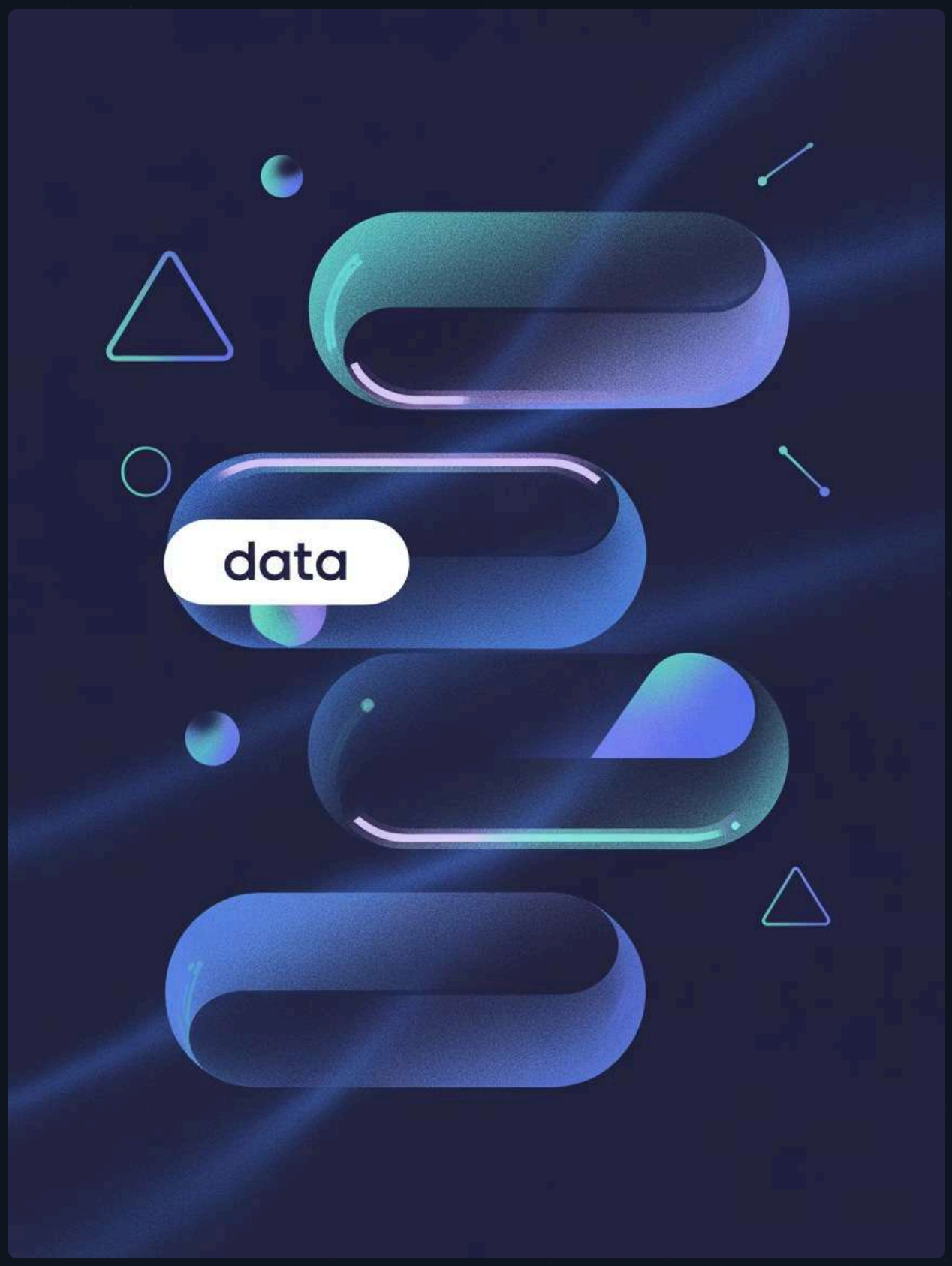
Linked Lists in C++

Dynamic data structures for efficient handling of variable-sized data



What are Linked Lists?

Arrays have a fixed size and expensive insertion/deletion operations. A linked list is a dynamic data structure consisting of a set of nodes that are linked to each other by pointers.

