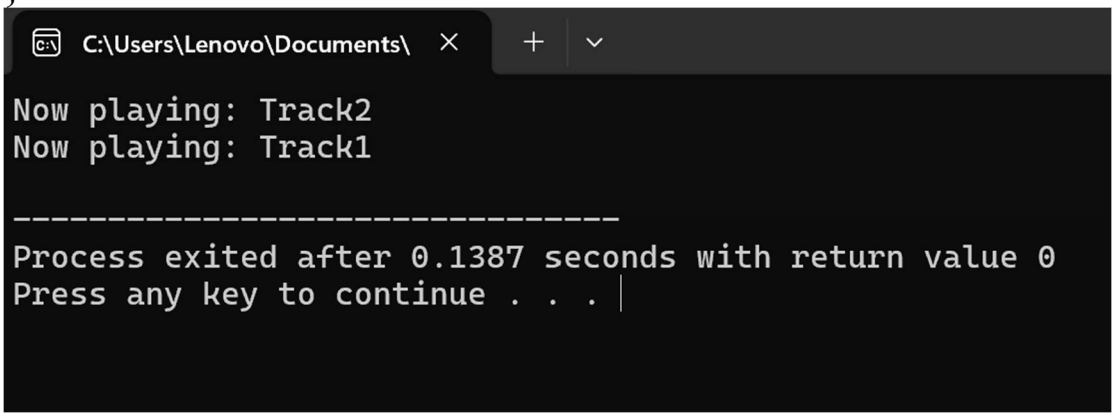
Basic Operations on Lists:

1. **Adding elements**: push_back(value), push_front(value)
2. **Accessing elements**: You must use an iterator to access elements, as lists do not provide direct indexing.
3. **Removing elements**: pop_back(), pop_front(), remove(value)
4. **Iterating through the list**: Use iterators (e.g., begin(), end())

Example:

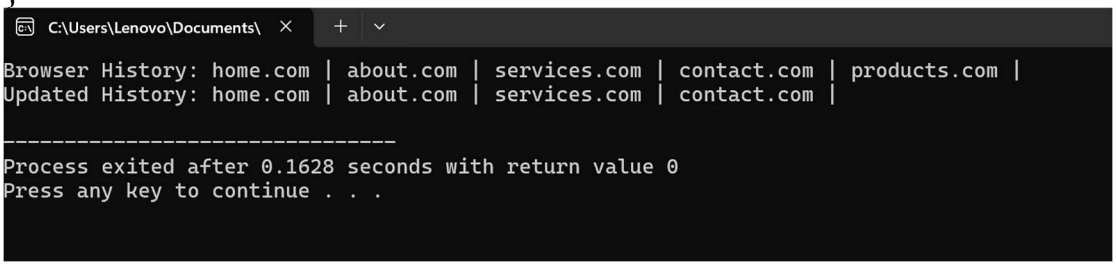
Doubly Linked List for Music Tracks:

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

int main() {
   list<string> tracks = {"Track1", "Track2", "Track3"};

   // Move to the next track
   auto it = tracks.begin();
   advance(it, 1); // Move iterator to the second track
   cout << "Now playing: " << *it << endl;

   // Move to the previous track
   it--;
   cout << "Now playing: " << *it << endl;

   return 0;
}
```



Browser History:

```
#include <iostream>#include <list>using namespace std;
int main() {
    list<string> browserHistory = {"home.com", "about.com",
"services.com"};

    // Add new pages to history
    browserHistory.push_back("contact.com");
    browserHistory.push_back("products.com");

    cout << "Browser History: ";
    for (const string& page : browserHistory) {
        cout << page << " | ";
    }
    cout << endl;

    // Remove the last visited page (user presses back)
    browserHistory.pop_back();

    cout << "Updated History: ";
    for (const string& page : browserHistory) {
        cout << page << " | ";
    }
    cout << endl;

    return 0;
}
```



Real-time Log of Sensor Data

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> sensorData = {10, 15, 20, 25};

    // New sensor data comes in
    sensorData.push_back(30);
```

```
    sensorData.push_back(35);

    cout << "Sensor Data: ";
    for (const int& data : sensorData) {
        cout << data << " ";
    }
    cout << endl;

    return 0;
}
```

```
C:\Users\Lenovo\Documents\
Sensor Data: 10 15 20 25 30 35

--------------------------------
Process exited after 0.1473 seconds with return value 0
Press any key to continue . . .
```

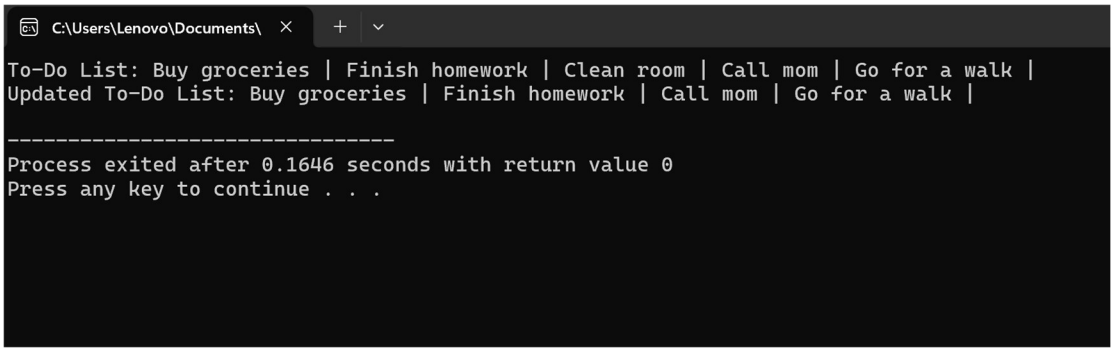
Task Prioritization List

```
#include <iostream>#include <list>using namespace std;
int main() {
    list<string> tasks = {"Finish report", "Email client", "Clean desk"};

    // Add a high-priority task at the front
    tasks.push_front("Call manager");

    // Display tasks based on priority
    cout << "Task List: ";
    for (const string& task : tasks) {
        cout << task << " | ";
    }
    cout << endl;

   return 0;
   }
```

```
C:\Users\Lenovo\Documents\
Task List: Call manager | Finish report | Email client | Clean desk |

--------------------------------
Process exited after 0.2327 seconds with return value 0
Press any key to continue . . .
```

