Singly Linked Lists in C++

Using a Template Node Structure and Solving Common Problems

1. Introduction

In this lecture, we'll explore singly linked lists in C++ without wrapping them in a class. Instead, we'll work directly with:

- A templated Node structure
- Free functions to manipulate the list
- Raw pointers and manual memory management

This approach helps you:

- Understand the core mechanics of linked lists
- Practice pointer manipulation
- Solve classic interview-style problems

We'll also use templates so our list can store any data type (int, string, double, etc.).

2. What Is a Singly Linked List?

A singly linked list is a linear collection of elements called nodes.

Each node contains:

- Data (e.g., an integer, string, or object)
- A pointer to the next node in the sequence

The last node

[Data | Next] → [Data | Next] → [Data | Next] → nullptr

↑

Head (points to first node)

points to `nullptr`, marking the end of the list.

🔑 Key idea: No random access — you must traverse from the head to reach any node.

2. The Node Structure (Templated)

We define a generic Node that can hold any type of data:

```
template <typename T>
struct Node {
    T data;
    Node<T>* next;
    // Constructor
    Node(const T& value) : data(value),
next(nullptr) {}
};
```

✅ Node<int>, Node<std::string>, etc., will be generated automatically by the compiler.

3. Basic Operations as Free Functions

Since we're not using a class, the head pointer is managed by the user and passed to functions by reference (so we can modify it).

3.1. Insert at Front

```
template <typename T>
void push_front(Node<T>*& head, const T& value) {
    Node<T>* newNode = new Node<T>(value);
    newNode->next = head;
    head = newNode;
}
```

🔑 Note: `Node<T>*& head` — we pass the pointer by reference to update the original head.

3. Basic Operations as Free Functions

3.2. Print the List

```
template <typename T>
void print_list(const Node<T>* head) {
    while (head) {
        std::cout << current->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr\n";
}
```

🔒 const ensures we don’t accidentally modify the list.

3. Basic Operations as Free Functions

3.3. Delete the Entire List

```
template <typename T>
void delete_list(Node<T>*& head) {
    while (head != nullptr) {
        Node<T>* temp = head;
        head = head->next;
        delete temp;
    }
    // head is now nullptr
}
```

✅ Always clean up to avoid memory leaks.

4. Example: Basic Usage

```
#include <iostream>
#include <string>

int main() {
    Node<int>* head = nullptr;  // Start with empty list
    push_front(head, 30);
    push_front(head, 20);
    push_front(head, 10);
    print_list(head);  // Output: 10 → 20 → 30 → nullptr
    delete_list(head);  // Clean up
    std::cout << "List deleted. head is "
              << (head ? "not null" : "nullptr") << std::endl;
    return 0;
}
```

5. Solving Typical Problems

Let's implement solutions to common linked list problems using our template-based approach.

Problem 1: Find the Length of the List

Goal: Return the number of nodes.

```
template <typename T>
int get_length(const Node<T>* head) {
    int count = 0;
    while (head != nullptr) {
        count++;
        head = head->next;
    }
    return count;
}
```

Usage:

```
std::cout << "Length: " << get_length(head) << std::endl; // e.g., 3
```

Problem 2: Search for a Value

Goal: Return true if value exists.

```
template <typename T>
bool search(const Node<T>* head, const T& value) {
    while (head != nullptr) {
        if (head->data == value) {
            return true;
        }
        head = head->next;
    }
    return false;
}
```

Usage:

```
if (search(head, 20)) { std::cout << "Found 20!\n"; }
```

Problem 3: Insert at the End

Goal: Add a new node at the tail.

```
template <typename T>
void push_back(Node<T>*& head, const T& value) {
    Node<T>* newNode = new Node<T>(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node<T>* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
}
```

 Time complexity: O(n)

