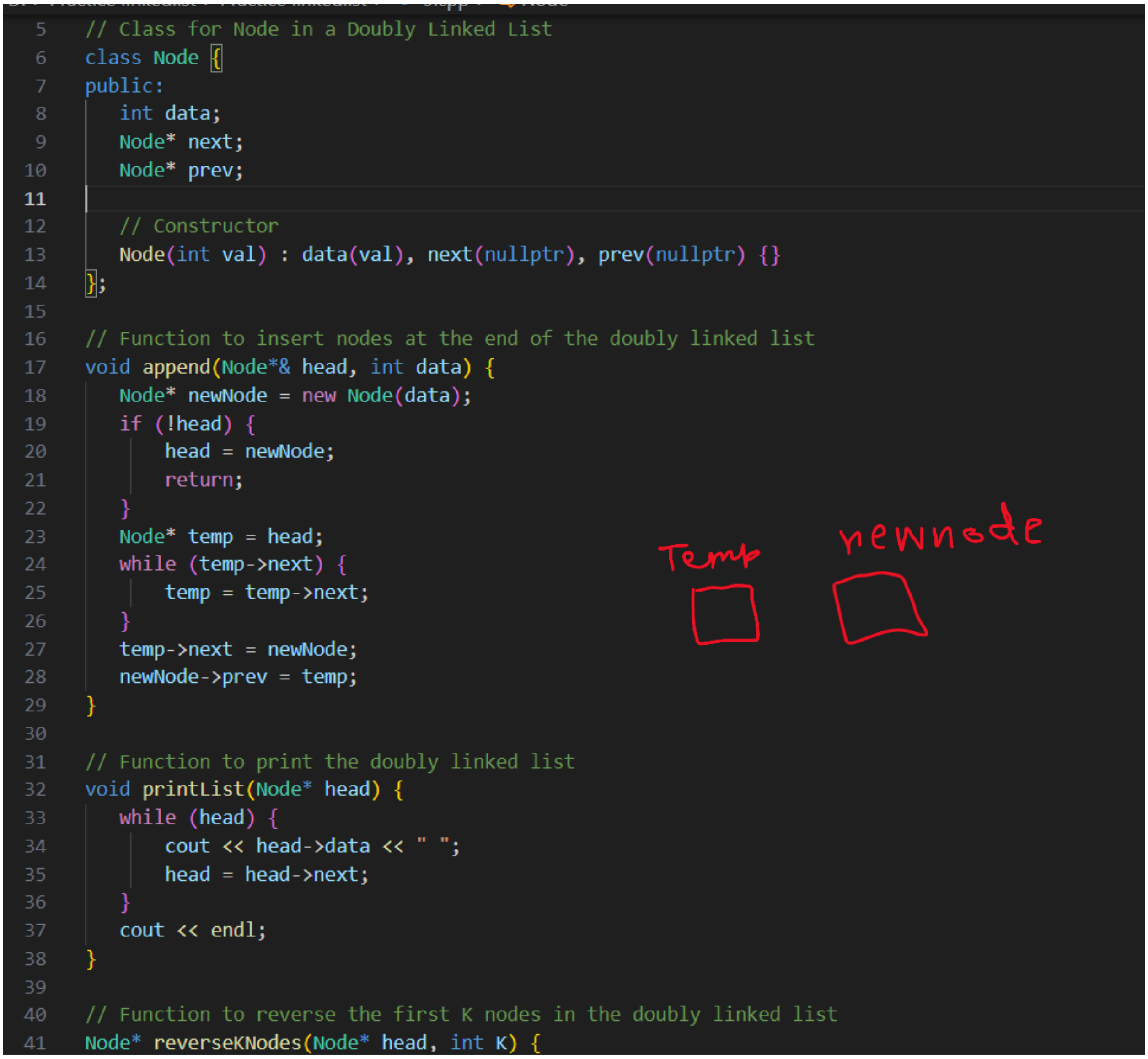
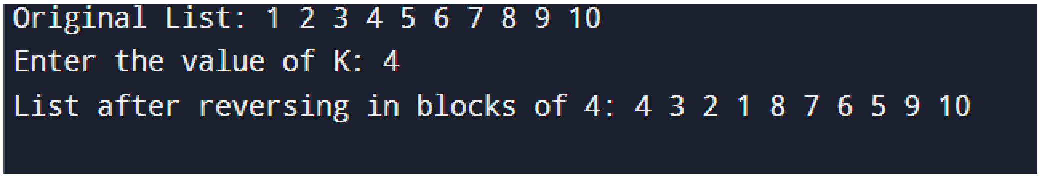
- **Ques 9**

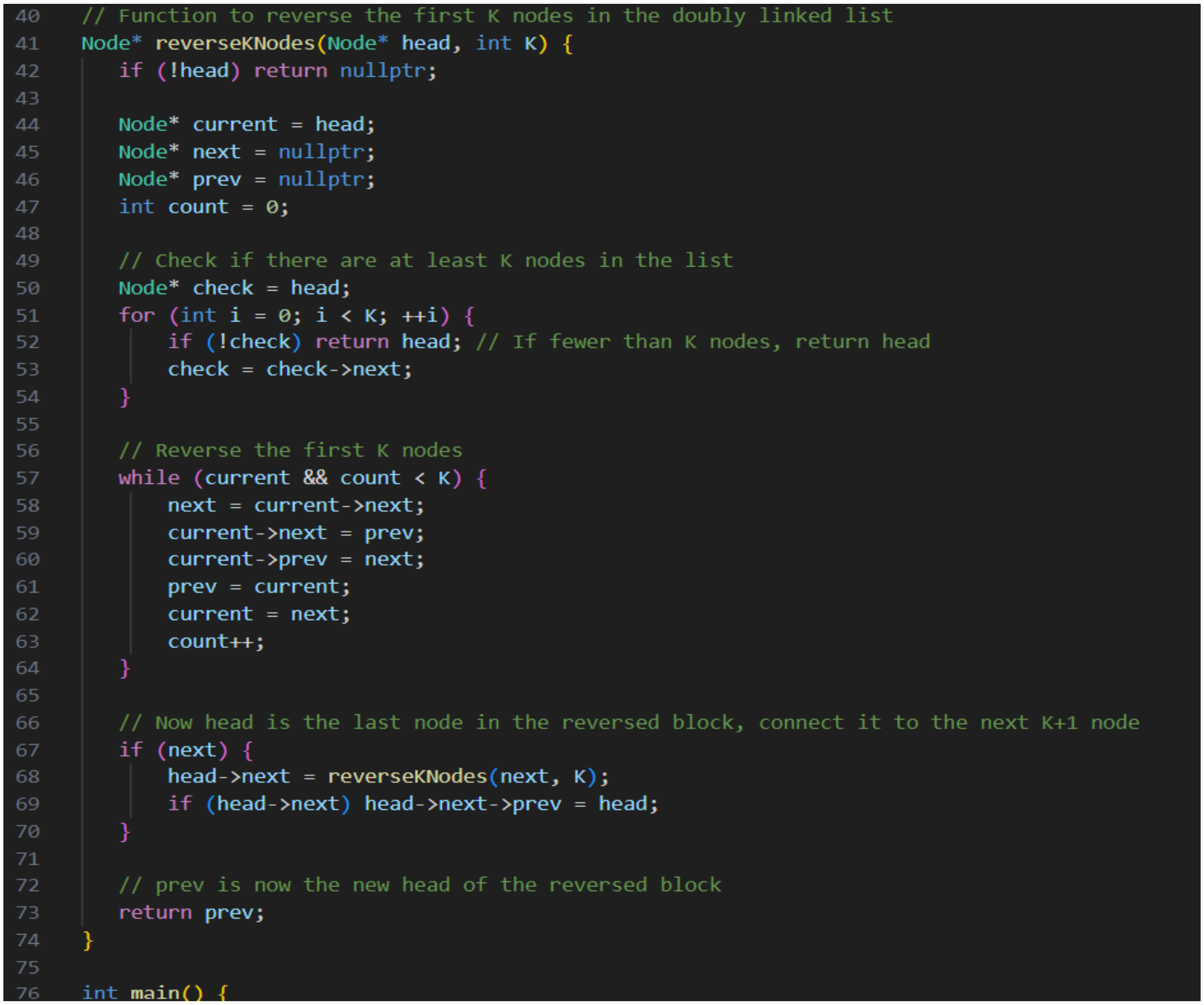


```cpp
// Class for Node in a Doubly Linked List
class Node {
public:
    int data;
    Node* next;
    Node* prev;

    // Constructor
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};

// Function to insert nodes at the end of the doubly linked list
void append(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to print the doubly linked list
void printList(Node* head) {
    while (head) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Function to reverse the first K nodes in the doubly linked list
Node* reverseKNodes(Node* head, int K) {
```

