CC-213L

Data Structures and Algorithms

Laboratory 06

Singly Linked List

Version: 1.0.0

Release Date: 25-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Pointers and Dynamic Memory Allocation
- Self-Referential Objects
- Singly LinkedList

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Muhammad Tahir Mustafvi	bcsf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

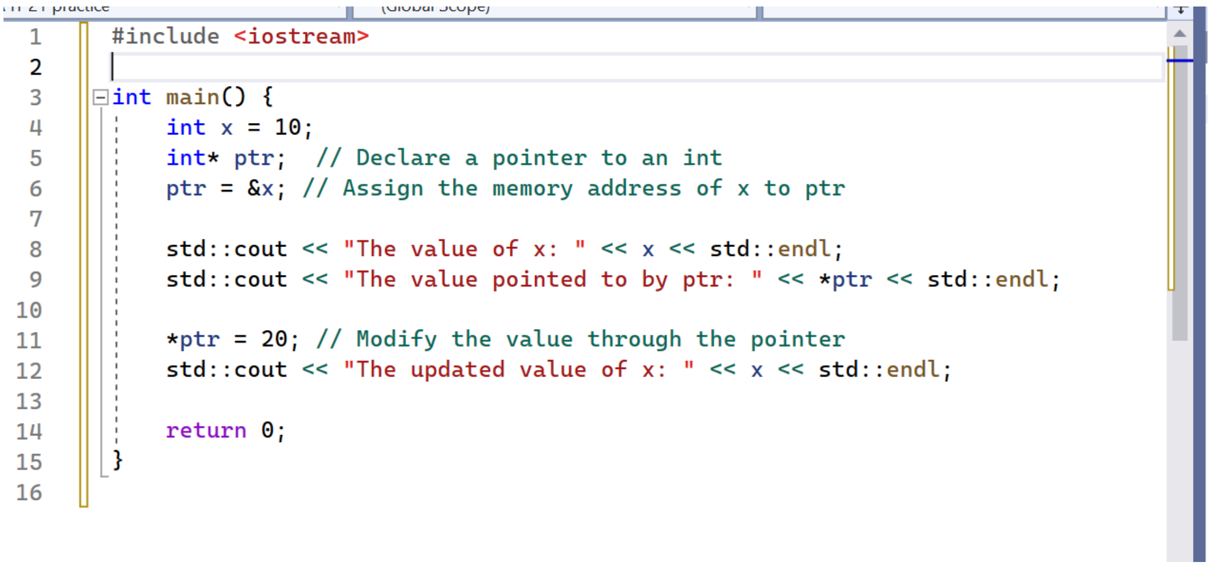
Background and Overview

Pointers and Dynamic Memory Allocation

Pointers and dynamic memory allocation are important concepts in programming, particularly in languages like C and C++. Pointers allow you to work with memory addresses, while dynamic memory allocation allows you to manage memory at runtime.

Pointers:

A pointer is a variable that stores the memory address of another variable. It allows you to indirectly access the value of the variable stored at that address. Pointers are often used for various purposes, such as dynamically allocated memory, working with arrays, and passing functions as arguments.

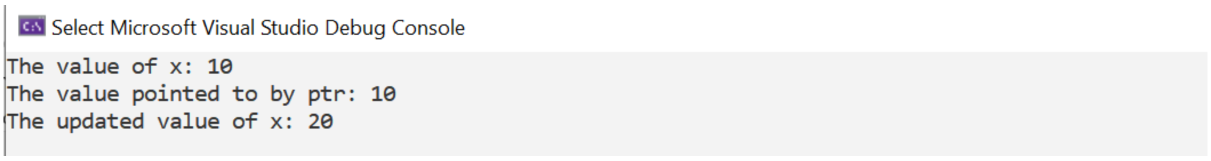


```cpp
#include <iostream>

int main() {
    int x = 10;
    int* ptr;  // Declare a pointer to an int
    ptr = &x; // Assign the memory address of x to ptr

    std::cout << "The value of x: " << x << std::endl;
    std::cout << "The value pointed to by ptr: " << *ptr << std::endl;

    *ptr = 20; // Modify the value through the pointer
    std::cout << "The updated value of x: " << x << std::endl;

    return 0;
}
```

Figure 1(Pointers)

Explanation:

In this example, ptr is a pointer to an integer, and it is assigned the memory address of the variable x. You can access and modify the value of x through the pointer using the dereference operator (*ptr).

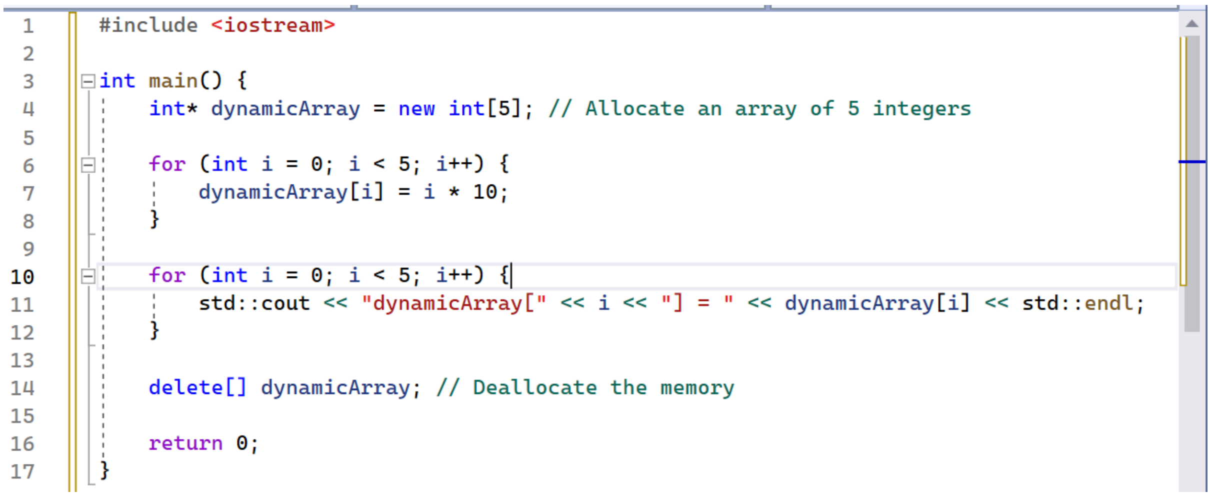


```
The value of x: 10
The value pointed to by ptr: 10
The updated value of x: 20
```

Figure 2(output)

Dynamic Memory Allocation

Dynamic memory allocation allows you to allocate memory for variables at runtime. In C++, you can use new and delete operators to allocate and deallocate memory for objects on the heap.



