THE UNIVERSITY OF AZAD JAMMU AND KASHMIR, MUZAFFARABAD

Department of Software Engineering

Submitted to:	Engr. Sidra Rafique
Course Title:	Data Structure and Algorithm
Course Code:	CS-2101
Session:	2024-2028
Lab No:	05
Roll No:	2024-SE-15
Submitted By:	Shahzad Ahmed Awan
Submission date:	November 29, 2025

Table of Contents

Table of Figures

Lab 05 – Circular Linked List Implementation

1. Objective

To implement Singly Circular Linked List (SCLL) and Doubly Circular Linked List (DCLL) in C++ to perform insertion, deletion, and traversal operations, demonstrating circular linking and bidirectional traversal.

2. Background

- **Singly Circular Linked List:**
  - Each node points to the next node.
  - The last node points back to the head, forming a circular structure.

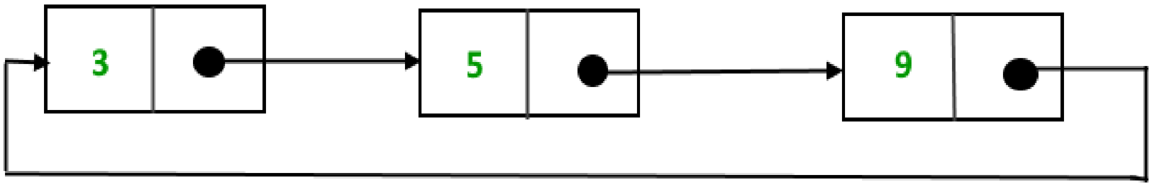


Figure 1: A Diagram Demonstrating Singly Circular Linked List

- **Doubly Circular Linked List:**
  - Each node contains `next` and `prev` pointers.
  - Tail node points to head, and head's `prev` points to tail.
  - Enables bidirectional traversal.

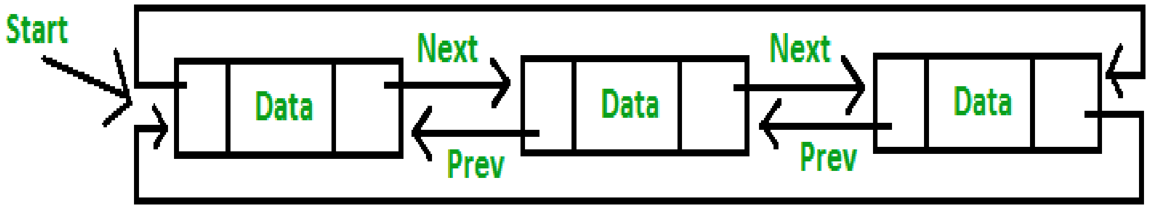


Figure 2: A Diagram Demonstrating Doubly Circular Linked List

Advantages:

- Efficient insertion/deletion at both ends.
- Traversal possible in both directions.

Disadvantages:

- Requires extra memory for `prev` pointer.
- Slightly more complex than singly linked list.