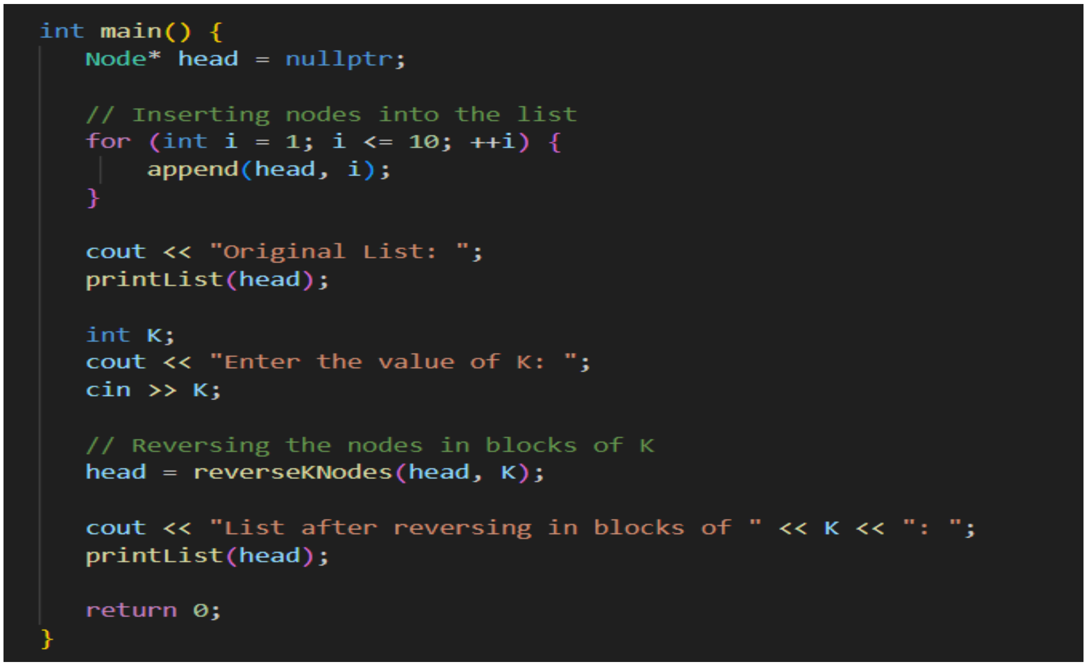
Output:



```
Original List: 1 2 3 4 5 6 7 8 9 10
Enter the value of K: 4
List after reversing in blocks of 4: 4 3 2 1 8 7 6 5 9 10
```