Task List / To-Do List

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<string> todoList = {"Buy groceries", "Finish homework", "Clean
room"};

    todoList.push_back("Call mom");  // Add new task
    todoList.push_back("Go for a walk");

    cout << "To-Do List: ";
    for (const string& task : todoList) {
        cout << task << " | ";
    }
    cout << endl;

    todoList.remove("Clean room");  // Remove completed task

    cout << "Updated To-Do List: ";
    for (const string& task : todoList) {
        cout << task << " | ";
    }
    cout << endl;

    return 0;
}
```



LAB NO 5
Stack

Definition:

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. It is commonly used in various algorithms and is particularly useful for problems related to recursive calls, parsing expressions, and backtracking.

In C++, the **stack** is part of the **Standard Template Library (STL)** and is defined in the <stack> header. The stack allows operations like:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element of the stack.
- **Top**: Accesses the top element without removing it.
- **Empty**: Checks if the stack is empty.
- **Size**: Returns the number of elements in the stack.

Syntax:

To use a stack in C++, include the <stack> header. The basic syntax for declaring a stack is:

```
#include <stack>

stack<data_type> stack_name;
```

Where:

- **data_type** is the type of elements the stack will store (e.g., int, string).
- **stack_name** is the name of the stack.

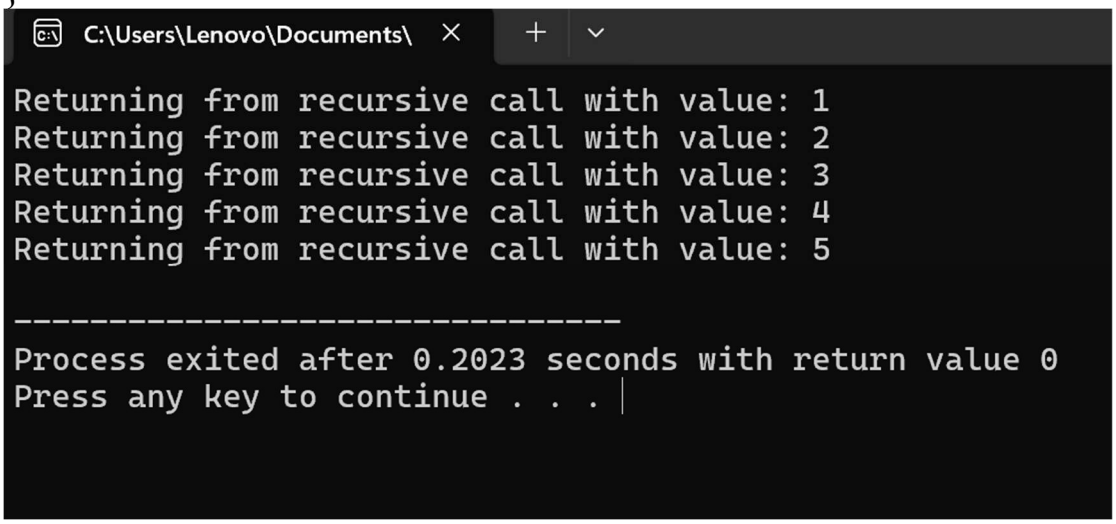
Basic Operations on Stacks:

1. **Push**: stack_name.push(value)
2. **Pop**: stack_name.pop()
3. **Top**: stack_name.top()
4. **Check if empty**: stack_name.empty()
5. **Size**: stack_name.size()

Recursive Function Call Simulation

```
#include <iostream>
#include <stack>
using namespace std;
void simulateRecursiveCall(int n) {
   stack<int> s;

   while (n > 0) {
      s.push(n);
      n--;
   }
```

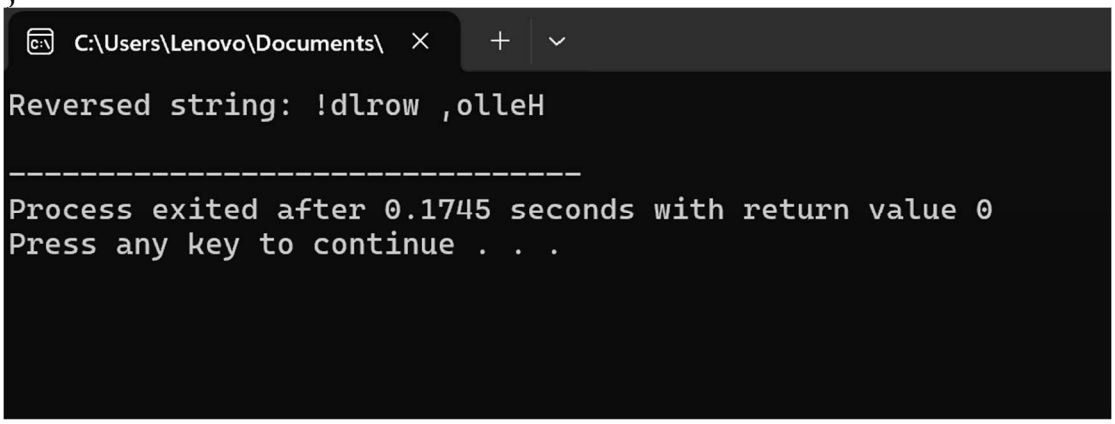
```
    while (!s.empty()) {
        cout << "Returning from recursive call with value: " << s.top() <<
endl;
        s.pop();
    }
}
int main() {
    simulateRecursiveCall(5);