Problem 4: Delete the First Occurrence of a Value

Goal: Remove one node with given value (if exists).

```
template <typename T>
void delete_value(Node<T>*& head, const T& value) {
    // Case 1: empty list
    if (head == nullptr) return;
    // Case 2: delete head
    if (head->data == value) {
        Node<T>* temp = head;
        head = head->next;
        delete temp;
        return;
    }
    // Case 3: delete in the middle or end
    Node<T>* current = head;
    while (current->next != nullptr && current->next->data != value) current = current->next;
    if (current->next != nullptr) { // Found it
        Node<T>* toDelete = current->next;
        current->next = toDelete->next;
        delete toDelete;
    }
}
```

Problem 5: Reverse the List (In-Place)

Goal: Reverse the direction of pointers.

```
template <typename T>
void reverse_list(Node<T>*& head) {
    Node<T>* prev = nullptr;
    Node<T>* current = head;
    Node<T>* next = nullptr;
    while (current != nullptr) {
        next = current->next;   // Save next
        current->next = prev;   // Reverse link
        prev = current;         // Move prev forward
        current = next;         // Move current forward
    }
    head = prev;  // New head is the last node
}
```

✅ This is a classic interview question!

Problem 5: Reverse the List (In-Place)

```
template <typename T>
Node<T>* reverseListRecursive(Node<T>* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;  // Base case: empty or single node
    }
    Node<T>* newHead = reverseListRecursive(head->next);
    head->next->next = head;  // Reverse the link
    head->next = nullptr;     // Break original link
    return newHead;
}
```

7. Key Takeaways

CONCEPT	WHY IT MATTERS
Templated Node	Reusable for any data type
Pass head by reference	Allows functions to change which node is the head
Manual memory management	You must `delete` every `new`
Pointer traversal	Core skill for linked structures
Edge cases	Always test: empty list, single node, head/tail operations

8. Common Pitfalls

❌ Forgetting to initialize head to `nullptr`
❌ Passing head by value → changes don’t persist
❌ Not handling empty list in deletion/search
❌ Memory leaks from missing delete
✅ Always test with:

- Empty list
- One element
- Two elements
- Operation on head/tail

10. Practice Problems

- Write a function to find the middle node (use slow/fast pointers).
- Check if the list is a palindrome.
- Detect a cycle in the list (Floyd’s cycle-finding algorithm).
- Merge two sorted lists into one sorted list.

 These are common in technical interviews!

Write a function to find the middle node (use slow/fast pointers).

```
template <typename T>
Node<T>* find_middle(Node<T>* head) {
    if (head == nullptr) {
        return nullptr;  // Empty list
    }
    Node<T>* slow = head;
    Node<T>* fast = head;
    // Move fast two steps and slow one step at a time
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    // When fast reaches the end, slow is at the middle
    return slow;
}
```

Detect a cycle in the list (Floyd’s cycle-finding algorithm).

Below is a clear and efficient implementation of Floyd’s Cycle-Finding Algorithm (also known as the Tortoise and Hare algorithm) to detect a cycle in a singly linked list.
This algorithm uses two pointers moving at different speeds:
Slow pointer (tortoise) moves 1 step at a time.
Fast pointer (hare) moves 2 steps at a time.
👉 If there is a cycle, the fast pointer will eventually catch up to the slow pointer.
👉 If there is no cycle, the fast pointer will reach nullptr.

```
template <typename T>
bool has_cycle(Node<T>* head) {
    if (head == nullptr) {
        return false; // Empty list has no cycle
    }
    Node<T>* slow = head;
    Node<T>* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;           // Move 1 step
        fast = fast->next->next;     // Move 2 steps
        if (slow == fast) {
            return true; // Cycle detected!
        }
    }
    return false; // Reached end → no cycle
}
```

Detect a cycle in the list (Floyd’s cycle-finding algorithm).

🔍 How It Works

No cycle:

fast reaches the end (nullptr) → loop exits → return false.

Cycle exists:

Both pointers enter the loop. Because fast moves faster, it will eventually lap slow, and they will meet at some node inside the cycle.

🧠 Why must they meet?

In a cycle of length L, the distance between fast and slow decreases by 1 each step. So they must meet within L iterations.

Detect a cycle in the list (Floyd’s cycle-finding algorithm).