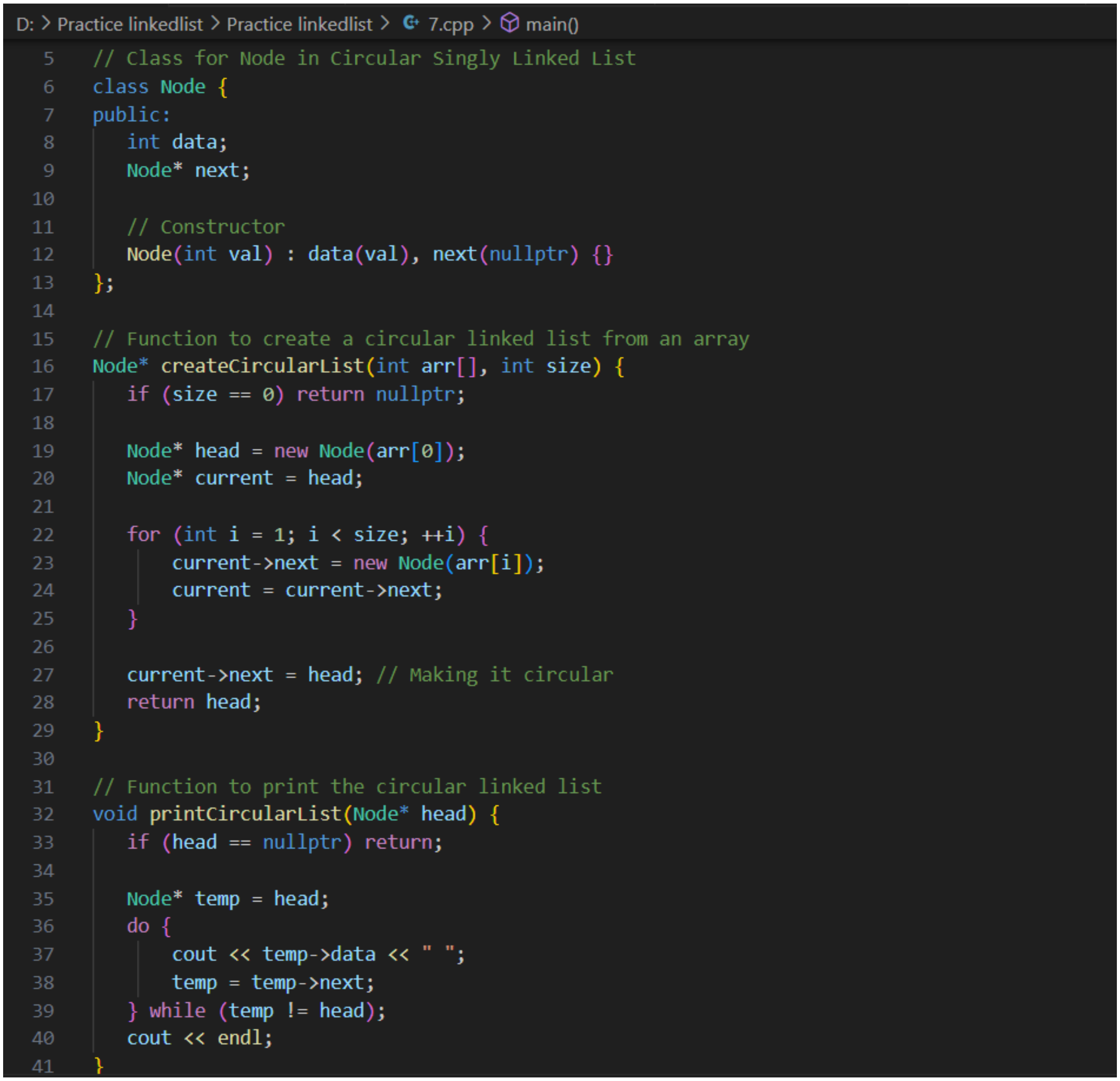


```cpp
// Function to reverse the first K nodes in the doubly linked list
Node* reverseKNodes(Node* head, int K) {
    if (!head) return nullptr;

    Node* current = head;
    Node* next = nullptr;
    Node* prev = nullptr;
    int count = 0;

    // Check if there are at least K nodes in the list
    Node* check = head;
    for (int i = 0; i < K; ++i) {
        if (!check) return head; // If fewer than K nodes, return head
        check = check->next;
    }

    // Reverse the first K nodes
    while (current && count < K) {
        next = current->next;
        current->next = prev;
        current->prev = next;
        prev = current;
        current = next;
        count++;
    }

    // Now head is the last node in the reversed block, connect it to the next K+1 node
    if (next) {
        head->next = reverseKNodes(next, K);
        if (head->next) head->next->prev = head;
    }

    // prev is now the new head of the reversed block
    return prev;
}

int main() {
```

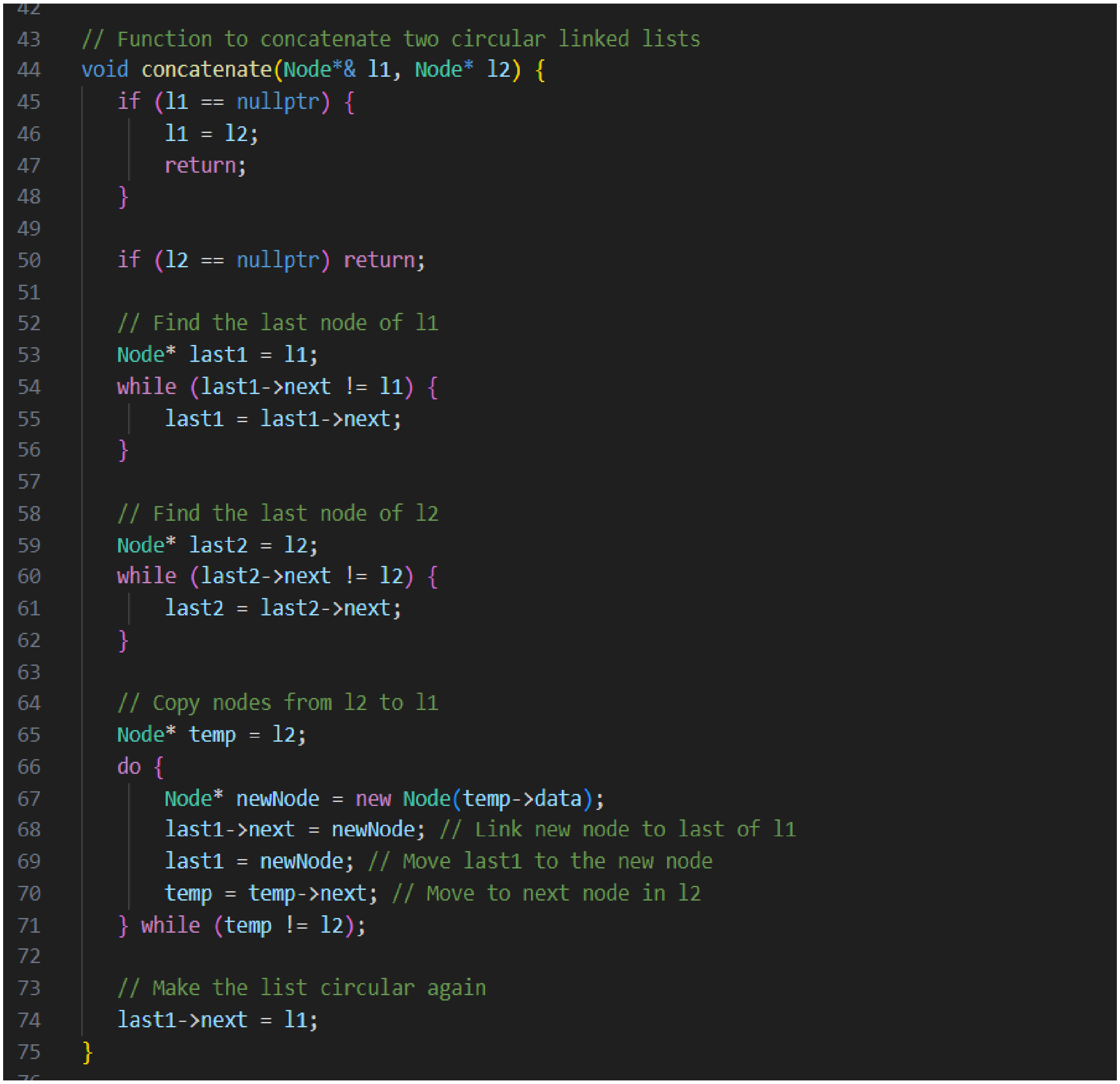


```cpp
int main() {
    Node* head = nullptr;

    // Inserting nodes into the list
    for (int i = 1; i <= 10; ++i) {
        append(head, i);
    }

    cout << "Original List: ";
    printList(head);

    int K;
    cout << "Enter the value of K: ";
    cin >> K;

    // Reversing the nodes in blocks of K
    head = reverseKNodes(head, K);

    cout << "List after reversing in blocks of " << K << ": ";
    printList(head);

    return 0;
}
```