    return 0;
}
```



Reversing a String

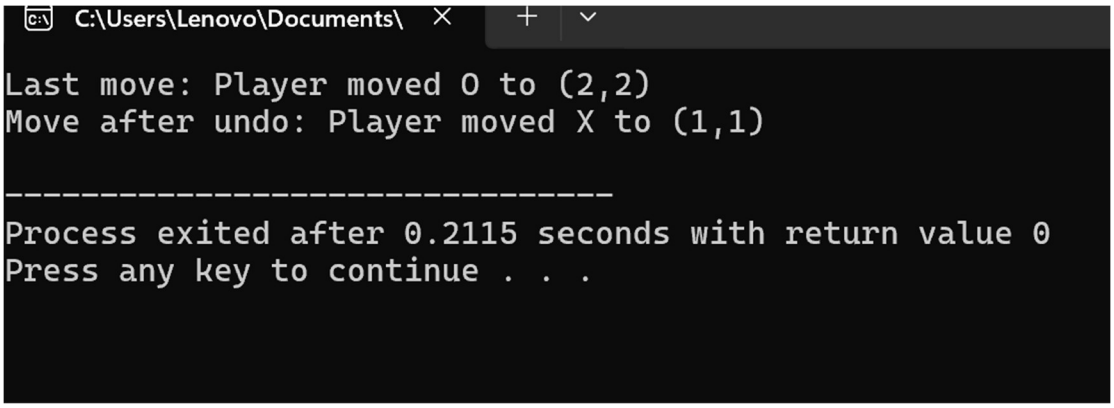
```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
string reverseString(string str) {
    stack<char> s;
    for (char c : str) {
        s.push(c);
    }

    string reversed;
    while (!s.empty()) {
        reversed += s.top();
        s.pop();
    }

    return reversed;
```

```
}
int main() {
    string str = "Hello, world!";
    cout << "Reversed string: " << reverseString(str) << endl;

    return 0;
}
```



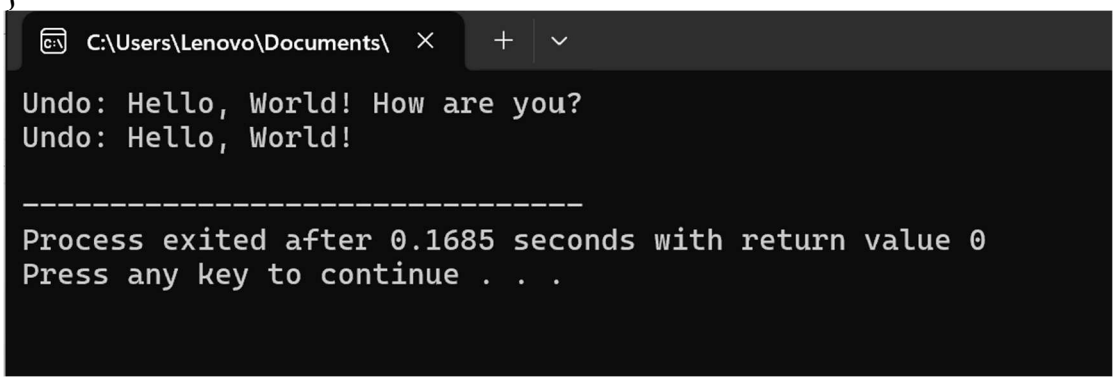
Game Move History

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main() {
    stack<string> moveHistory;

    // Player makes moves
    moveHistory.push("Player moved X to (1,1)");
    moveHistory.push("Player moved O to (2,2)");

    cout << "Last move: " << moveHistory.top() << endl;

    // Undo last move
    moveHistory.pop();

    cout << "Move after undo: " << moveHistory.top() << endl;

    return 0;
}
```



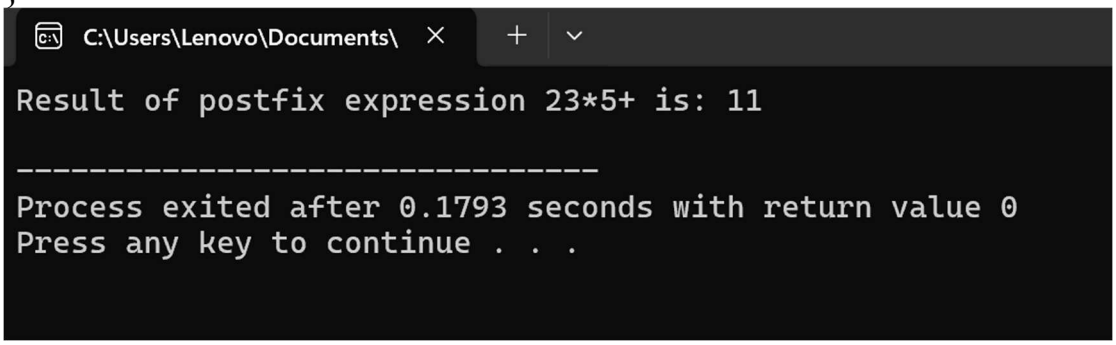
Undo Functionality in Text Editors

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int main() {
    stack<string> undoStack;