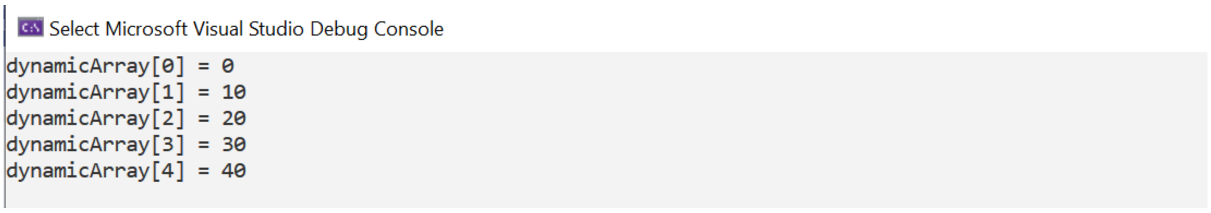
```cpp
#include <iostream>

int main() {
    int* dynamicArray = new int[5]; // Allocate an array of 5 integers

    for (int i = 0; i < 5; i++) {
        dynamicArray[i] = i * 10;
    }

    for (int i = 0; i < 5; i++) {
        std::cout << "dynamicArray[" << i << "] = " << dynamicArray[i] << std::endl;
    }

    delete[] dynamicArray; // Deallocate the memory

    return 0;
}
```

Figure 3(Dynamic Memory)

Explanation:

In this example, dynamicArray is allocated on the heap with space for 5 integers. After using it, it is essential to deallocate the memory using delete[] to prevent memory leaks.

Note: In modern C++ (C++11 and later), it is recommended to use smart pointers like std::unique_ptr and std::shared_ptr for better memory management, as they automatically handle memory deallocation.



Select Microsoft Visual Studio Debug Console

```
dynamicArray[0] = 0
dynamicArray[1] = 10
dynamicArray[2] = 20
dynamicArray[3] = 30
dynamicArray[4] = 40
```

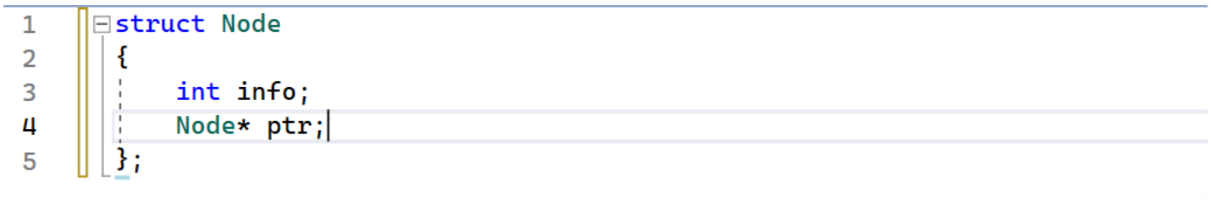
Figure 4(Output)

Self-Referential Objects:

Classes that have the capability to refer to their own types of objects are called **Self Referential Classes/Structs.** Objects of such classes are called self-referential Objects.

Self-referential structure in C++ are those structures that contain one or more than one pointer as their member which will be pointing to the structure of the same type. In simple words, a structure that is pointing to the structure of the same type is known as a self-referential structure.

Example in C++



```cpp
struct Node
{
    int info;
    Node* ptr;
};
```

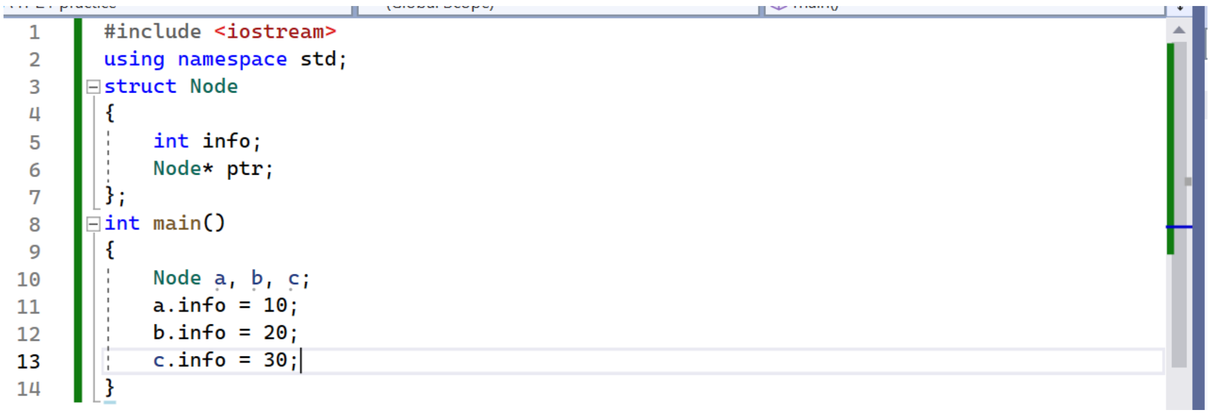
Figure 5(Self Referencing)

Explanation:

In Figure 5 we have declared a struct Node. It has two data members info and ptr.