Data

Each node stores useful information — numbers, strings, or any other data types

Link

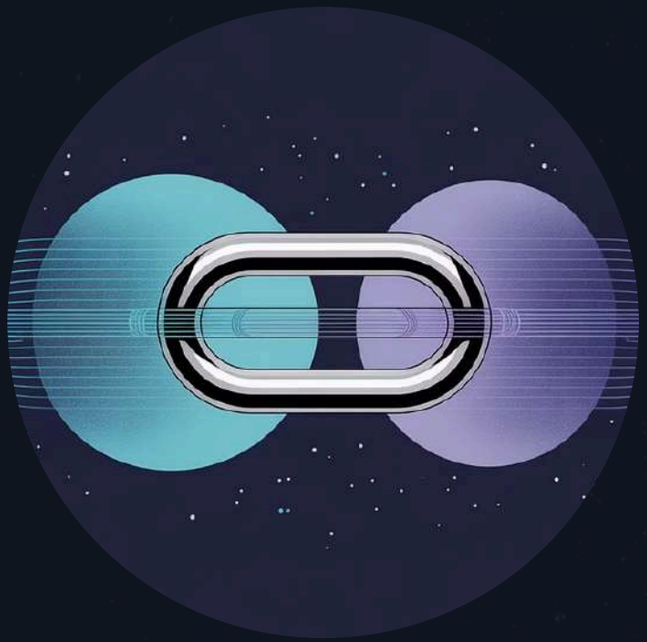
A pointer to the next element in the list, creating a connection between nodes

End of List

The last node points to nullptr, indicating the end of the structure

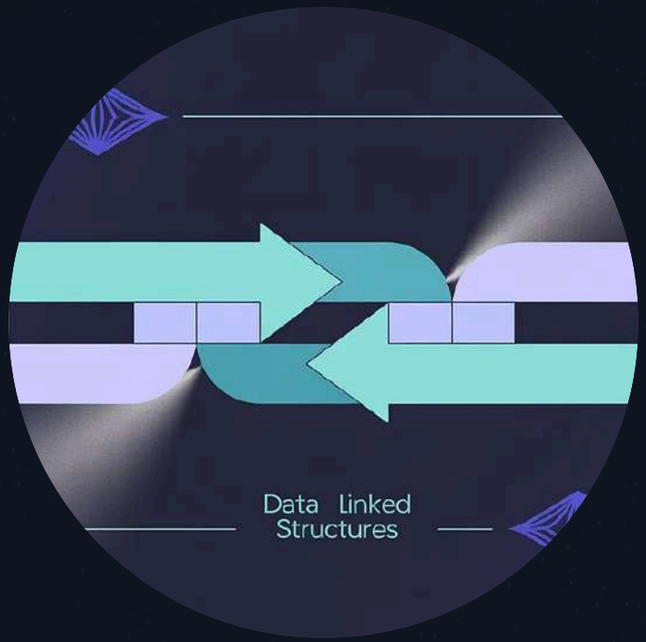
> **Key Features:** Size grows and shrinks dynamically, easy insertion and deletion of elements, sequential data access

Types of Linked Lists



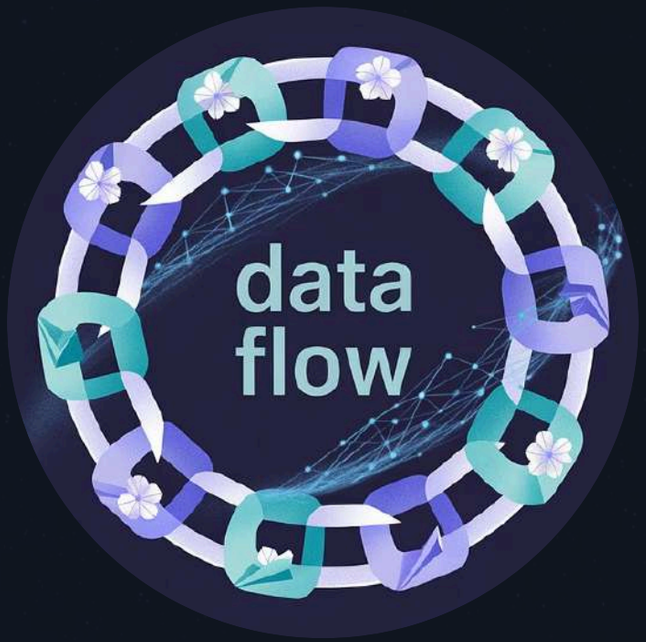
Singly Linked List

Each node contains a pointer only to the next element. A simple and efficient structure for basic operations.



Doubly Linked List

Nodes store references to both the next AND previous elements. Allows bidirectional traversal of the list.



Circular Linked List

The last node points back to the first, forming a closed structure without an explicit end.

