Linked List (C++)

This C++ file is an excellent demonstration of how to implement the **Linked List data structure** using the principles of **Object-Oriented Programming**. It moves beyond simple procedural code by encapsulating the logic for different list types into well-defined classes. This approach provides better organization, data protection, and resource management.

The file covers three fundamental variations of the Linked List:

1. **Singly Linked List:** The standard implementation where each node points forward to the next.
2. **Circular Linked List:** A variation where the last node points back to the first, creating a continuous loop.
3. **Doubly Linked List:** An advanced version where each node points to both the next and the previous node, allowing for efficient two-way traversal.

A key feature of this implementation is the use of **constructors** for easy object creation (e.g., building a list from an array) and **destructors** for automatic memory management. The destructor in each class iterates through the list and deletes each node, which is a crucial practice for preventing memory leaks. This file serves as a perfect example of building robust, reusable data structures in modern C++.

**Linked List and its functions using c++**

This section implements a standard **Singly Linked List** as a C++ class. The data (the first node pointer) is kept private to protect it from accidental outside modification, and all operations are handled through public member functions.

* class Node { ... }; class LinkedList { ... }; This defines the basic building blocks. The Node class holds the data and a next pointer. The LinkedList class **encapsulates** the logic and contains a private pointer, first, which always points to the head of the list.
* LinkedList::LinkedList(int A[], int n) This is a **constructor**. It's a special function that automatically runs when a LinkedList object is created. Its job is to build the entire list by creating new nodes on the heap from the elements of an input array A.
* LinkedList::~LinkedList() This is the **destructor**. It's another special function that automatically runs when the LinkedList object is destroyed (e.g., when main finishes). Its crucial role is to **prevent memory leaks** by iterating through the list and freeing the memory allocated for each node.
* void LinkedList::Insert(int index, int x) This method inserts a new node. The logic handles two cases: inserting at the beginning (index 0), which involves updating the first pointer, and inserting in the middle, which requires looping to the correct position before rewiring the pointers.
* int LinkedList::Delete(int index) This method removes a node. To delete a middle node, it uses two pointers, p and q, where q is a "tailing pointer" that stays one step behind p. This allows q to update its next pointer to bypass p, effectively removing p from the list.

**Circular Linked List**

This section implements a **Circular Linked List**. The main difference is that the last node’s next pointer doesn't point to NULL; it points back to the head node, forming a loop.

* t->next = tail->next; tail->next = t; tail = t; This is the clever logic inside the constructor for adding a new node t to the end of the list. Since tail->next always points to the head, this first makes the new node's next point to the head. Then, it updates the old tail to point to the new node, making it the new end of the list.
* do { ... } while (p != head); Displaying a circular list requires a do-while loop. A standard while (p != head) loop would fail because the condition is immediately true at the start. The do-while ensures the code inside the loop runs **at least once** before checking the condition, correctly printing the head node.
* static int flag = 0; if (p != head || flag == 0){ ... } The recursive display function needs a static int flag to work correctly. This flag ensures the function enters the block on its very first call (when p == head but flag == 0) and is then set to 1. For all subsequent calls, the recursion continues until p loops back to head, at which point the condition p != head is false, and the recursion stops.

**Inserting, Deleting and Reversing a Doubly linked list**

This section implements a **Doubly Linked List**. Each node here has two pointers: next (pointing to the following node) and prev (pointing to the preceding node). This structure allows for traversal in both directions.

* Node\* prev; int data; Node\* next; This defines the structure for a doubly linked node. The prev pointer is the key addition that enables bidirectional capabilities.
* t->prev = p; t->next = p->next; if (p->next){ p->next->prev = t; } p->next = t; This shows the four crucial pointer updates needed to insert a new node t after a node p. It correctly establishes links in **both directions** between t and its surrounding nodes, ensuring the list's integrity.
* p->prev->next = p->next; if (p->next){ p->next->prev = p->prev; } Deleting a node p is more efficient in a doubly linked list. The p->prev pointer gives us direct access to the preceding node, so we can update its next pointer to bypass p. We then check if p->next exists before updating its prev pointer.
* temp = p->next; p->next = p->prev; p->prev = temp; Reversing a doubly linked list is remarkably elegant. This code iterates through the list, and for each node, it simply **swaps the next and prev pointers**. This single operation, repeated for every node, effectively reverses the entire list's direction.