    // Simulate text editing
    undoStack.push("Hello");
    undoStack.push("Hello, World!");
    undoStack.push("Hello, World! How are you?");

    // Simulate undo operation
    cout << "Undo: " << undoStack.top() << endl;
    undoStack.pop();  // Undo last change

    cout << "Undo: " << undoStack.top() << endl;
    undoStack.pop();  // Undo again

    return 0;
}
```



Expression Evaluation (Postfix)

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
int evaluatePostfix(string expr) {
    stack<int> s;

    for (char c : expr) {
        if (isdigit(c)) {
            s.push(c - '0');
        } else {
            int b = s.top(); s.pop();
            int a = s.top(); s.pop();
            if (c == '+') s.push(a + b);
            else if (c == '-') s.push(a - b);
            else if (c == '*') s.push(a * b);
            else if (c == '/') s.push(a / b);
        }
    }

    return s.top();
}
int main() {
    string expr = "23*5+";
    cout << "Result of postfix expression " << expr << " is: " <<
evaluatePostfix(expr) << endl;
    return 0;
}
```



LAB NO 6

Queue

Definition:

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. In a queue, the element that is inserted first is the one that gets removed first. This is like a queue at a movie theater where the first person in line is the first one to get a ticket. It is used in scenarios like task scheduling, handling requests in servers, and simulating real-life queues.

In C++, the **queue** is part of the **Standard Template Library (STL)** and is defined in the <queue> header. A queue supports several operations:

- **Enqueue (push)**: Adds an element to the back of the queue.
- **Dequeue (pop)**: Removes an element from the front of the queue.
- **Front**: Retrieves the element at the front without removing it.
- **Back**: Retrieves the element at the back without removing it.
- **Empty**: Checks if the queue is empty.
- **Size**: Returns the number of elements in the queue.

Syntax:

cpp
Copy code

```
#include <queue>

queue<data_type> queue_name;
```

Where:

- **data_type** is the type of elements the queue will store (e.g., int, string).
- **queue_name** is the name of the queue.

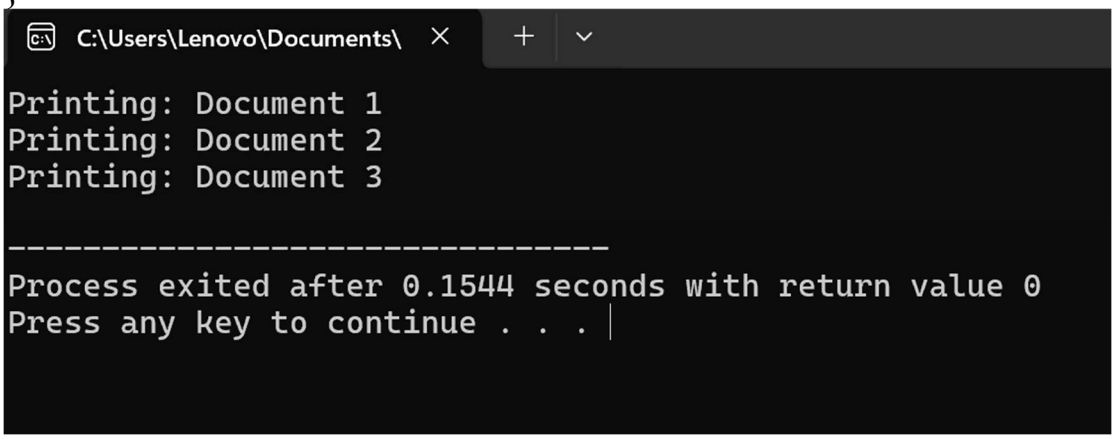
Basic Operations on Queues:

1. **Enqueue**: queue_name.push(value)
2. **Dequeue**: queue_name.pop()
3. **Front**: queue_name.front()
4. **Back**: queue_name.back()
5. **Check if empty**: queue_name.empty()
6. **Size**: queue_name.size()

Print Job Scheduling

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
```

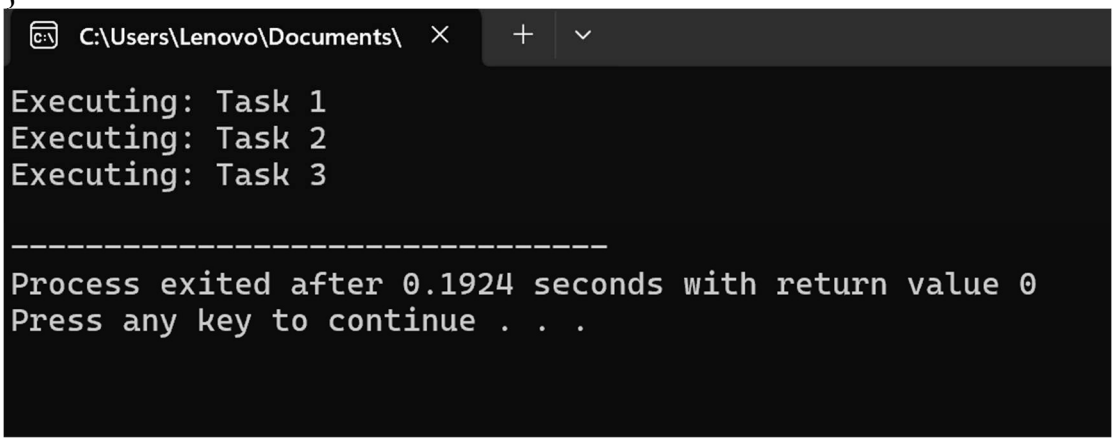
```
    queue<string> printQueue;