- **Ques 8**

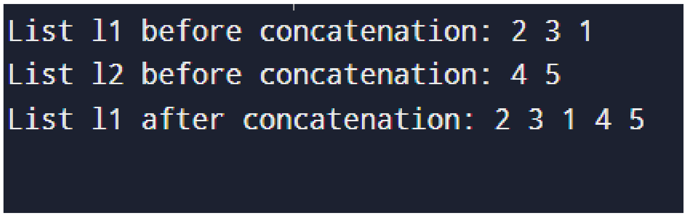

D: > Practice linkedlist > Practice linkedlist > 7.cpp > main()

```cpp
// Class for Node in Circular Singly Linked List
class Node {
public:
    int data;
    Node* next;

    // Constructor
    Node(int val) : data(val), next(nullptr) {}
};

// Function to create a circular linked list from an array
Node* createCircularList(int arr[], int size) {
    if (size == 0) return nullptr;

    Node* head = new Node(arr[0]);
    Node* current = head;

    for (int i = 1; i < size; ++i) {
        current->next = new Node(arr[i]);
        current = current->next;
    }

    current->next = head; // Making it circular
    return head;
}

// Function to print the circular linked list
void printCircularList(Node* head) {
    if (head == nullptr) return;

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

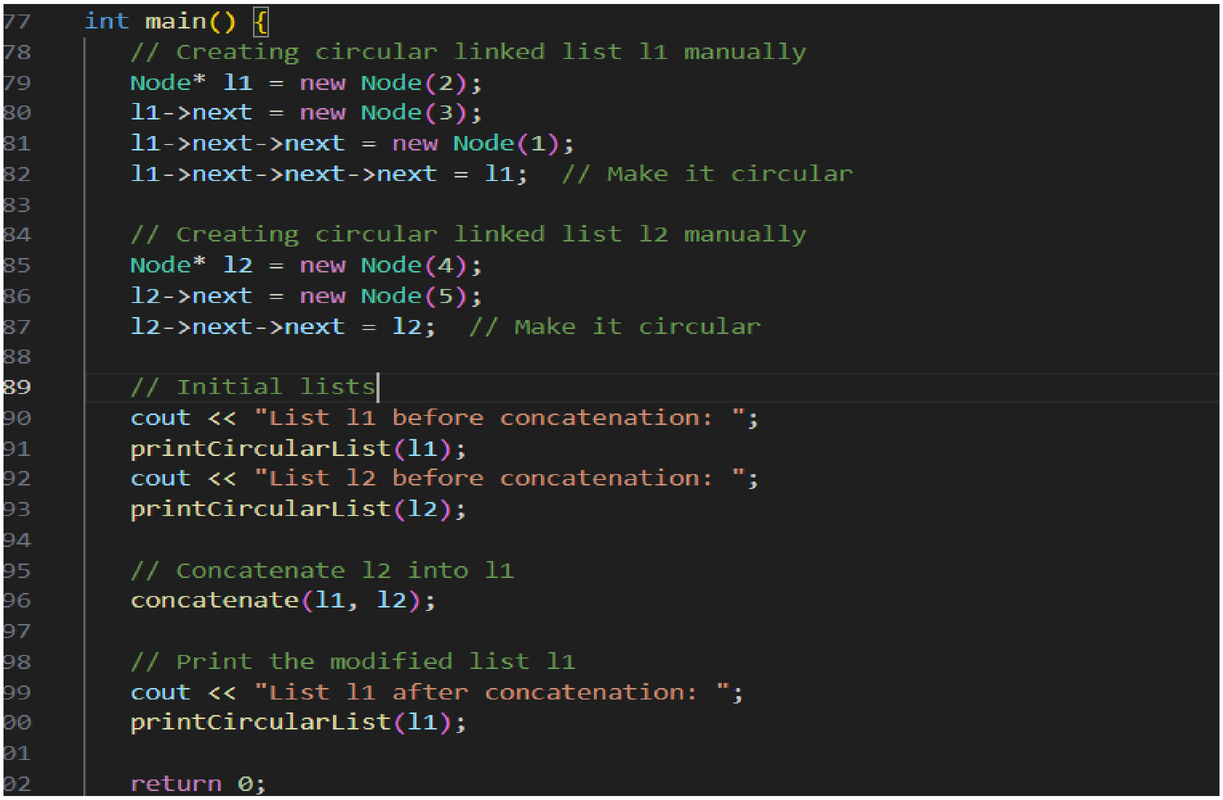


```cpp
// Function to concatenate two circular linked lists
void concatenate(Node*& l1, Node* l2) {
    if (l1 == nullptr) {
        l1 = l2;
        return;
    }

    if (l2 == nullptr) return;

    // Find the last node of l1
    Node* last1 = l1;
    while (last1->next != l1) {
        last1 = last1->next;
    }

    // Find the last node of l2
    Node* last2 = l2;
    while (last2->next != l2) {
        last2 = last2->next;
    }

    // Copy nodes from l2 to l1
    Node* temp = l2;
    do {
        Node* newNode = new Node(temp->data);
        last1->next = newNode; // Link new node to last of l1
        last1 = newNode; // Move last1 to the new node
        temp = temp->next; // Move to next node in l2
    } while (temp != l2);