🔁 Bonus: Find the Start of the Cycle (Optional Extension)

If you also need to find the node where the cycle begins, you can extend Floyd’s algorithm:

```
template <typename T>
Node<T>* find_cycle_start(Node<T>* head) {
    Node<T>* slow = head;
    Node<T>* fast = head;
    // Step 1: Detect if cycle exists
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) break;
    }
```

Detect a cycle in the list (Floyd's cycle-finding algorithm).

```
    if (fast == nullptr || fast->next == nullptr) {
        return nullptr; // No cycle
    }
    // Step 2: Find the start of the cycle
    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }
    return slow; // Start of the cycle
}
```

📌 Why this works:

When the two pointers meet, resetting one to the head and moving both at the same speed makes them meet at the cycle's entrance.

Detect a cycle in the list (Floyd’s cycle-finding algorithm).

⚠ Important Note on Memory

If a list contains a cycle, you cannot safely delete all nodes using a standard loop (it will infinite-loop). Always detect and break the cycle before cleanup in real applications.

This algorithm is a classic interview question and a beautiful example of pointer manipulation in C++!

🔍 How It Works

slow moves 1 node per iteration.
fast moves 2 nodes per iteration.
When fast reaches the end (nullptr or last node), slow will be at the middle.
Examples:
List: 1 → 2 → 3 → 4 → 5 → middle = 3 (3rd node)
List: 1 → 2 → 3 → 4 → middle = 3 (2nd of the two middles — second middle in even-length lists)
 This version returns the second middle in even-length lists (common convention, used in LeetCode problems).
If you want the first middle in even-length lists, change the loop condition to:

```
while (fast->next != nullptr && fast->next->next != nullptr)
```

Merge two sorted lists into one sorted list.

```
template <typename T>
Node<T>* merge_sorted(Node<T>* list1, Node<T>* list2) {
    // Handle empty lists
    if (!list1) return list2; if (!list2) return list1;
    // Create a dummy head to simplify logic
    Node<T> dummy(0); // value doesn't matter
    Node<T>* tail = &dummy;
    // Traverse both lists and link the smaller node
    while (list1 && list2) {
        if (list1->data <= list2->data) {
            tail->next = list1;
            list1 = list1->next;
        } else {
            tail->next = list2;
            list2 = list2->next;
        }
        tail = tail->next;
    }
    // Attach the remaining part (one of the lists is exhausted)
    tail->next = list1 ? list1 : list2;
    return dummy.next; // skip dummy node
}
```

🔑 Key idea: Use a dummy node to avoid handling the head insertion as a special case.

Merge two sorted lists into one sorted list.

✅ Option 2: Create a New List (Does Not Modify Inputs)

If you need to preserve the original lists, you can create new nodes:

```
template <typename T>
Node<T>* merge_sorted_copy(Node<T>* list1, Node<T>* list2) {
    if (!list1) return deep_copy(list2);
    if (!list2) return deep_copy(list1);
    Node<T>* dummy = new Node<T>(0);
    Node<T>* tail = dummy;
    while (list1 && list2) {
        if (list1->data <= list2->data) {
            tail->next = new Node<T>(list1->data);
            list1 = list1->next;
        } else {
            tail->next = new Node<T>(list2->data);
            list2 = list2->next;
        }
        tail = tail->next;
    }
```

Merge two sorted lists into one sorted list.

```
// Copy remaining
    while (list1) {
        tail->next = new Node<T>(list1->data);
        tail = tail->next;
        list1 = list1->next;
    }    while (list2) {
        tail->next = new Node<T>(list2->data);
        tail = tail->next;
        list2 = list2->next;
    }
    Node<T>* result = dummy->next;
    delete dummy; // clean up dummy
    return result;
}

// Helper: deep copy a list
template <typename T>
Node<T>* deep_copy(Node<T>* head) {
    if (!head) return nullptr;
    Node<T>* new_head = new Node<T>(head->data);
    Node<T>* current = new_head;
    head = head->next;
    while (head) {
        current->next = new Node<T>(head->data);
        current = current->next;
        head = head->next;
    }
    return new_head;
}
```

Merge two sorted lists into one sorted list.

 Notes

The function works for any comparable type T (as long as operator<= is defined).