Info Represents the information data part. Enables the object to store relevant information in it. There can be more than one identifier of same/different datatypes depending upon the application /situation.

ptr Represents the link part. Enables the object to be a self-referential object. There can be more than one such reference used for different purposes in different applications /situations.

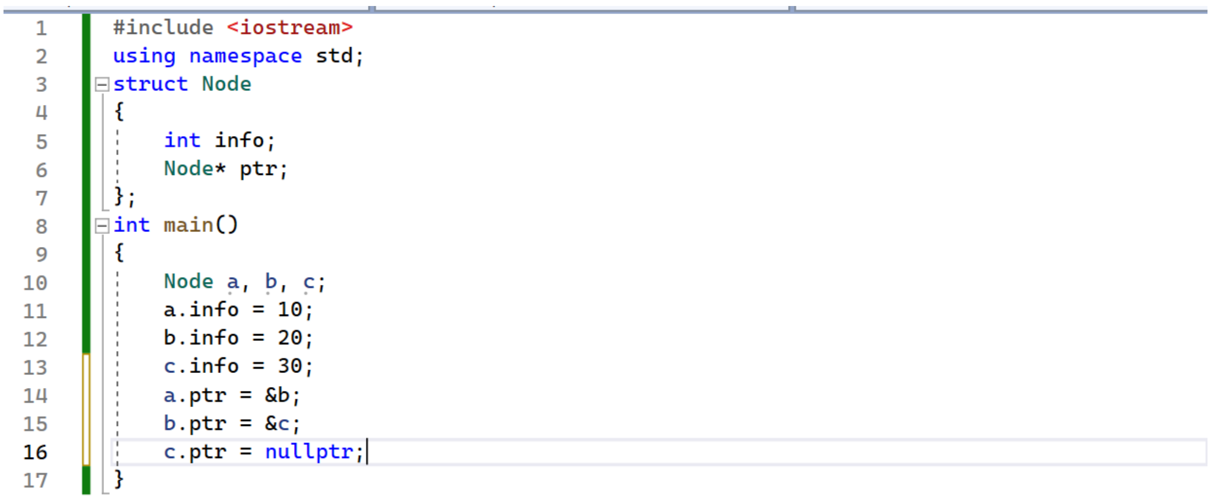


```
#include <iostream>
using namespace std;
struct Node
{
    int info;
    Node* ptr;
};
int main()
{
    Node a, b, c;
    a.info = 10;
    b.info = 20;
    c.info = 30;
}
```

Figure 6(Self Referential Objects)

Explanation:

We have declared three Node types of variables a, b and stored proper values in their info data member.



```
#include <iostream>
using namespace std;
struct Node
{
    int info;
    Node* ptr;
};
int main()
{
    Node a, b, c;
    a.info = 10;
    b.info = 20;
    c.info = 30;
    a.ptr = &b;
    b.ptr = &c;
    c.ptr = nullptr;
}
```

Figure 7(Self Referencing)

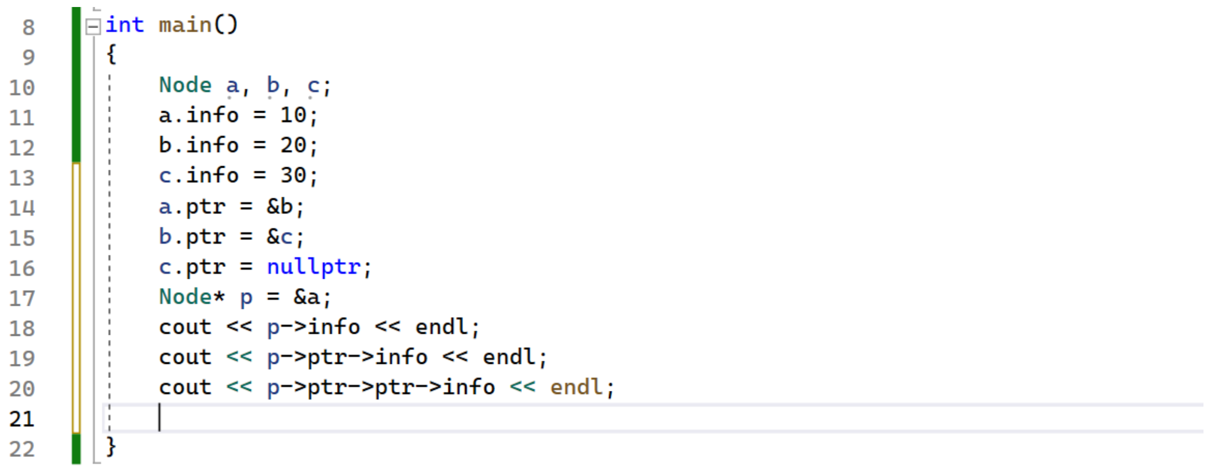
Explanation:

We have declared three Node types variables a, b and stored proper values in their info data member.

At line number 14 the ptr variable of type Node* (pointer to Node) is assigned to the address of Node b and similarly at line 15 ptr variable of Node b is assigned address of Node c. ptr of Node c is pointing to null.

Some Important Operators:

- ***Member Access operators*** arrow operator (→) and dot operator (**.**) have same precedence with associativity from left to right.
- ***Dereference /indirection operator (*)*** *address* ***operator (&)*** have same precedence but associativity from right to left.
- Member access operators have higher priority than indirection and address operators.