    // Add print jobs
    printQueue.push("Document 1");
    printQueue.push("Document 2");
    printQueue.push("Document 3");

    // Print jobs in order
    while (!printQueue.empty()) {
        cout << "Printing: " << printQueue.front() << endl;
        printQueue.pop();
    }

    return 0;
}
```



Task Scheduling (Operating System)

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
    queue<string> taskQueue;

    // Add tasks to the queue
    taskQueue.push("Task 1");
    taskQueue.push("Task 2");
    taskQueue.push("Task 3");

    // Execute tasks in order
    while (!taskQueue.empty()) {
        cout << "Executing: " << taskQueue.front() << endl;
        taskQueue.pop();
```

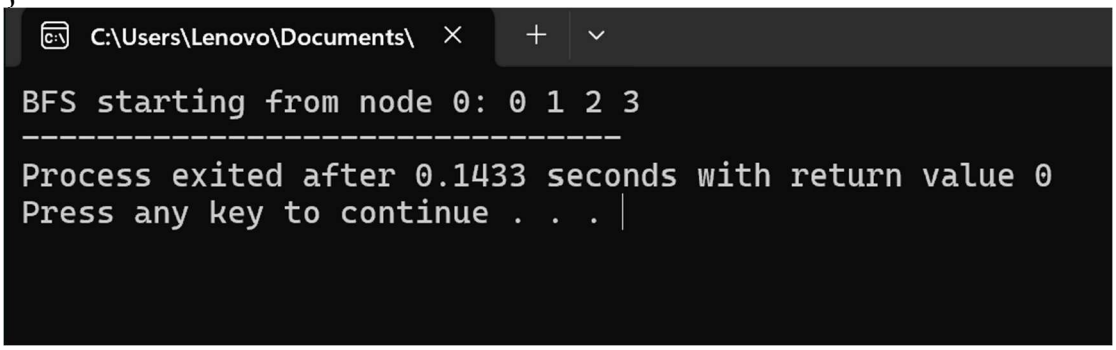
```
    }

    return 0;
}
```



Breadth-First Search (BFS) in Graphs

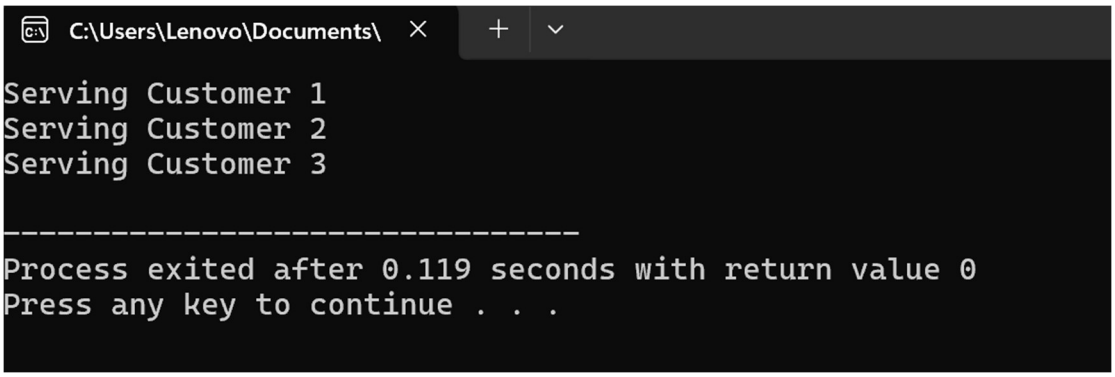
```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
void BFS(int start, vector<vector<int>>& adjList) {
    queue<int> q;
    vector<bool> visited(adjList.size(), false);

    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();

        cout << node << " ";

        for (int neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}
int main() {
    vector<vector<int>> adjList = {
```

```
        {1, 2},    // Node 0
        {0, 3},    // Node 1
        {0, 3},    // Node 2
        {1, 2}     // Node 3
    };

    cout << "BFS starting from node 0: ";
    BFS(0, adjList);

    return 0;
}
```



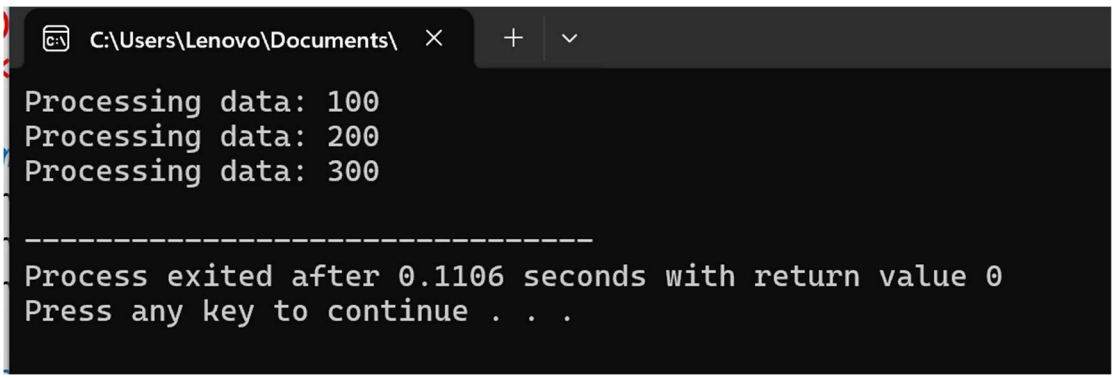
Waiting Line at a Supermarket

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<string> checkoutQueue;