Comparison with Arrays

Characteristic	Array	Linked List
Index Access	O(1)	O(n)
Insertion at Beginning	O(n)	O(1)
Insertion in Middle	O(n)	O(n), but simpler
Memory Usage	Contiguous	Dispersed

Node Implementation in C++

Node Structure

The foundation of any linked list is the node — a simple structure containing data and a pointer to the next element.

```
struct Node {
    int data;       // Данные узла
    Node* next;     // Указатель на следующий узел
};
```

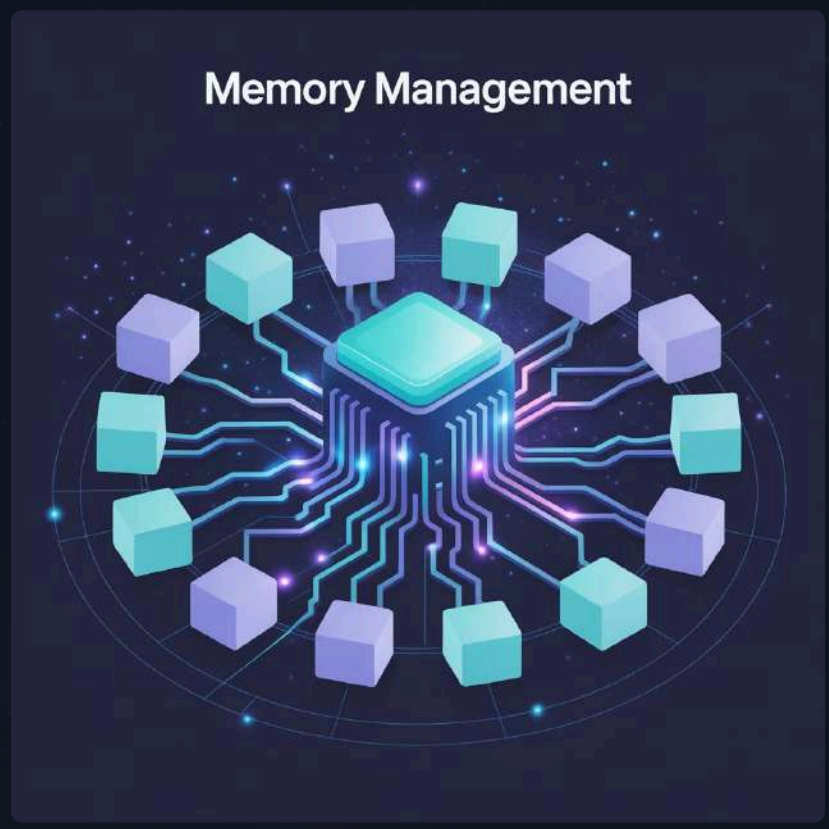
Creating a List

Let's create a simple list of three elements and traverse it:

```
// Создание узлов
Node* head = new Node{1, nullptr};
head->next = new Node{2, nullptr};
head->next->next = new Node{3, nullptr};

// Обход списка
Node* temp = head;
while (temp != nullptr) {
    cout << temp->data << " ";
    temp = temp->next;
}
```

Important! In real programs, always check pointers for nullptr before use.



Key Operations

01

Add to Front (pushFront)

Create a new node and make it the new head of the list

02

Add to End (pushBack)

Traverse to the end of the list and attach the new node

03

Remove from Front (popFront)

Save a reference to the second element and delete the first

04

Search and Delete by Value

Find the desired element and correctly rearrange the links

Insert at Front

```
void pushFront(Node*& head, int
value) {
    Node* newNode = new
Node{value, head};
    head = newNode;
}
```

Remove from Front

```
void popFront(Node*& head) {
 if (!head) return;
 Node* temp = head;
 head = head->next;
 delete temp;
}
```

Doubly Linked Lists

Doubly linked lists extend the capabilities of ordinary lists by adding a pointer to the previous element. This simplifies many operations and allows for efficient traversal of the list in both directions.

1

Node Structure

Contains data and two pointers — to the next and previous elements

2

Bidirectional Traversal

Can move both forwards and backwards through the list without additional cost

3

Simplified Deletion

Deleting an element does not require searching for the previous node

```
struct DNode {
    int data;          // Данные узла
    DNode* next;       // Указатель на следующий узел
    DNode* prev;       // Указатель на предыдущий узел
};
```

Compromise: Doubly linked lists use more memory (additional pointer) but provide greater flexibility in operations

Application and Conclusion

Dynamic Efficiency

Linked lists are ideal for situations where data size is unknown in advance and often changes. They efficiently use memory.

Fast Operations

Insertion and deletion at the beginning of the list are performed in O(1), making them an excellent choice for stacks and queues.

Standard Library

In real C++ projects, use `std::list` — a ready-made, optimized implementation of a doubly linked list.

> **Key Takeaway:** Linked lists are a fundamental data structure that trades random access speed for the flexibility of dynamic resizing.

Understanding the principles of linked lists will help you better comprehend more complex data structures and make informed decisions when choosing the appropriate container for your tasks.