If one list is empty, it returns the other — safe and correct.

The in-place version is preferred in interviews and real code unless you must keep originals intact.

This pattern is foundational and appears in many algorithms (e.g., merge sort for linked lists). Master it!

9. What’s Next?

Once comfortable with raw pointers and free functions, you can:

- Wrap everything in a LinkedList<T> class
- Add iterators
- Implement copy constructor and assignment operator
- Compare performance with std::forward_list

Recursion in C++ — Theory, Examples, and Applications with Dynamic Data Structures

1. Introduction to Recursion

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems.

Key Components of a Recursive Function:

- Base Case: A condition that stops the recursion (prevents infinite loops).
- Recursive Case: The function calls itself with a modified argument, moving toward the base case.
- Recursion is elegant, mathematically intuitive, and often mirrors the natural structure of data — especially dynamic data structures like trees and linked lists.

2. Basic Example: Factorial

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

int factorial(int n) {
    if (n == 0 || n == 1) {  // Base case
        return 1;
    }
    return n * factorial(n - 1);  // Recursive case
}
int main() {
    cout << "5! = " << factorial(5) << endl; // Output: 120
    return 0;
}
```

Why recursion?

Factorial is naturally defined recursively: $n! = n \times (n-1)!$ with $0! = 1$

3. Recursion with Dynamic Data Structures

```
// Recursive function to print all nodes
template<class T>
void printListRecursive(Node<T>* head) {
    if (head == nullptr) {  // Base case: end of list
        return;
    }
    cout << head->data << " -> ";
    printListRecursive(head->next);  // Recursive call on remainder
}

// Recursive function to calculate sum of all nodes
template<class T>
T sumListRecursive(Node<T>* head) {
    if (head == nullptr) {
        return 0;
    }
    return head->data + sumListRecursive(head->next);
}
```

Advantage: Recursive traversal is clean and mirrors the structure.

⚠ Caution: Deep recursion may cause stack overflow on very long lists.

3. Recursion with Dynamic Data Structures

3.2. Binary Tree Traversals

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

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// Recursive Inorder Traversal: Left -> Root -> Right
void inorder(TreeNode* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->val << " ";
    inorder(root->right);
}
```

3. Recursion with Dynamic Data Structures

```
// Recursive Preorder: Root -> Left -> Right
void preorder(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorder(root->left);
    preorder(root->right);
}

// Recursive Postorder: Left -> Right -> Root
void postorder(TreeNode* root) {
    if (root == nullptr) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->val << " ";
}
```

3. Recursion with Dynamic Data Structures

```
// Recursive height calculation
int treeHeight(TreeNode* root) {
    if (root == nullptr) return -1;  // Height of empty tree is -1
    int leftHeight = treeHeight(root->left);
    int rightHeight = treeHeight(root->right);
    return 1 + max(leftHeight, rightHeight);
}

// Recursive count of nodes
int countNodes(TreeNode* root) {
    if (root == nullptr) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}
```

💡 Insight: Tree algorithms are almost always recursive because the structure is self-similar. Iterative versions require explicit stacks — recursion does this implicitly.

4.2. Binary Search (Recursive)

```
int binarySearchRecursive(const vector<int>& arr,
int left, int right, int target) {
    if (left > right) return -1;
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    else if (arr[mid] > target) return
binarySearchRecursive(arr, left, mid - 1, target);
    else return binarySearchRecursive(arr, mid + 1,
right, target);
}
```

4. Introduction to the Fibonacci Sequence

The Fibonacci sequence is defined as:

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

It's a classic example of recursion because each number depends on the sum of the two previous numbers.

5. All Methods to Compute Fibonacci Numbers Recursively

2.1. Naive Recursive Approach

```
#include <iostream>
using namespace std;
long long fibonacci_naive(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_naive(n - 1) + fibonacci_naive(n - 2);
}
int main() {
    cout << "F(10) = " << fibonacci_naive(10) << endl; // Output: 55
    return 0;
}
```

Problem: Exponential time — recalculates same subproblems repeatedly.