---

3. Algorithm

3.1 Singly Circular Linked List

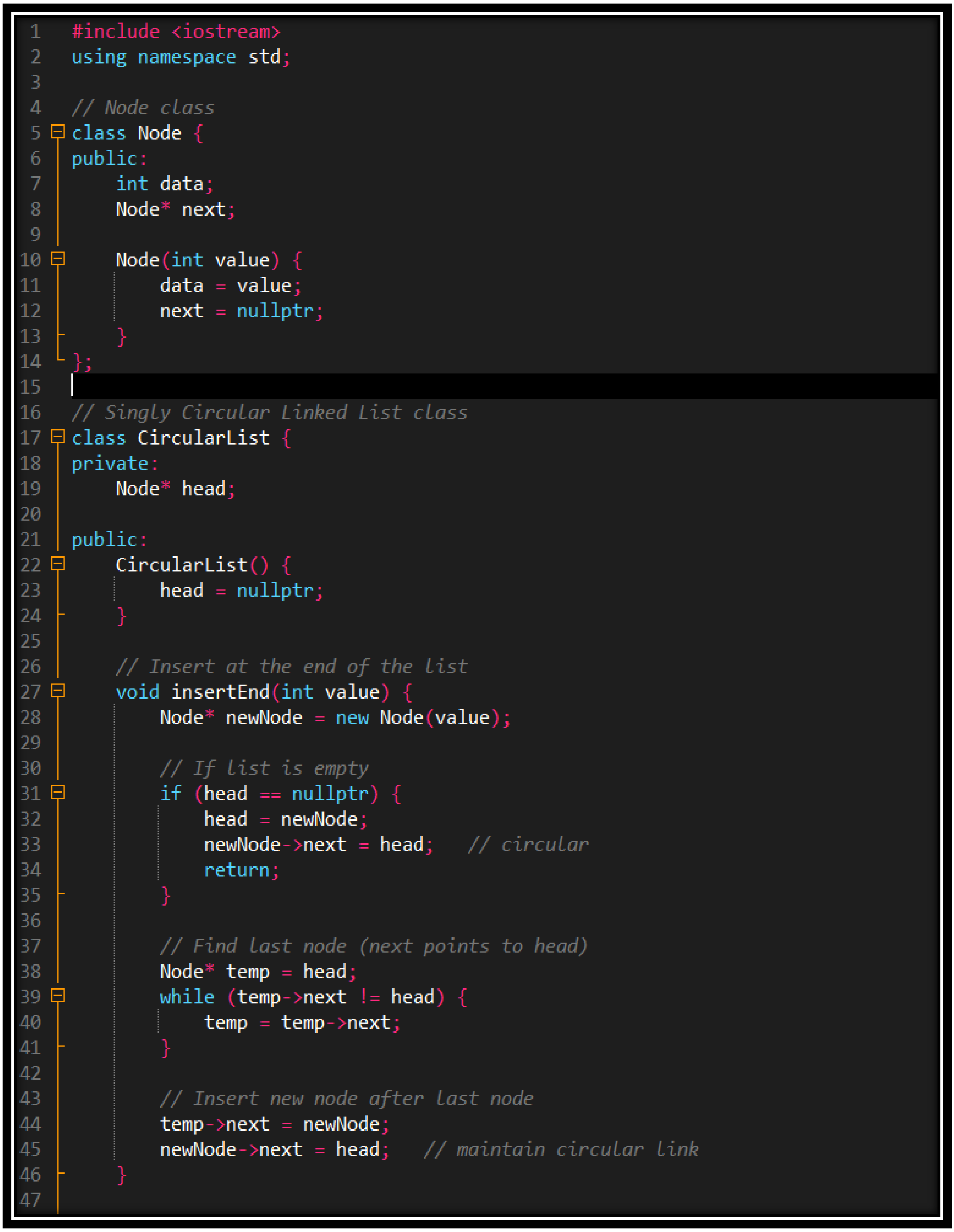
1. Start the program.
2. Define a Node structure with `data` and `next` pointer.
3. Implement functions:
   - `insertEnd()` – Insert node at the end.
   - `display()` – Print circular list starting from head.
4. Insert sample nodes and display the list.
5. End program.

3.2 Doubly Circular Linked List

1. Start the program.
2. Define Node structure with `data`, `prev`, and `next`.
3. Create `DoublyCircularList` class with `head` and `tail`.
4. Implement functions:
   - `insertAtBeginning()` – Insert node at start.
   - `insertAtEnd()` – Insert node at end.
   - `deleteFromBeginning()` – Remove first node.
   - `deleteFromEnd()` – Remove last node.
   - `displayForward()` – Traverse head → tail.
   - `displayBackward()` – Traverse tail → head.
5. Use a menu loop for repeated operations.
6. End program.