    // Make the list circular again
    last1->next = l1;
}
```

Output:

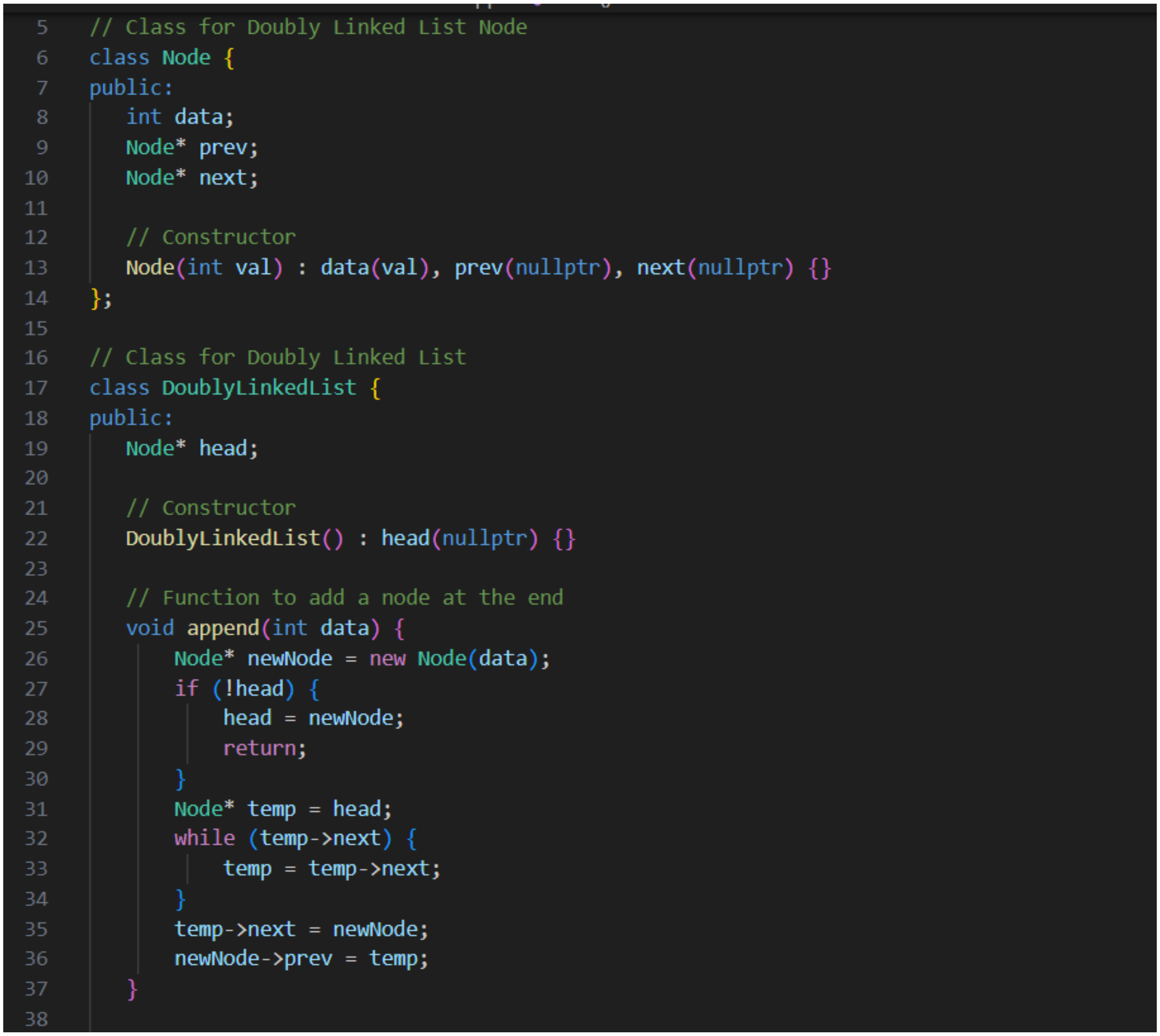


```
List l1 before concatenation: 2 3 1
List l2 before concatenation: 4 5
List l1 after concatenation: 2 3 1 4 5
```

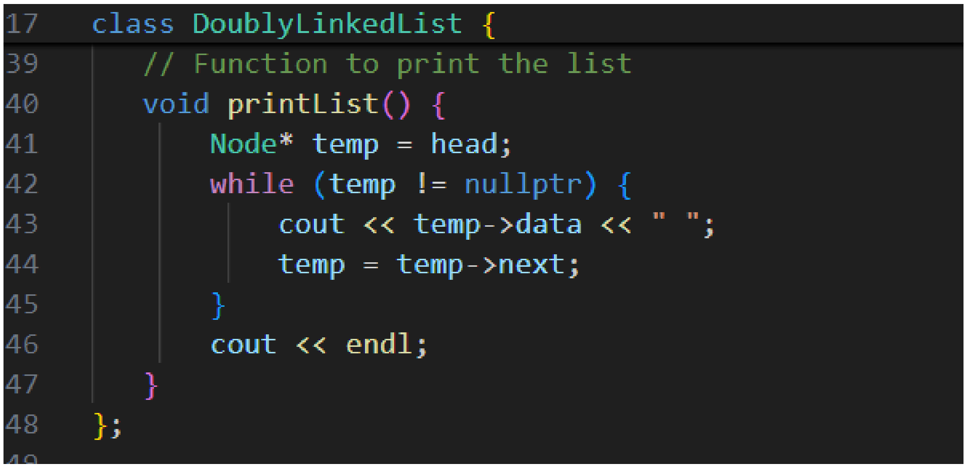


```
int main() {
    // Creating circular linked list l1 manually
    Node* l1 = new Node(2);
    l1->next = new Node(3);
    l1->next->next = new Node(1);
    l1->next->next->next = l1;  // Make it circular

    // Creating circular linked list l2 manually
    Node* l2 = new Node(4);
    l2->next = new Node(5);
    l2->next->next = l2;  // Make it circular

    // Initial lists
    cout << "List l1 before concatenation: ";
    printCircularList(l1);
    cout << "List l2 before concatenation: ";
    printCircularList(l2);

    // Concatenate l2 into l1
    concatenate(l1, l2);

    // Print the modified list l1
    cout << "List l1 after concatenation: ";
    printCircularList(l1);