5. All Methods to Compute Fibonacci Numbers Recursively

2.2. Recursive Approach with Memoization

```
#include <iostream>
#include <vector>
using namespace std;
vector<long long> memo(100, -1); // Memoization table
long long fibonacci_memo(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) {
        return memo[n]; // Return precomputed result
    }
    memo[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2);
    return memo[n];
}
```

Advantage: Avoids redundant calculations.

5. All Methods to Compute Fibonacci Numbers Recursively

2.4. Tail Recursion (Optimized Recursion)

```
#include <iostream>
using namespace std;
long long fibonacci_tail_recursive(int n, long long a = 0, long long b = 1) {
    if (n == 0) return a;
    if (n == 1) return b;
    return fibonacci_tail_recursive(n - 1, b, a + b);
}
```

Benefit: More efficient than naive recursion.

2.5. Matrix Exponentiation (Recursive Helper)

This method uses the mathematical property:

```
[F(n+1)]   [1 1]^n   [1]
[F(n)  ] = [1 0]   * [0]
```

```
#include <iostream>
using namespace std;
struct Matrix2x2 {
    long long a, b, c, d; // [a b; c d]
    Matrix2x2(long long a = 1, long long b = 1, long long c = 1, long long d = 0)
        : a(a), b(b), c(c), d(d) {}
};
Matrix2x2 multiply(const Matrix2x2& m1, const Matrix2x2& m2) {
    return Matrix2x2(
        m1.a * m2.a + m1.b * m2.c,
        m1.a * m2.b + m1.b * m2.d,
        m1.c * m2.a + m1.d * m2.c,
        m1.c * m2.b + m1.d * m2.d
    );
}
```

2.5. Matrix Exponentiation (Recursive Helper)

```
Matrix2x2 matrix_power(Matrix2x2 mat, int n) {
    if (n == 1) return mat;
    if (n % 2 == 0) {
        Matrix2x2 half = matrix_power(mat, n / 2);
        return multiply(half, half);
    } else {
        return multiply(mat, matrix_power(mat, n - 1));
    }
}
long long fibonacci_matrix(int n) {
    if (n <= 1) return n;
    Matrix2x2 base(1, 1, 1, 0);
    Matrix2x2 result = matrix_power(base, n);
    return result.b; // F(n) is in position [0][1]
}
```

Best for: Very large n.

4. When NOT to Use Recursion

While elegant, recursion has costs:

DRAWBACK	EXPLANATION
Stack Overflow	Deep recursion (e.g., 100,000 levels) may crash the program.
Memory Overhead	Each call adds a frame to the call stack.
Performance	Function call overhead can make recursion slower than iteration.

5. Best Practices for Recursive Programming

✅ Do:

- Always define a base case to stop recursion.
- Ensure the recursive case moves toward the base case.
- Use memoization when subproblems overlap.
- Consider tail recursion for efficiency.
- Be mindful of stack overflow for deep recursion.

❌ Avoid:

- Recalculating the same subproblems (without memoization).
- Using recursion for simple linear problems (e.g., factorial of large n).
- Ignoring memory limits on recursion depth.

7. Summary: Recursive Fibonacci and General Recursion

METHOD	PROS	CONS
Naive Recursive	Simple, intuitive	Exponential time
Memoized Recursive	Efficient, reusable	Extra space for cache
Tail Recursive	More efficient	Still O(n) space
Matrix Exponentiation	Fastest for large n	Complex to implement

🔑 Key Takeaways:

- Recursion is natural for problems with self-similar structure (trees, fractals, divide-and-conquer).
- Fibonacci is a perfect example of how recursion can be inefficient without optimization.
- Memoization turns exponential algorithms into polynomial ones.
- Recursion is foundational for dynamic programming, backtracking, and graph algorithms.

Time Complexity: O(n) – Explained in Detail

Time Complexity: O(n) – Explained in Detail

- What Does O(n) Mean?
- O(n) (pronounced “big O of n”) means that the running time of an algorithm grows linearly with the size of the input n.
- In other words, if you double the input size, the algorithm will take roughly twice as long to run.
- It describes the worst-case (or sometimes average-case) behavior as n becomes very large.