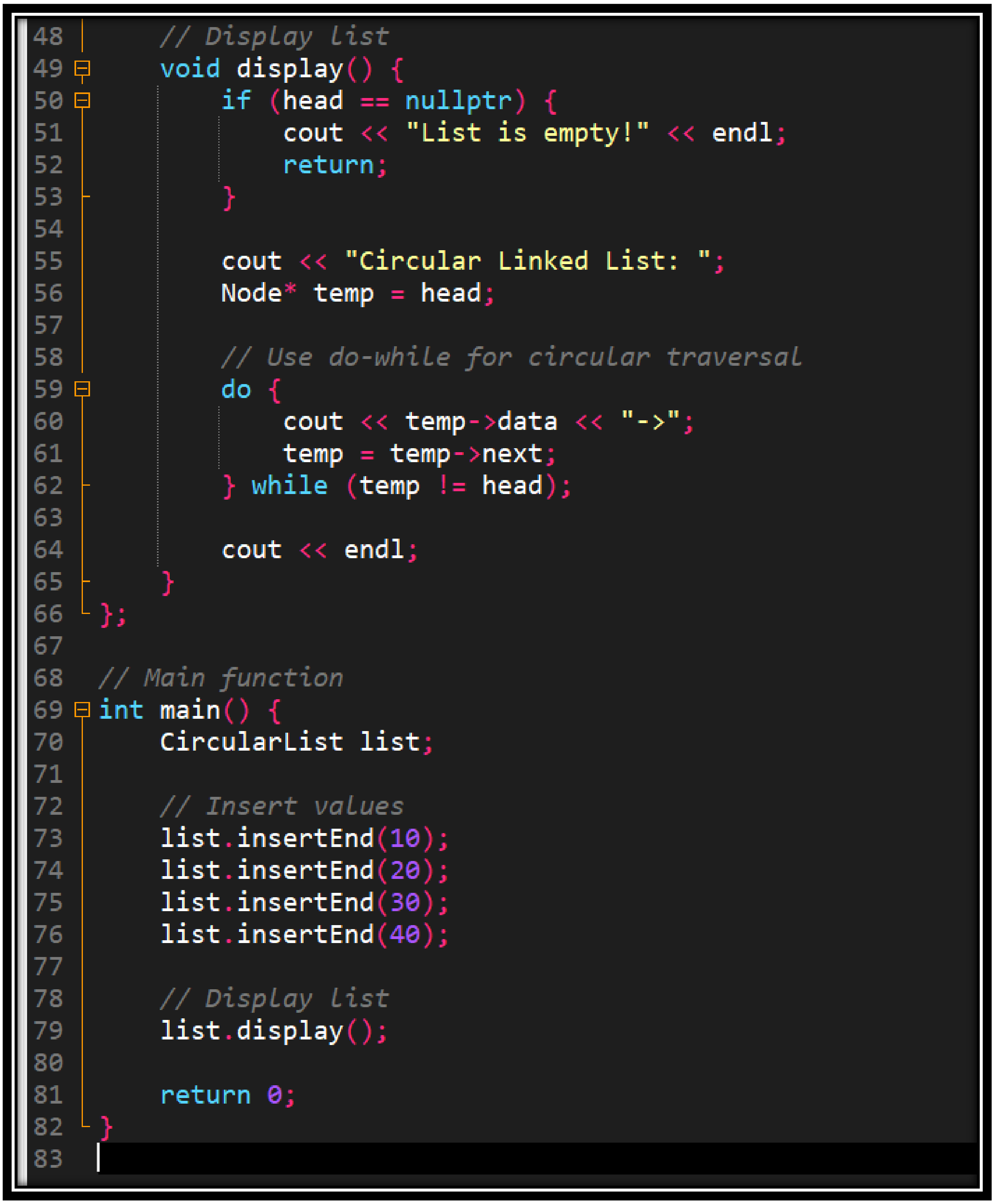
4. Source Code Explanation

4.1 Singly Circular Linked List



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

// Node class
class Node {
public:
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

// Singly Circular Linked List class
class CircularList {
private:
    Node* head;

public:
    CircularList() {
        head = nullptr;
    }

    // Insert at the end of the list
    void insertEnd(int value) {
        Node* newNode = new Node(value);