    return 0;
```

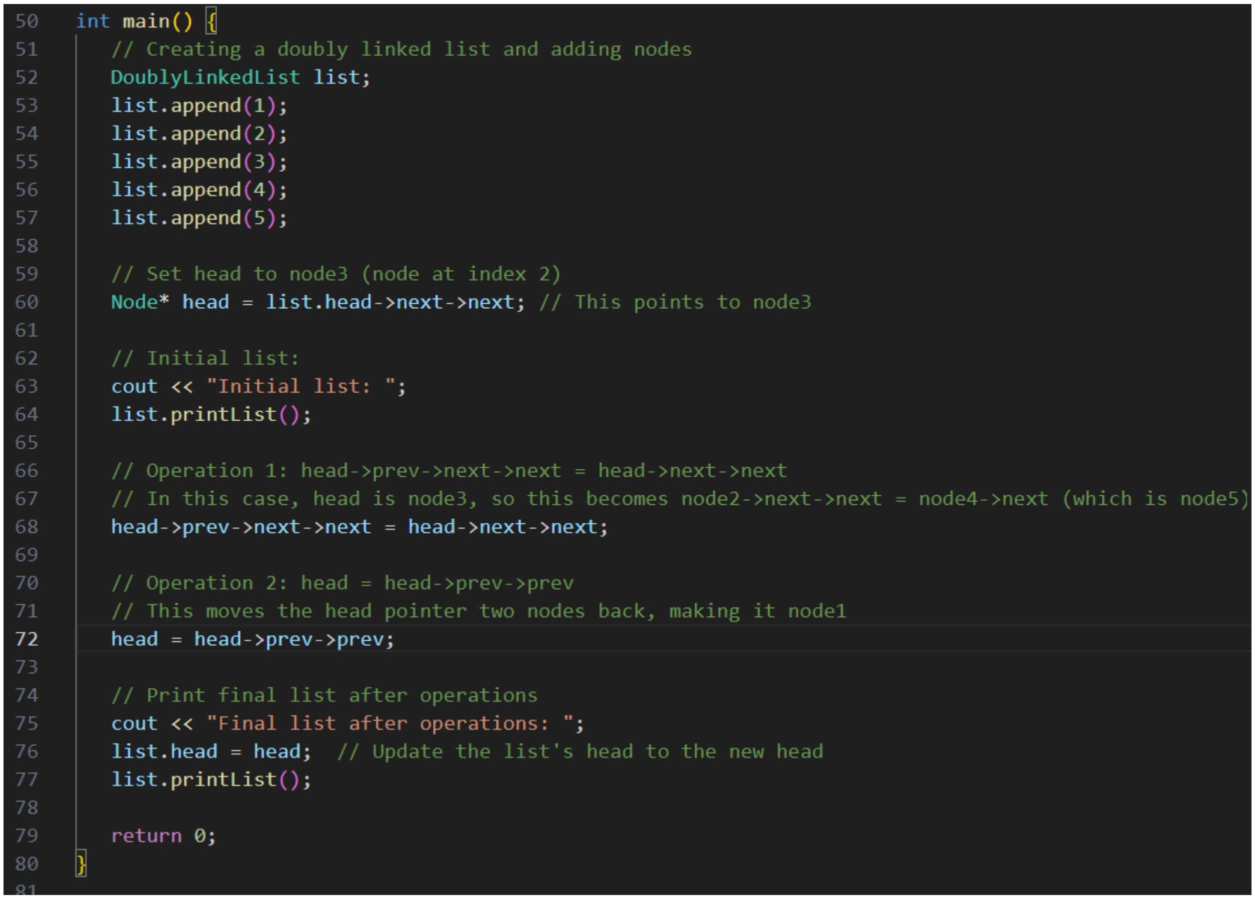
- **Ques 6**



```
// Class for Doubly Linked List Node
class Node {
public:
    int data;
    Node* prev;
    Node* next;

    // Constructor
    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};

// Class for Doubly Linked List
class DoublyLinkedList {
public:
    Node* head;

    // Constructor
    DoublyLinkedList() : head(nullptr) {}

    // Function to add a node at the end
    void append(int data) {
        Node* newNode = new Node(data);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }

```

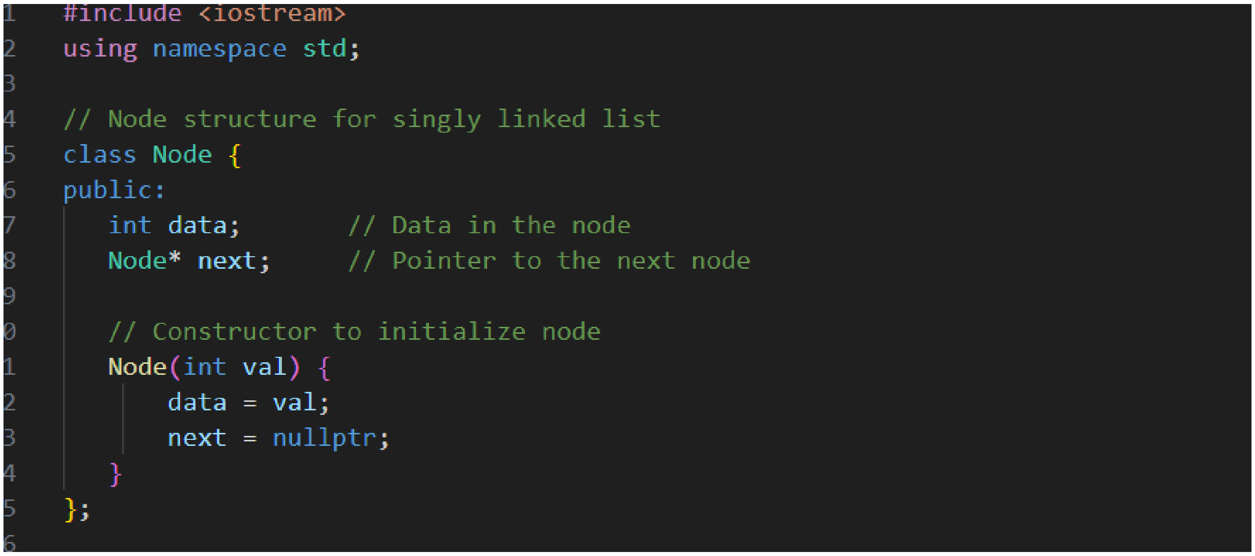


```
class DoublyLinkedList {
    // Function to print the list
    void printList() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

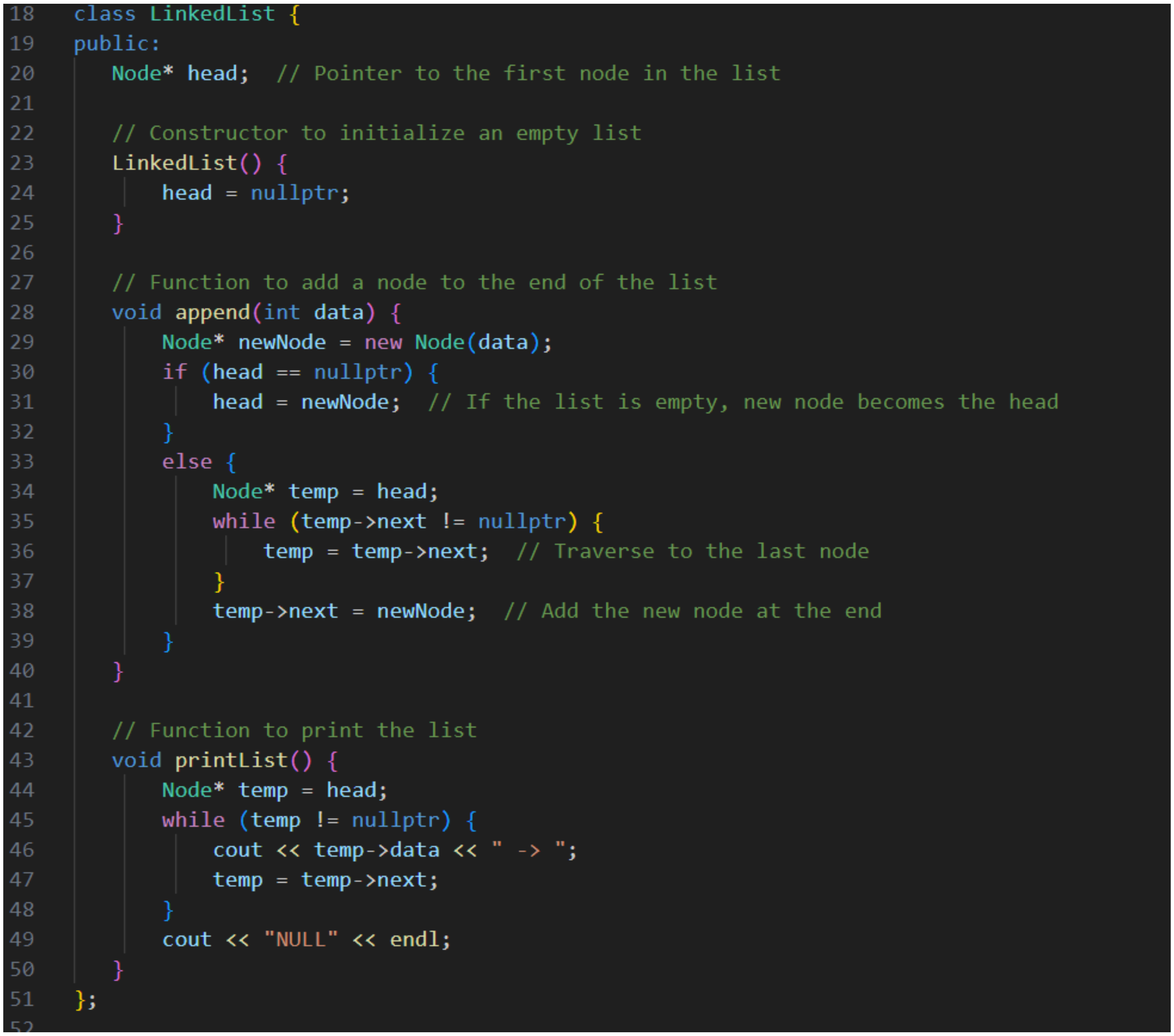


```cpp
int main() {
    // Creating a doubly linked list and adding nodes
    DoublyLinkedList list;
    list.append(1);
    list.append(2);
    list.append(3);
    list.append(4);
    list.append(5);

    // Set head to node3 (node at index 2)
    Node* head = list.head->next->next; // This points to node3

    // Initial list:
    cout << "Initial list: ";
    list.printList();

    // Operation 1: head->prev->next->next = head->next->next
    // In this case, head is node3, so this becomes node2->next->next = node4->next (which is node5)
    head->prev->next->next = head->next->next;

    // Operation 2: head = head->prev->prev
    // This moves the head pointer two nodes back, making it node1
    head = head->prev->prev;

    // Print final list after operations
    cout << "Final list after operations: ";
    list.head = head;  // Update the list's head to the new head
    list.printList();

    return 0;
}
```