    // Customers entering the queue
    checkoutQueue.push("Customer 1");
    checkoutQueue.push("Customer 2");
    checkoutQueue.push("Customer 3");

    // Serve customers at the checkout
    while (!checkoutQueue.empty()) {
        cout << "Serving " << checkoutQueue.front() << endl;
        checkoutQueue.pop();
    }

    return 0;
}
```



Real-Time Data Streaming

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<int> sensorData;

    // Simulate sensor data streaming
    sensorData.push(100);
    sensorData.push(200);
    sensorData.push(300);

    // Process data
    while (!sensorData.empty()) {
        cout << "Processing data: " << sensorData.front() << endl;
        sensorData.pop();
    }

    return 0;
}
```



LAB NO 7
DEQUEUE

Definition:

A **deque** (short for **double-ended queue**) is a linear data structure that allows elements to be inserted or removed from both ends (front and back). Unlike a regular queue, which follows the **First In, First Out (FIFO)** principle, a deque is more versatile because it supports insertion and removal at both ends. This makes it useful in scenarios where you need to add or remove elements from both ends efficiently.

In C++, the **deque** is part of the **StandaRd Template Library (STL)** and is defined in the <deque> header. It supports operations like:

- **Push_front**: Adds an element to the front of the deque.
- **Push_back**: Adds an element to the back of the deque.
- **Pop_front**: Removes an element from the front of the deque.
- **Pop_back**: Removes an element from the back of the deque.
- **Front**: Accesses the element at the front without removing it.
- **Back**: Accesses the element at the back without removing it.
- **Size**: Returns the number of elements in the deque.
- **Empty**: Checks if the deque is empty.

Syntax

```
#include <deque>

deque<data_type> deque_name;
```

Where:

- **data_type** is the type of elements the deque will store (e.g., int, string).
- **deque_name** is the name of the deque.

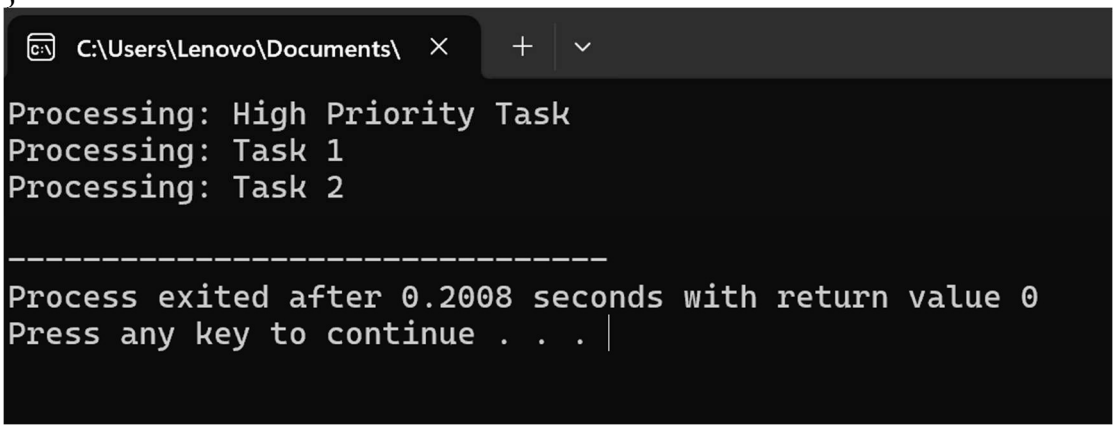
Basic Operations on Deques:

1. **Push_front**: deque_name.push_front(value)
2. **Push_back**: deque_name.push_back(value)
3. **Pop_front**: deque_name.pop_front()
4. **Pop_back**: deque_name.pop_back()
5. **Front**: deque_name.front()
6. **Back**: deque_name.back()
7. **Check if empty**: deque_name.empty()
8. **Size**: deque_name.size()

Two-Ended Queue for Task Scheduling

```
#include <iostream>
#include <deque>
#include <string>
using namespace std;
```

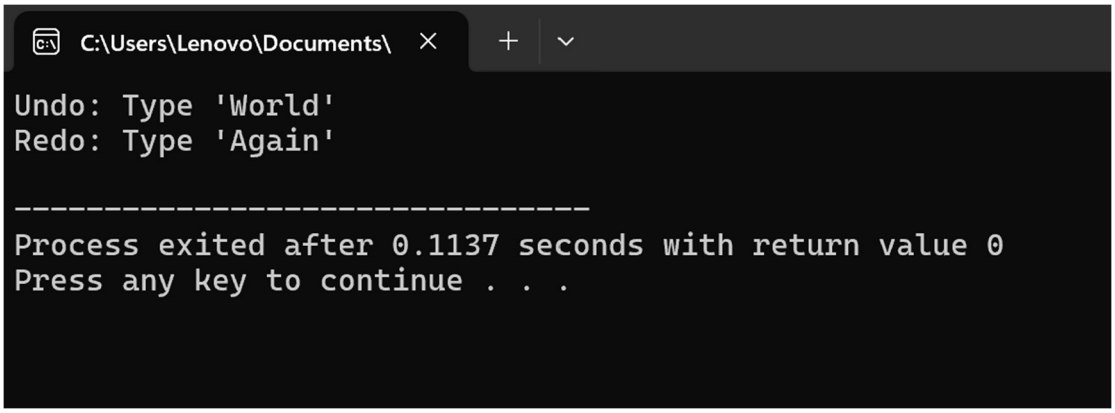
```
int main() {
    deque<string> taskQueue;

    // Add tasks to the back of the queue
    taskQueue.push_back("Task 1");
    taskQueue.push_back("Task 2");

    // Add high-priority task to the front of the queue
    taskQueue.push_front("High Priority Task");

    // Process tasks
    while (!taskQueue.empty()) {
        cout << "Processing: " << taskQueue.front() << endl;
        taskQueue.pop_front();
    }

    return 0;
}
```