        // If list is empty
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;    // circular
            return;
        }

        // Find last node (next points to head)
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }

        // Insert new node after last node
        temp->next = newNode;
        newNode->next = head;    // maintain circular link
    }

```

Figure 3: Source Code for Singly Circular Linked List using OOP (Screenshot -1)

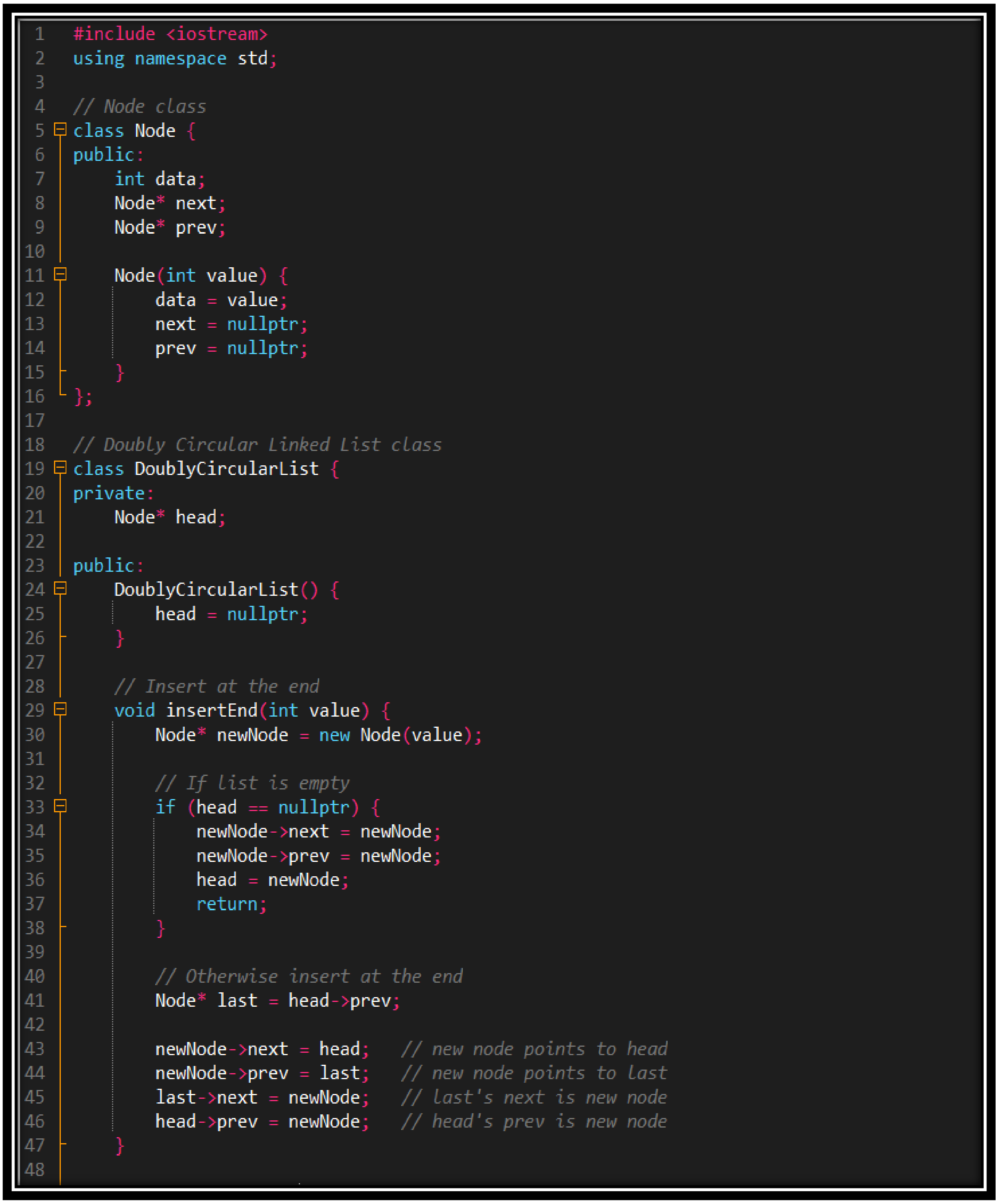