- **Ques 5:**

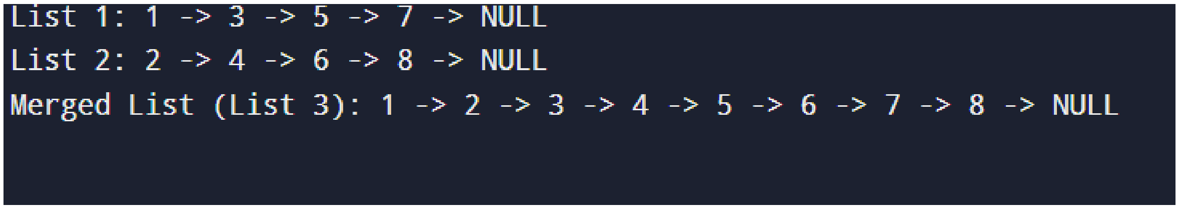


```cpp
#include <iostream>
using namespace std;

// Node structure for singly linked list
class Node {
public:
    int data;       // Data in the node
    Node* next;     // Pointer to the next node

    // Constructor to initialize node
    Node(int val) {
        data = val;
        next = nullptr;
    }
};
```

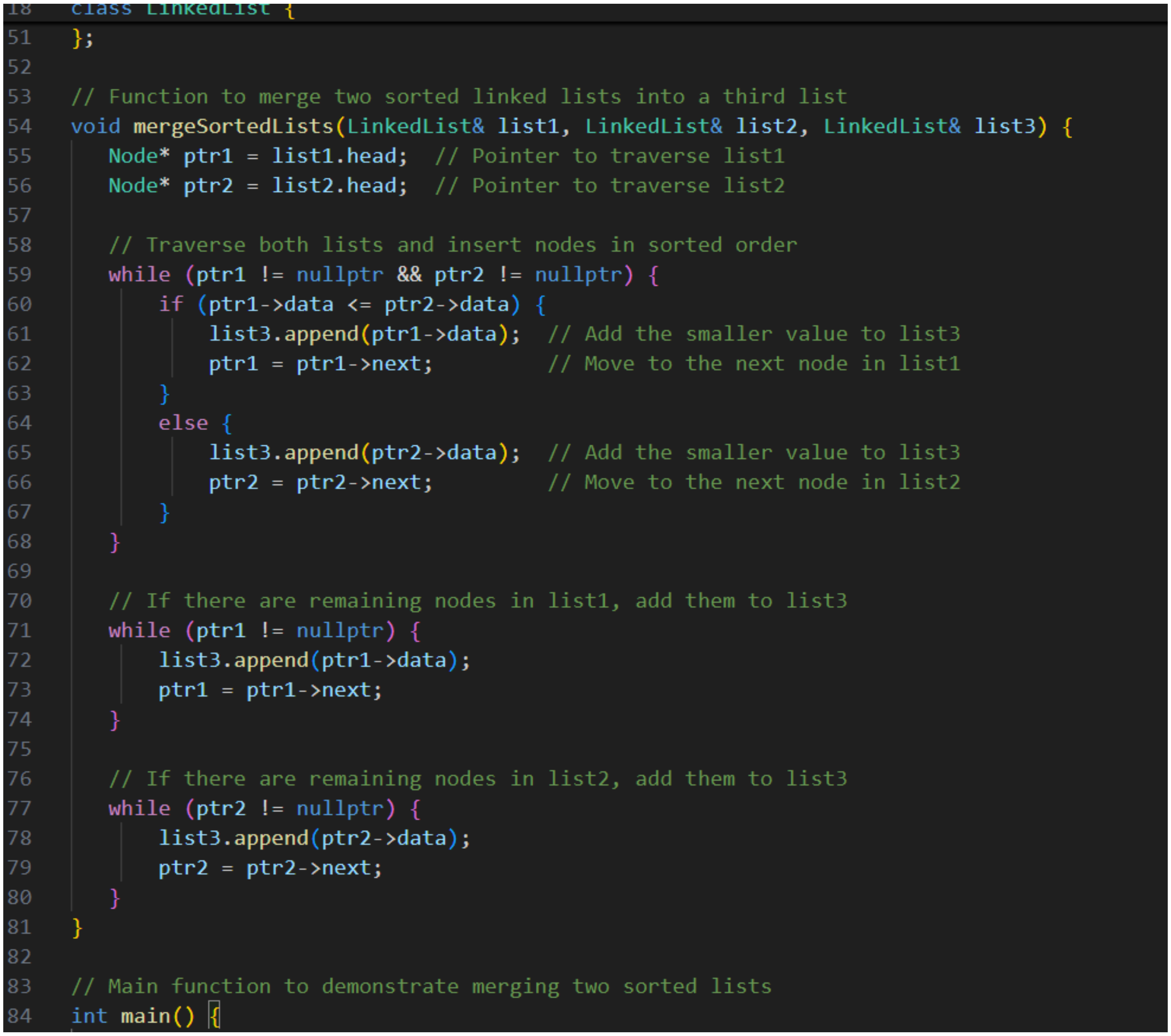


```cpp
class LinkedList {
public:
    Node* head;  // Pointer to the first node in the list

    // Constructor to initialize an empty list
    LinkedList() {
        head = nullptr;
    }

    // Function to add a node to the end of the list
    void append(int data) {
        Node* newNode = new Node(data);
        if (head == nullptr) {
            head = newNode;  // If the list is empty, new node becomes the head
        }
        else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;  // Traverse to the last node
            }
            temp->next = newNode;  // Add the new node at the end
        }
    }

    // Function to print the list
    void printList() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }
};
```