Undo/Redo Operations in a Text Editor

```
#include <iostream>
#include <deque>
#include <string>
using namespace std;
int main() {
    deque<string> actions;

    // Simulate actions
    actions.push_back("Type 'Hello'");
    actions.push_back("Type 'World'");

    // Undo last action
    cout << "Undo: " << actions.back() << endl;
```

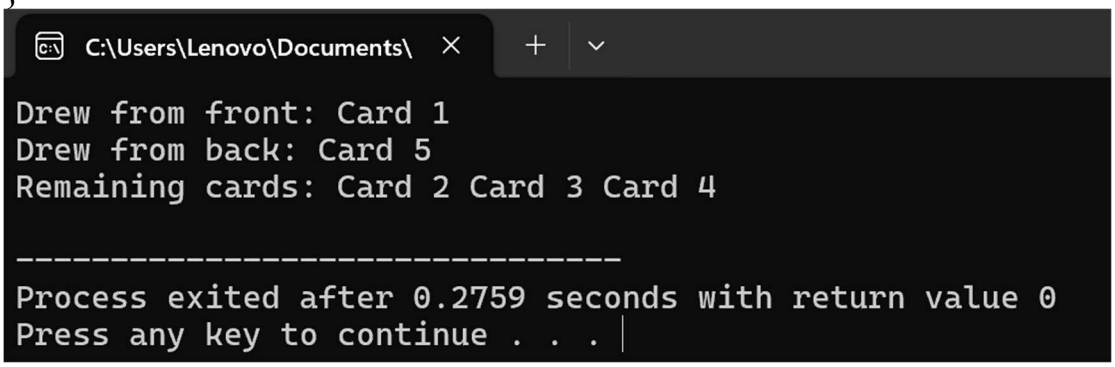
```
    actions.pop_back();

    // Redo action
    actions.push_back("Type 'Again'");
    cout << "Redo: " << actions.back() << endl;

    return 0;
}
```



Deck of Cards Simulation

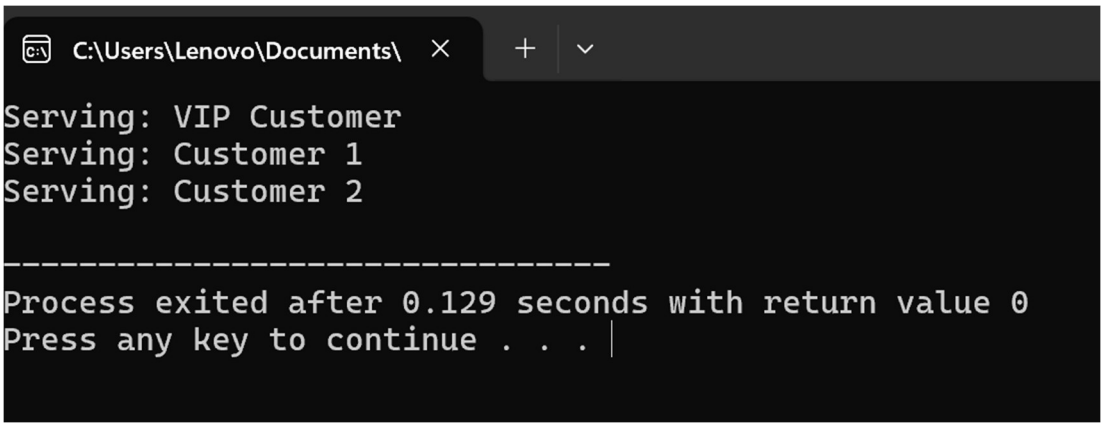
```
#include <iostream>
#include <deque>
#include <string>
using namespace std;
int main() {
    deque<string> deck = {"Card 1", "Card 2", "Card 3", "Card 4", "Card
5"};

    // Draw card from the front
    cout << "Drew from front: " << deck.front() << endl;
    deck.pop_front();

    // Draw card from the back
    cout << "Drew from back: " << deck.back() << endl;
    deck.pop_back();

    // Display remaining cards
    cout << "Remaining cards: ";
    for (const string& card : deck) {
        cout << card << " ";
    }
    cout << endl;
```

```
    return 0;
}
```