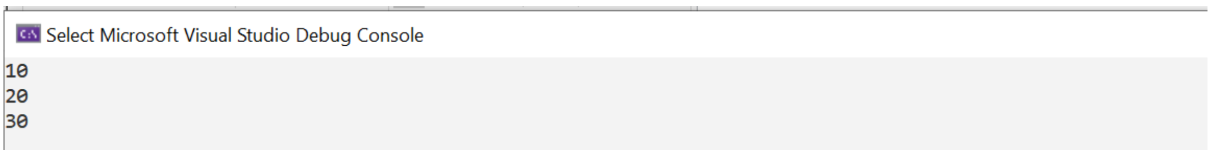


```
8   int main()
9   {
10      Node a, b, c;
11      a.info = 10;
12      b.info = 20;
13      c.info = 30;
14      a.ptr = &b;
15      b.ptr = &c;
16      c.ptr = nullptr;
17      Node* p = &a;
18      cout << p->info << endl;
19      cout << p->ptr->info << endl;
20      cout << p->ptr->ptr->info << endl;
21      
22  }
```

Figure 8(Member Access)

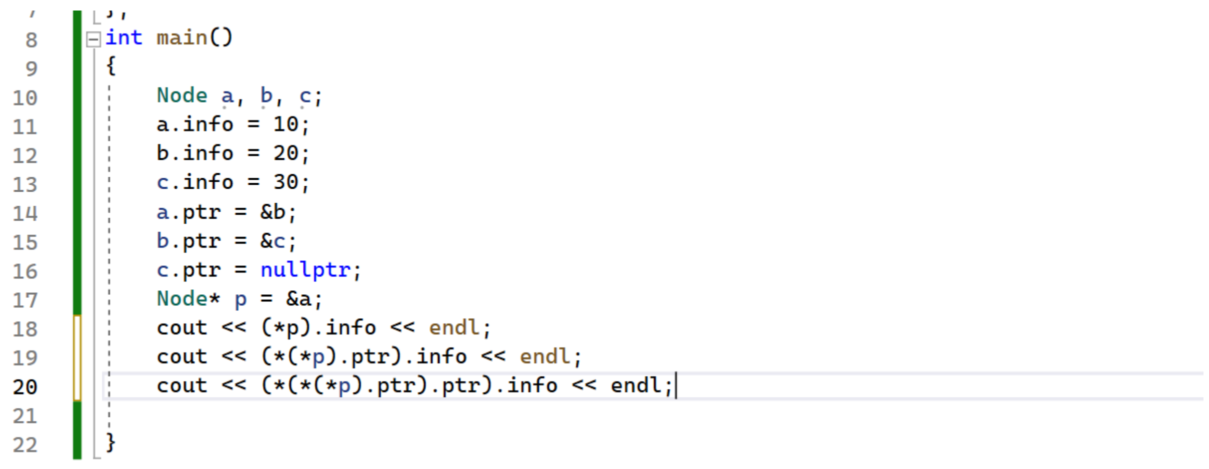
Explanation:

On line 17, a pointer of type Node named **p** is declared. This pointer will be used to reference **a** node. With this **p** pointer, it becomes straightforward to traverse through the linked list, as each node contains a **ptr** member that points to the next node in the sequence. We have accessed all the next node as well as information through the → (pointer member access operator).



```
10
20
30
```

Figure 9(Output)

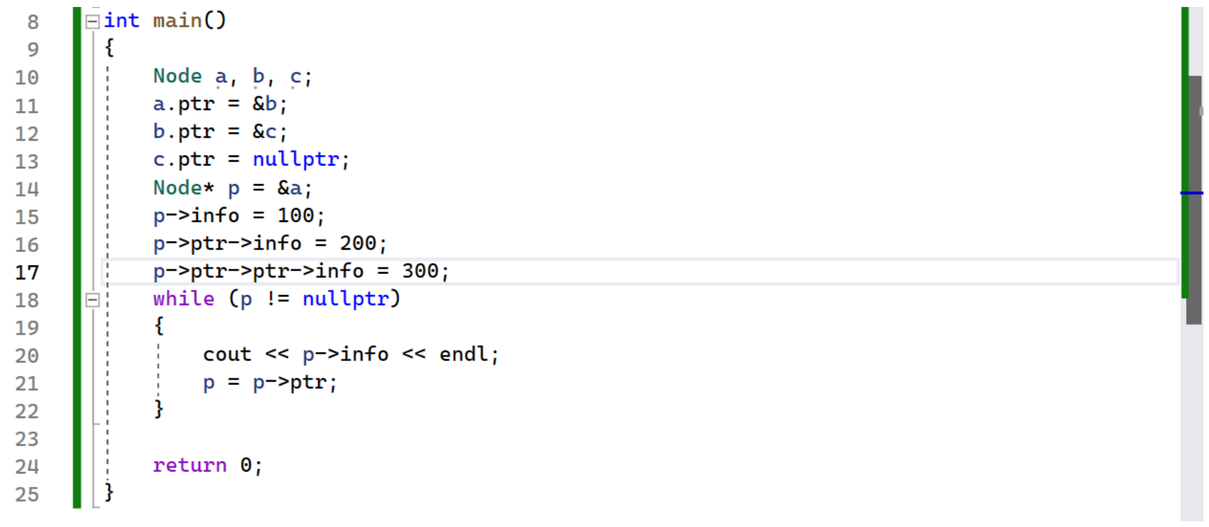


```
 8  int main()
 9  {
10      Node a, b, c;
11      a.info = 10;
12      b.info = 20;
13      c.info = 30;
14      a.ptr = &b;
15      b.ptr = &c;
16      c.ptr = nullptr;
17      Node* p = &a;
18      cout << (*p).info << endl;
19      cout << (*(*p).ptr).info << endl;
20      cout << (*(*(*p).ptr).ptr).info << endl;
21
22  }
```

Figure 10(Indirection Operator)

Explanation:

At line number 18,19 and 20 members have been accessing through the indirection and dot operator that is object member access operator.