```cpp
    // Display list
    void display() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }

        cout << "Circular Linked List: ";
        Node* temp = head;

        // Use do-while for circular traversal
        do {
            cout << temp->data << "->";
            temp = temp->next;
        } while (temp != head);

        cout << endl;
    }
};

// Main function
int main() {
    CircularList list;

    // Insert values
    list.insertEnd(10);
    list.insertEnd(20);
    list.insertEnd(30);
    list.insertEnd(40);

    // Display list
    list.display();

    return 0;
}
```

Figure 4: Source Code for Singly Circular Linked list using OOP (Screenshot -2)

- **`insertEnd()`**: Adds a node after last node; points it back to head.
- **`display()`**: Traverses from head until it returns to head.

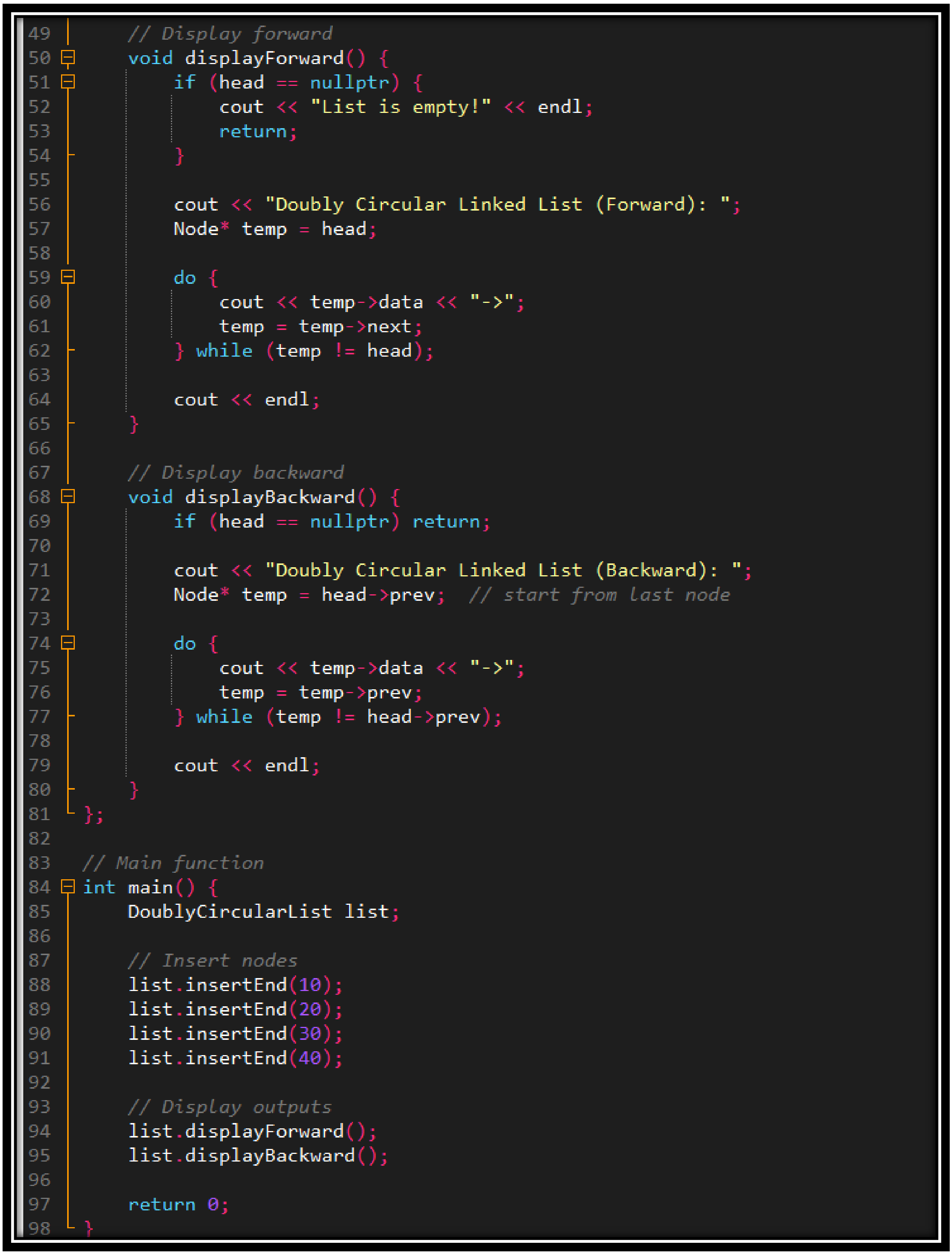
4.2 Doubly Circular Linked List



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

// Node class
class Node {
public:
    int data;
    Node* next;
    Node* prev;

    Node(int value) {
        data = value;
        next = nullptr;
        prev = nullptr;
    }
};

// Doubly Circular Linked List class
class DoublyCircularList {
private:
    Node* head;

public:
    DoublyCircularList() {
        head = nullptr;
    }

    // Insert at the end
    void insertEnd(int value) {
        Node* newNode = new Node(value);

        // If list is empty
        if (head == nullptr) {
            newNode->next = newNode;
            newNode->prev = newNode;
            head = newNode;
            return;
        }

        // Otherwise insert at the end
        Node* last = head->prev;

        newNode->next = head;   // new node points to head
        newNode->prev = last;   // new node points to last
        last->next = newNode;   // last's next is new node
        head->prev = newNode;   // head's prev is new node
    }