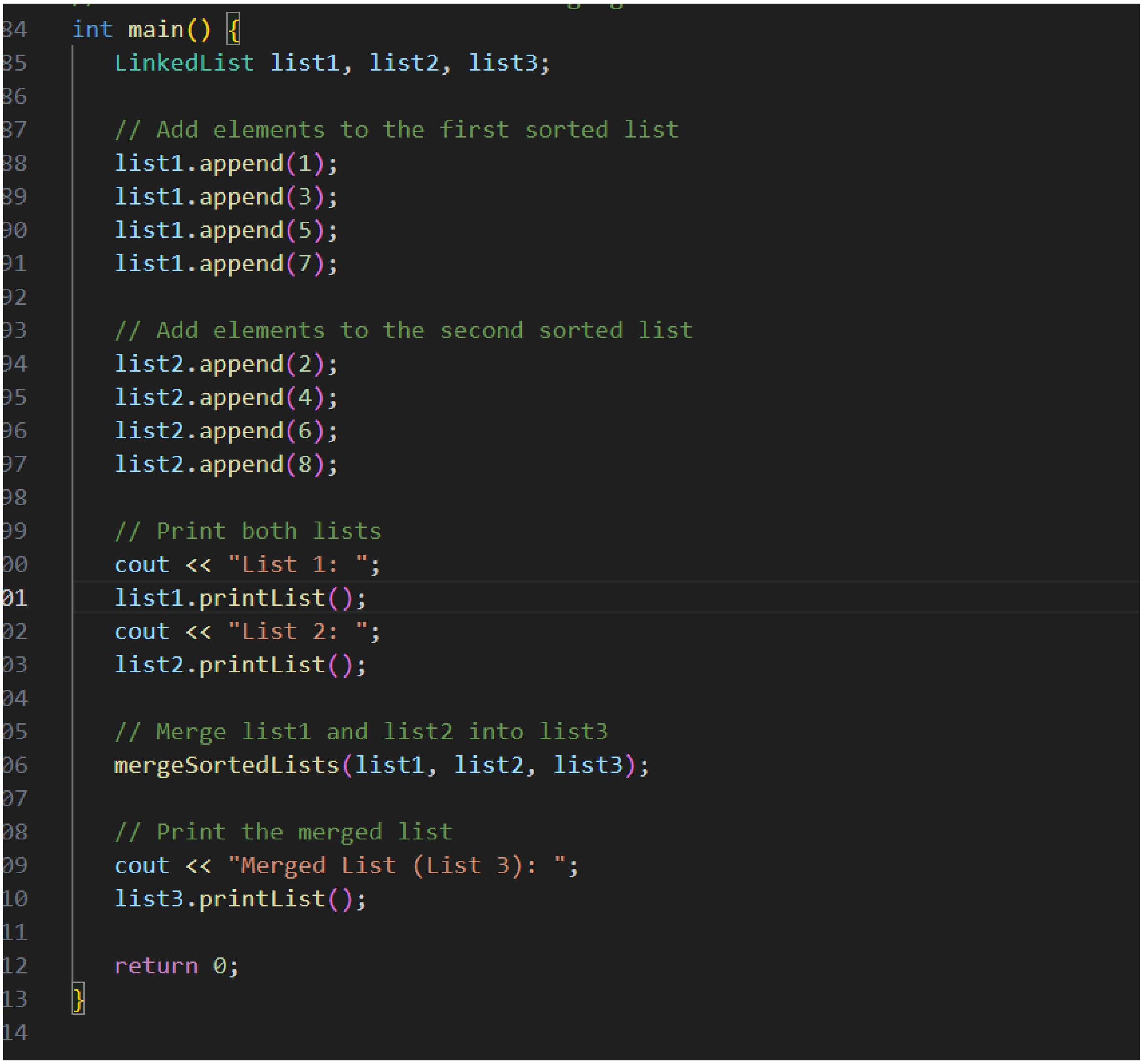
Output:



```
List 1: 1 -> 3 -> 5 -> 7 -> NULL
List 2: 2 -> 4 -> 6 -> 8 -> NULL
Merged List (List 3): 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> NULL
```



```cpp
class LinkedList {
};

// Function to merge two sorted linked lists into a third list
void mergeSortedLists(LinkedList& list1, LinkedList& list2, LinkedList& list3) {
    Node* ptr1 = list1.head;  // Pointer to traverse list1
    Node* ptr2 = list2.head;  // Pointer to traverse list2

    // Traverse both lists and insert nodes in sorted order
    while (ptr1 != nullptr && ptr2 != nullptr) {
        if (ptr1->data <= ptr2->data) {
            list3.append(ptr1->data);  // Add the smaller value to list3
            ptr1 = ptr1->next;         // Move to the next node in list1
        }
        else {
            list3.append(ptr2->data);  // Add the smaller value to list3
            ptr2 = ptr2->next;         // Move to the next node in list2
        }
    }

    // If there are remaining nodes in list1, add them to list3
    while (ptr1 != nullptr) {
        list3.append(ptr1->data);
        ptr1 = ptr1->next;
    }

    // If there are remaining nodes in list2, add them to list3
    while (ptr2 != nullptr) {
        list3.append(ptr2->data);
        ptr2 = ptr2->next;
    }
}

// Main function to demonstrate merging two sorted lists
int main() {
```



```cpp
int main() {
    LinkedList list1, list2, list3;

    // Add elements to the first sorted list
    list1.append(1);
    list1.append(3);
    list1.append(5);
    list1.append(7);

    // Add elements to the second sorted list
    list2.append(2);
    list2.append(4);
    list2.append(6);
    list2.append(8);

    // Print both lists
    cout << "List 1: ";
    list1.printList();
    cout << "List 2: ";
    list2.printList();

    // Merge list1 and list2 into list3
    mergeSortedLists(list1, list2, list3);

    // Print the merged list
    cout << "Merged List (List 3): ";
    list3.printList();

    return 0;
}
```