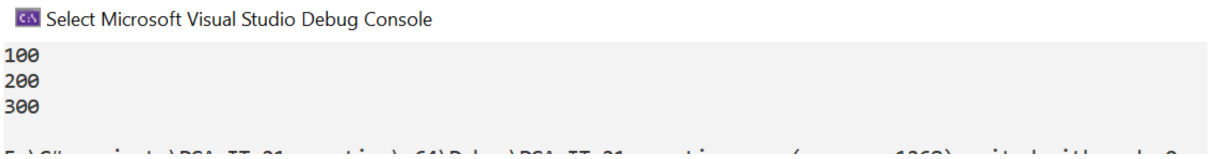


```
int main()
{
    Node a, b, c;
    a.ptr = &b;
    b.ptr = &c;
    c.ptr = nullptr;
    Node* p = &a;
    p->info = 100;
    p->ptr->info = 200;
    p->ptr->ptr->info = 300;
    while (p != nullptr)
    {
        cout << p->info << endl;
        p = p->ptr;
    }

    return 0;
}
```

Figure 11(Traverse Nodes)

Explanation:

Rather than individually accessing information from each node, a more efficient approach is to employ a loop for traversing the nodes. This way, you can access the information in each node if you haven't reached a node with its **ptr** member pointing to **nullptr.**



```
100
200
300
```

Figure 12(Output)

Singly LinkedList

A linked list is a data structure used in computer science and programming to organize and store a collection of elements. It consists of a sequence of nodes, where each node contains two components:

Data: This component holds the actual value or information you want to store.

Pointer (or reference): This component points to the next node in the sequence, effectively linking the nodes together.

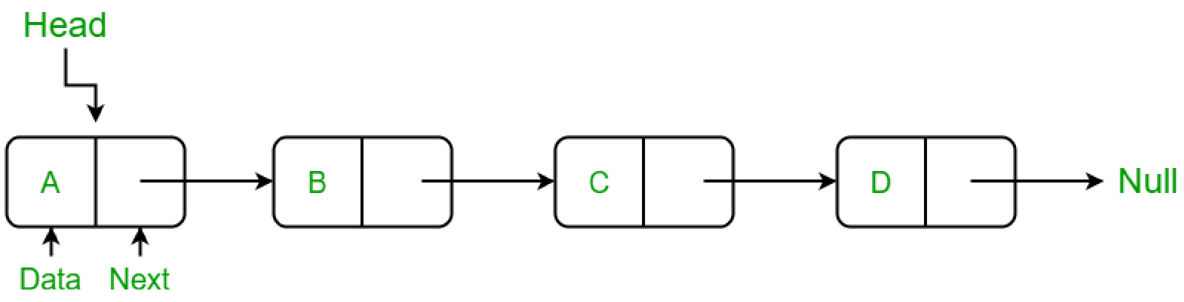
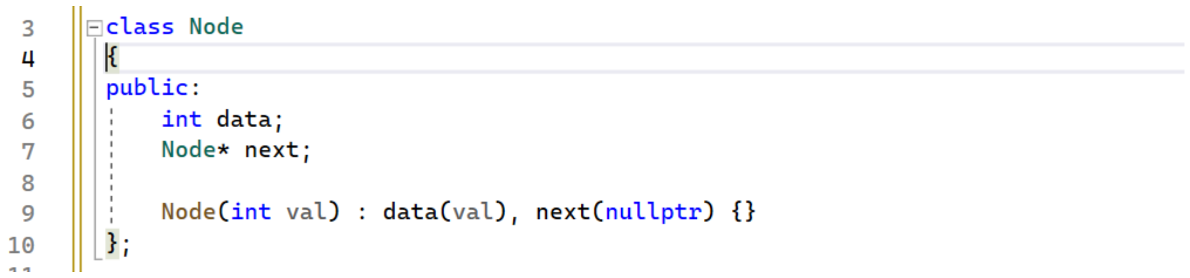


Figure 13(Singly LinkedList)

In a singly linked list, each node points to the next node in the list, forming a linear structure. The last node typically points to a special value like nullptr or NULL to indicate the end of the list.

Linked lists come in various forms, including singly linked lists (described above), doubly linked lists (where each node has a pointer to both the next and the previous node), and circular linked lists (where the last node points back to the first node, forming a closed loop).

Linked lists are particularly useful when you need to efficiently insert or delete elements at arbitrary positions in the list, as these operations typically involve updating a few pointers. They often contrasted with arrays, which have fixed sizes and require elements to be shifted when inserting or deleting in the middle.



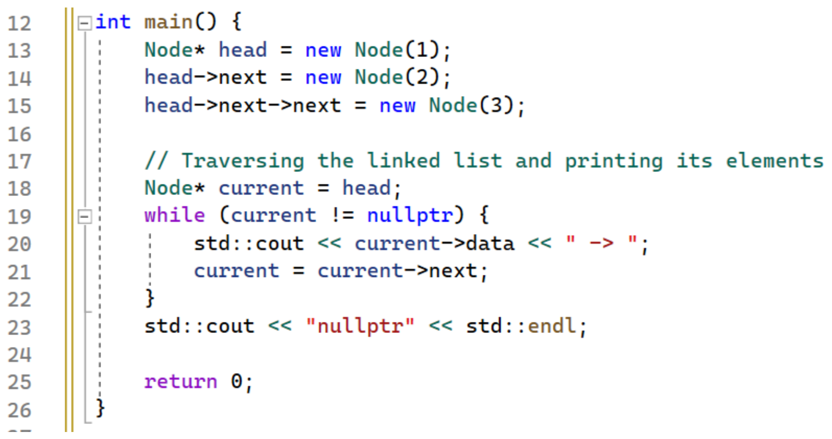
```cpp
class Node
{
public:
    int data;
    Node* next;

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

Figure 14(class Node)

Explanation:

A class Node has been declared that is self-referential class.

Insert Node:

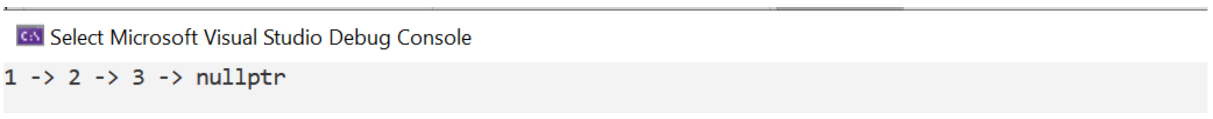