```

Figure 5: Source Code for Doubly Circular Linked List using OOP (Screenshot- 1)

- **insertEnd()** – Adds a new node at the end by linking it between the last node and the head, keeping both `next` and `prev` circular.



```cpp
    // Display forward
    void displayForward() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }

        cout << "Doubly Circular Linked List (Forward): ";
        Node* temp = head;

        do {
            cout << temp->data << "->";
            temp = temp->next;
        } while (temp != head);

        cout << endl;
    }

    // Display backward
    void displayBackward() {
        if (head == nullptr) return;

        cout << "Doubly Circular Linked List (Backward): ";
        Node* temp = head->prev;  // start from last node

        do {
            cout << temp->data << "->";
            temp = temp->prev;
        } while (temp != head->prev);

        cout << endl;
    }
};

// Main function
int main() {
    DoublyCircularList list;

    // Insert nodes
    list.insertEnd(10);
    list.insertEnd(20);
    list.insertEnd(30);
    list.insertEnd(40);

    // Display outputs
    list.displayForward();
    list.displayBackward();

    return 0;
}
```

Figure 6: Source Code for the Doubly Circular Linked List (Screenshot- 2)

- **displayForward()** – Prints the list from head to tail using `next` pointers.

- **displayBackward()** – Prints the list from tail to head using `prev` pointers.

`displayBackward()`: Prints tail → head.