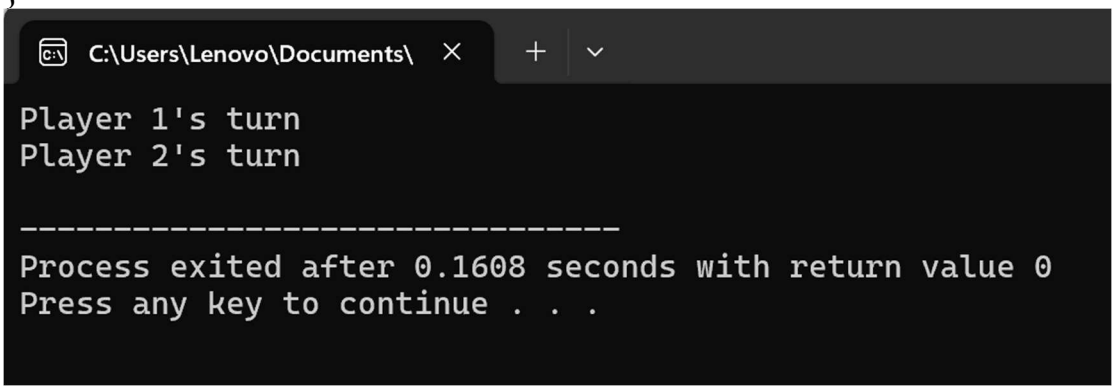
Movie Queue Simulation (First Come, First Served and Priority)

```
#include <iostream>
#include <deque>
#include <string>
using namespace std;
int main() {
    deque<string> movieQueue;

    // Normal customer queue
    movieQueue.push_back("Customer 1");
    movieQueue.push_back("Customer 2");

    // Priority customer at the front
    movieQueue.push_front("VIP Customer");

    // Serve customers
    while (!movieQueue.empty()) {
        cout << "Serving: " << movieQueue.front() << endl;
        movieQueue.pop_front();
    }

    return 0;
}
```



Simulation of Player's Turn in a Game

```
#include <iostream>
#include <deque>
#include <string>
using namespace std;
int main() {
    deque<string> players = {"Player 1", "Player 2", "Player 3"};

    // Players take turns
    cout << players.front() << "'s turn" << endl;
    players.push_back(players.front());
    players.pop_front();

    cout << players.front() << "'s turn" << endl;
    players.push_back(players.front());
    players.pop_front();

    return 0;
}
```



LAB NO 8
Tree

Definition:

A **tree** is a hierarchical data structure where each node is connected by edges. It consists of a **root node**, child nodes, and leaf nodes (nodes without children). Trees are used to represent hierarchical relationships such as file directories, organizational structures, or expression parsing.

Syntax for a Tree in C++

```
#include <iostream>#include <vector>using namespace std;
class TreeNode {public:
    int value;                  // Value of the node
    vector<TreeNode*> children; // Vector to store child nodes

    // Constructor
    TreeNode(int val) {
        value = val;
    }

    // Add a child to the current node
    void addChild(TreeNode* child) {
        children.push_back(child);
    }
};
```

Examples of Trees in C++

1. General Tree

A tree where each node can have multiple children.

```
int main() {
    TreeNode* root = new TreeNode(1);
    TreeNode* child1 = new TreeNode(2);
    TreeNode* child2 = new TreeNode(3);

    root->addChild(child1);
    root->addChild(child2);

    cout << "Root: " << root->value << endl;
    for (TreeNode* child : root->children) {
        cout << "Child: " << child->value << endl;
    }

    return 0;
```

```
}
```

2. Binary Tree

A tree where each node has at most two children.

```
class BinaryTreeNode {
public:
    int value;
    BinaryTreeNode* left;
    BinaryTreeNode* right;

    // Constructor
    BinaryTreeNode(int val) {
        value = val;
        left = right = nullptr;
    }
};

int main() {
    BinaryTreeNode* root = new BinaryTreeNode(10);
    root->left = new BinaryTreeNode(5);
    root->right = new BinaryTreeNode(15);

    cout << "Root: " << root->value << endl;
    cout << "Left Child: " << root->left->value << endl;
    cout << "Right Child: " << root->right->value << endl;

    return 0;
}
```

3. Binary Search Tree (BST)

A binary tree with specific properties: left child values are less than the parent, and right child values are greater.

```
class BST {
public:
    int value;
    BST* left;
    BST* right;

    BST(int val) {
        value = val;
        left = right = nullptr;
    }
};
```

```
void insert(BST*& root, int val) {
    if (root == nullptr) {
        root = new BST(val);
        return;
    }
    if (val < root->value) {
        insert(root->left, val);
    } else {
        insert(root->right, val);
    }
}

int main() {
    BST* root = nullptr;
    insert(root, 10);
    insert(root, 5);
    insert(root, 15);

    cout << "Root: " << root->value << endl;
    cout << "Left Child: " << root->left->value << endl;
    cout << "Right Child: " << root->right->value << endl;

    return 0;
}
```

4. Expression Tree

A tree used to evaluate mathematical expressions.

```
class ExprTreeNode {
public:
    char value;
    ExprTreeNode* left;
    ExprTreeNode* right;

    ExprTreeNode(char val) {
        value = val;
        left = right = nullptr;
    }
};

int main() {
    ExprTreeNode* root = new ExprTreeNode('+');
    root->left = new ExprTreeNode('3');
```

```
    root->right = new ExprTreeNode('5');

    cout << "Expression: " << root->left->value << " " << root->value <<
" " << root->right->value << endl;

    return 0;
}
```

5. Trie (Prefix Tree)

A specialized tree used for storing strings or prefixes.

```
class TrieNode {
public:
    char value;
    vector<TrieNode*> children;
    bool isEndOfWord;

    TrieNode(char val) {
        value = val;
        isEndOfWord = false;
    }
};

int main() {
    TrieNode* root = new TrieNode('/'); // Root node

    // Example of adding child nodes
    TrieNode* a = new TrieNode('a');
    TrieNode* b = new TrieNode('b');
    root->children.push_back(a);
    root->children.push_back(b);

    cout << "Root: " << root->value << endl;
    for (TrieNode* child : root->children) {
        cout << "Child: " << child->value << endl;
    }

    return 0;
}
```