```cpp
int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);

    // Traversing the linked list and printing its elements
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " -> ";
        current = current->next;
    }
    std::cout << "nullptr" << std::endl;

    return 0;
}
```

Figure 15(Linked list)

Explanation:

In this example, a singly linked list with three nodes is created, and then the elements are traversed and printed.

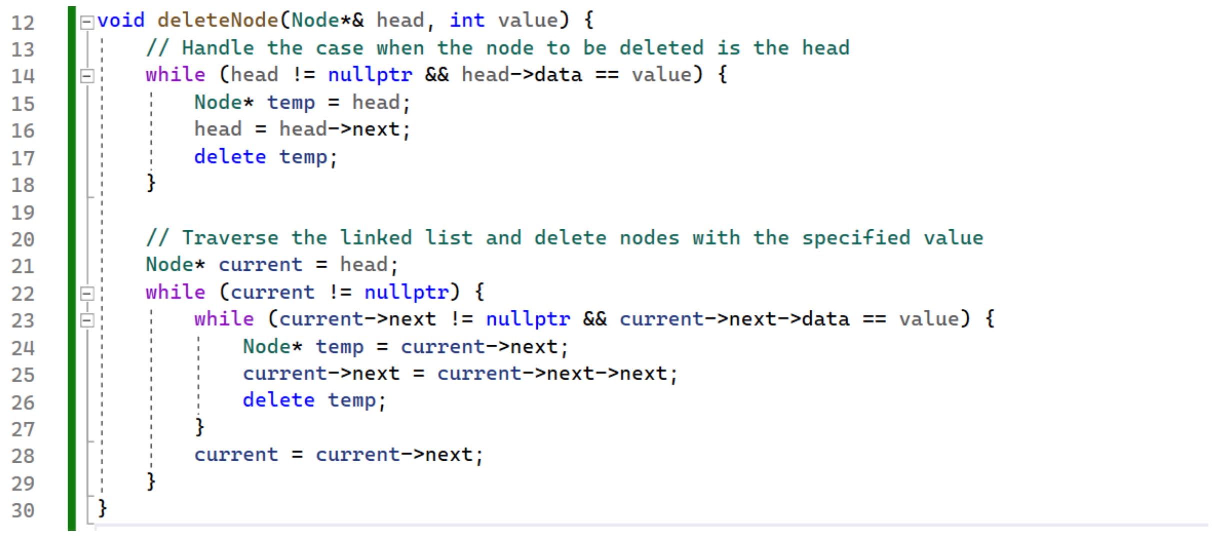


Select Microsoft Visual Studio Debug Console

```
1 -> 2 -> 3 -> nullptr
```

Figure 16(Output)

Delete Node:



```
void deleteNode(Node*& head, int value) {
    // Handle the case when the node to be deleted is the head
    while (head != nullptr && head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    // Traverse the linked list and delete nodes with the specified value
    Node* current = head;
    while (current != nullptr) {
        while (current->next != nullptr && current->next->data == value) {
            Node* temp = current->next;
            current->next = current->next->next;
            delete temp;
        }
        current = current->next;
    }
}
```

Figure 17(Delete Nodes)

Activities

Pre-Lab Activities:

Task 01: Singly LinkedList implementation

Create a linked list with following class definitions.

```
template<class T>
class LinkedList;
template<class T>
class Node
{
public:
	T info;
	Node<T>* next;
	// Methods…
};
template<class T>
class LinkedList
{
	Node<T>* head;

	// Methods…
};
```

Implement following functions for List class.

1. Constructor, destructor, Copy-constructor.

2. void insertAtHead(T value)

3. void insertAtTail(T value)

4. bool deleteAtHead( )

5. bool deleteAtTail( )

6. void printList( )

7. Node* getNode(int n)

This function should return pointer to nth node in the list. Returns last node if n is greater than the number of nodes present in the list.

8. bool insertAfter(T value, T key)

Insert a node after some node whose info equals input key parameter and returns true if node is successfully inserted, false otherwise.

9. bool insertBefore(T value, T key)

Insert a node before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise.

10. bool deleteBefore(T key)

Delete a node that is before some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or first node in the list.

11. bool deleteAfter(T value)

Delete a node that is after some node whose info equals input parameter key and returns true if node is successfully inserted, false otherwise. Check boundary cases i.e if node to be deleted is last node or first node in the list.

12. int getLength( ) returns the total number of nodes in the list.

13. Node* search(T x)

Search a node with value “x” from list and return its link. If multiple nodes of same value exist, then return pointer to first node having the value “x”.

Task 02: Linked Queue

We studied how to represent queues using sequential organization. Such a representation is efficient if we have a circular queue of fixed size. However, there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major drawback is the updating of front and rear. For correctness of the said implementation, the shifting of the queue to the left is necessary and to be done frequently. Here is a good solution to this problem which is used by the linked list.

Implement following linked Queue.

```
template <class T>
class Queue;
template<class T>
class QNode
{
    friend Queue<T>;
    T data;
    QNode* link;
public:
    // methods
};
template<class T>
class Queue
{
    QNode* front, * fear;
    int size;
public:
    Queue()
    {
        front = fear = Null;
    }
    Queue(const Queue<T>&);
    void enqueue(int element);
    T dequeue();
    int currentSize();
    int frontElement();
    bool isEmpty();
    bool isFull();
    void display();
    ~Queue();
};
```

Task 03: Linked Stack

You have already implemented an array-based stack that needs resizing after its array is full. But there is no need to be resized in the linked stack. Implement a linked Stack as provided with the template classes.