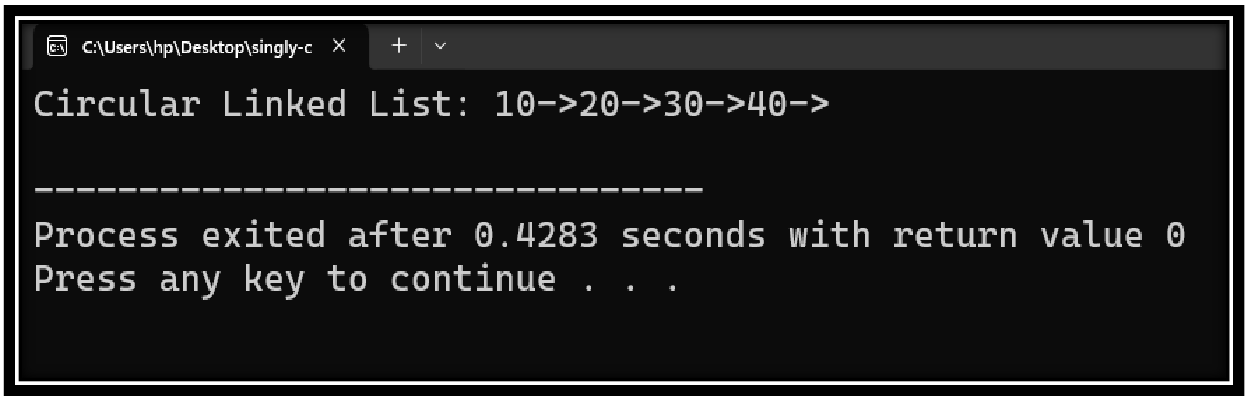
Menu loop: Handles multiple operations with screen refresh.

5. Output Screenshots

5.1 Singly Circular Linked List

Expected Output:

```
Circular Linked List: 10 20 30 40
```



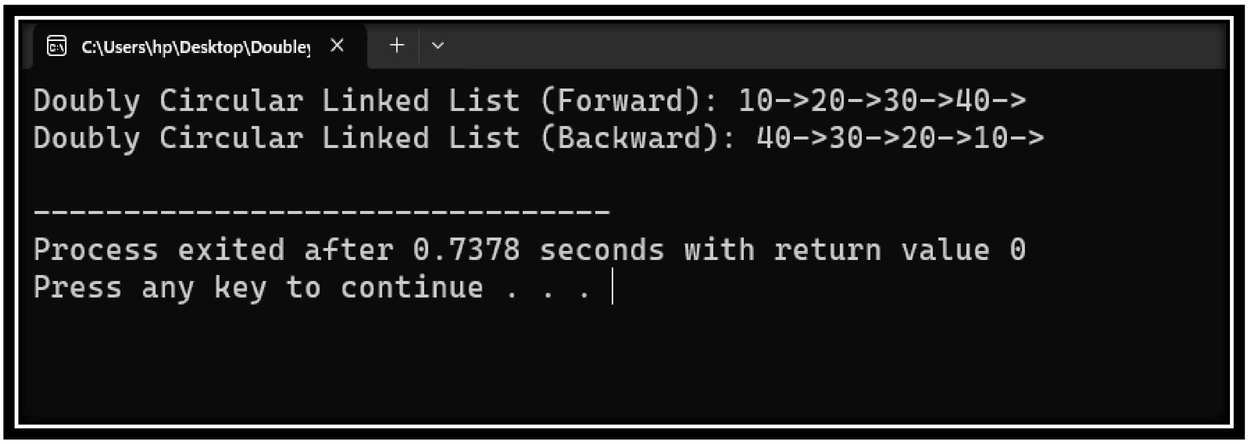
```
Circular Linked List: 10->20->30->40->

--------------------------------
Process exited after 0.4283 seconds with return value 0
Press any key to continue . . .
```

Figure 7: Output for the Singly Circular linked List Code

5.2 Doubly Circular Linked List

Expected Output:

```
Forward: 10 -> 20 -> 30 -> 40
Backward: 40 -> 30 -> 20 -> 10
```



```
Doubly Circular Linked List (Forward): 10->20->30->40->
Doubly Circular Linked List (Backward): 40->30->20->10->

--------------------------------
Process exited after 0.7378 seconds with return value 0
Press any key to continue . . .
```

Figure 8: Output for the Doubly linked List Code

6. Conclusion

- Successfully implemented SCLL and DCLL with insertion, deletion, and traversal operations.
- Demonstrated circular linking, pointer manipulation, and bidirectional traversal in C++.

---

7. Reflection

- Improved understanding of dynamic memory and pointers.
- Learned differences between singly and doubly circular structures.
- Menu-driven operations improve user interactivity and readability.