```
// forward declaration of template class Stack
template <class T>
class Stack;
template<class T>
class StackNode
{

    T data;
    StackNode* link;
    friend Stack<T>
public:
    // methods
};
template<class T>
class Stack
{
private:
    StackNode* Top;
    int Size;

public:
    Stack()
    {
        Top = nullptr;
        Size = 0;
    }
    ~Stack();
    Stack(const Stack<T>&); // copy constructor
    int getTop();
    int pop();
    void push(int Element);
    int currSize();
    bool isEmpty();
    bool isFull();
    T Peek(int nodeNumber);
};
```

In-Lab Activities

Task 01: Merge Two Sorted Linked Lists

Problem Background: Merging two sorted linked lists is a common problem in computer science, often used in algorithms that require combining data from multiple sources while maintaining order. The goal is to create a new sorted linked list by merging two input linked lists that are already sorted. This problem is a great exercise in pointer manipulation and understanding linked list operations.

Problem Statement:

You are required to implement the following function:
Node* mergeSorted(Node* head1, Node* head2);

This function should return merged sorted linked list

Test Case 1:

- Input:
```
      List1: 0 -> 2 -> 4
      List2: 1 -> 3 -> 5
```
- Expected Output:
```
       0 -> 1 -> 2 -> 3 -> 4 -> 5
```

Task 02: Help City Council

Problem Statement:

You are part of a smart city infrastructure team. The city already has two circular train routes. You are tasked with enhancing the transportation system by merging two circular train routes into one continuous loop. After merging, passengers should be able to travel seamlessly from any station in the first route through all stations in the second route and then back to the start of the first route, maintaining a continuous circular structure.

Your goal is to develop a program that merges the two circular train routes into one seamless circular route. This merged route should maintain the circular property, allowing passengers to travel from the last station of the second route back to the first station of the first route.

Test Case 1:

- `Input:`
  Route 1: `[10, 20, 30]` (Circular: `10 -> 20 -> 30 -> 10`)
  Route 2: `[40, 50, 60]` (Circular: `40 -> 50 -> 60 -> 40`)
- `Expected Output:`
  Merged Circular Route: `10 -> 20 -> 30 -> 40 -> 50 -> 60 -> 10`

Test Case 2:

- `Input:`
  Route 1: `[10, 20]` (Circular: `10 -> 20 -> 10`)
  Route 2: `[]` (No stations on the second route)
- `Expected Output:`
  Merged Circular Route: `10 -> 20 -> 10`

Task 03: Linked List Cycle

Problem Statement:

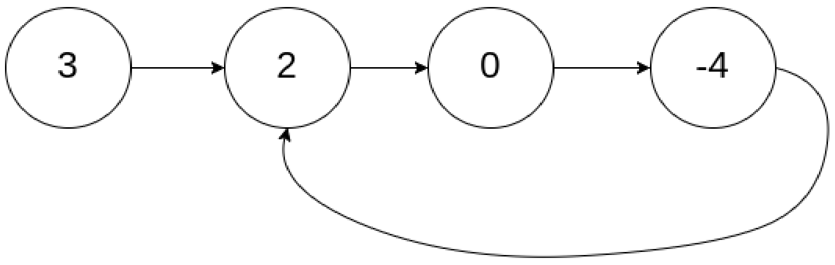
Given a linked list, determine if it has a cycle in it. If there is a cycle, return the node where the cycle begins. If there is no cycle, return null.

A cycle in a linked list occurs when a node's next pointer points back to one of the previous nodes in the list, creating a closed loop. This means that if you traverse the list starting from the head, you will eventually revisit some nodes, rather than reaching the end of the list (which would be null).

```
Node* FindCycleStart (Node* head) [
        // Your implementation here
}
```

Test Case 1:

- `Input:`
  `List: 3 -> 2 -> 0 -> -4 -> 2`
  This creates a cycle where the tail node points back to the second node.
- `Expected Output:`
  `Node with value: 2`



Task 04: Delete a Given Node from a Singly Linked List

Given a node from singly Linked List Your task is to implement a function that deletes a specified node from the singly linked list. The function will modify the linked list in place, ensuring that the node is removed without directly accessing the head pointer.

Function Signature:

```
void deleteNode (Node* node);
```

→ It is guaranteed to be a valid node in the linked list and will not be the tail node (i.e., it has a subsequent node).

Test Case 1:

- `Input:`
  List: `1 -> 2 -> 3 -> 4 -> 5`
  Node to Delete: Pointer to node with value 3
- `Expected Output:`
  ```
  1 -> 2 -> 4 -> 5
  ```

Bonus:

- Solve the problem using constant space
- Can you do it in Constant Time?

Task 05: Reverse a Singly Linked List

Given Head of a Singly Linked List, your task is to implement a function that reverses a singly linked list. The function should modify the linked list in place, so that the head of the list becomes the tail, and all nodes are reversed in order.

Function Signature:

```
Node* reverseList(Node* head);
```

Test Case:

- `Input:`
  ```
  List: 1 -> 2 -> 3 -> 4 -> 5
  ```
- `Expected Output:`
  ```
  Resulting List: 5 -> 4 -> 3 -> 2 -> 1
  ```

Constraints:

- You are not allowed to use any extra space

Post-Lab Activities

Task 01: Reverse Linked Lists in K Groups

Problem Statement:

Given a singly linked list, reverse the **K** nodes of the list at a time and return the modified list. The reversal should occur in **contiguous groups of k nodes**, maintaining the original relative order of nodes outside these groups. If the number of nodes remaining is less than k, leave those nodes as they are, without reversing them.

The function should modify the list in place, without using extra space for a new list.

Function Signature:

```
Node* reverseKGroup (Node* head, int k) {
        //Your Implementation here
}
```

Input: A Singly Linked List, and K i.e. number of nodes in each group.
Output: Reversed Linear Singly Linked List

Test Case 1:

- Input:
  ```
  List: 1 -> 2 -> 3 -> 4 -> 5      k = 2
  ```
- Expected Output:
  ```
  Reversed List: 2 -> 1 -> 4 -> 3 -> 5
  ```

Test Case 2:

- Input:
  ```
  List: 1 -> 2 -> 3 -> 4 -> 5      k = 3
  ```
- Expected Output:
  ```
  Reversed List: 3 -> 2 -> 1 -> 4 -> 5
  ```

Task 02: Round-Robin Scheduling

Problem Statement:

In operating systems, Round-Robin (RR) Scheduling is a CPU scheduling algorithm that allocates each process an equal time slice, known as a quantum, ensuring fair access to CPU time. Each process gets CPU time in a round-robin manner. Once a process uses up its quantum, if it still needs more time to complete, it is moved to the back of the queue. This cycle continues until all processes finish their execution.

In this task, you are required to implement a Round-Robin scheduler using a circular linked list, representing the queue of processes. Each process has a burst time (the total time required for it to complete). The CPU will allocate a fixed time slice to each process on each turn. The program will simulate this execution and switch between processes in a circular manner until all processes have been executed.

Implementation Requirements:

1. **Circular Linked List Representation**:
    - Create a circular linked list where each node represents a process in the ready queue.
    - Each node contains:
        - **Process ID**: A unique identifier for the process (e.g., P1, P2, etc.).
        - **Burst Time**: The remaining time the process needs to complete its execution.
2. **Round-Robin Scheduling Process**:
    - Each process is given a fixed CPU time slice (quantum) per round:
        - If the process's remaining burst time is greater than the quantum, subtract the quantum from the burst time, and move the process to the back of the queue.
        - If the remaining burst time is less than or equal to the quantum, use the remaining burst time to complete the process, mark it as completed, and remove it from the queue.
    - Repeat this for all processes until the queue is empty.
    - After each round, display the sequence in which processes execute, along with their remaining burst times. Once a process completes, display that it has been removed from the queue.

Constraints:

- The circular linked list should correctly simulate the queue, providing each process an equal time slice in a round-robin manner.
- Processes execute in the same initial order, and you should not skip any process until it completes.
- The burst time and time slice are non-negative integers.

Input:

1. The number of processes and each process's burst time.
2. A time slice (quantum) to allocate to each process.

Output:

1. Display the round number and, within each round, show:
    - The process being executed.
    - The time the process uses in that round.
    - The remaining burst time after each execution.
2. Indicate when a process has completed and is removed from the queue.

Example Execution:

Input:

Total Number of Processes: **2**
Enter the burst time for P1: **10**
Enter the burst time for P2: **5**

Enter the burst time for P3: **8**
Enter the CPU time slice (Quantum): **4**

Expected Output:

```
Round 1:
Process P1 executes for 4 units.          Remaining time: 6
Process P2 executes for 4 units.          Remaining time: 1
Process P3 executes for 4 units.          Remaining time: 4

Round 2:
Process P1 executes for 4 units.          Remaining time: 2
Process P2 executes for 1 unit.           Process completed.
Process P3 executes for 4 units.          Process completed.

Round 3:
Process P1 executes for 2 units.          Process completed.
```

Explanation:

1. **Initial Setup**:
   - The program takes the burst times for each process (P1: 10, P2: 5, P3: 8) and the quantum of 4 units.
2. **First Round**:
   - **P1** has 10 units of burst time and executes for 4 units, leaving it with 6 units.
   - **P2** has 5 units and executes for 4 units, leaving it with 1 unit.
   - **P3** has 8 units and executes for 4 units, leaving it with 4 units.
3. **Second Round**:
   - **P1** executes for another 4 units, reducing its burst time to 2 units.
   - **P2** has 1 unit left, so it uses 1 unit and completes its execution.
   - **P3** executes for its remaining 4 units and completes.
4. **Third Round**:
   - Only **P1** remains with 2 units of burst time, which it uses to complete its execution.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's

Division of Pre-Lab marks:	**[25 marks]**
Task 01: Singly LinkedList	[10 marks]
Task 02: Linked Queue	[08 marks]
Task 03: Linked Stack	[07 marks]
Division of In-Lab marks:	**[50 marks]**
Task 01: Merge Sorted Linked Lists	[10 marks]
Task 02: Help City Council	[10 marks]
Task 03: Linked List Cycle	[10 marks]
Task 04: Delete given Node	[10 marks]
Task 05: Reverse Linked List	[10 marks]
Division of Post-Lab marks:	**[25 marks]**
Task 01: Reverse Linked List in nodes of K groups	[10 marks]
Task 02: Round Robin Simulations	[15 marks]

References and Additional Material:

Singly LinkedList	https://www.geeksforgeeks.org/what-is-linked-list/
Circular Singly Linked List	https://www.javatpoint.com/circular-singly-linked-list
Round Robin Scheduling Algo	https://en.wikipedia.org/wiki/Round-robin_scheduling

Lab Time Activity Simulation Log:

Slot – 01 – 02:15 – 02:30:	Class Settlement
Slot – 02 – 02:30 – 03:00:	In-Lab Task 01
Slot – 03 – 03:00 – 03:45:	In-Lab Task 02
Slot – 04 – 03:45 – 04:30:	In-Lab Task 03
Slot – 05 – 04:30 – 05:00:	In-Lab Task 04
Slot – 06 – 05:00 – 05:30:	In-Lab Task 05
Slot – 07 – 05:30 – 06:00:	Evaluation