Notes2(Doubly_LL).txt
Document

Beautify my notes , dont delete any points if you are in ambiguity how to present it , simple copy paste and some additional important points as well

Here are your notes, organized and formatted for better readability. Each section is clearly separated, and additional points and code comments have been added for clarification where necessary.

Understanding Member Access Operators in C++

In C++, `.` (dot) and `->` (arrow) are operators used to access members (attributes or methods) of objects. They are used in different contexts depending on how the object or structure is stored or referenced. Here's a detailed explanation:

The `.` Operator

- **Context**: Used with an object or structure when you have a direct instance of that object or structure.
- **Usage**: `object.member` or `struct.member`
- **Example**:

```cpp
struct Point {
    int x, y;
};

Point p;
p.x = 10; // Accessing and setting member 'x'
p.y = 20; // Accessing and setting member 'y'
```

- **Explanation**: In this example, `p` is a direct instance of `Point`. To access the members `x` and `y`, you use the `.` operator.

The `->` Operator

- **Context**: Used with a pointer to an object or structure.
- **Usage**: `pointer->member`
- **Example**:

```cpp
struct Point {
    int x, y;
};

Point* ptr = new Point;
ptr->x = 10; // Accessing and setting member 'x' through a pointer
ptr->y = 20; // Accessing and setting member 'y' through a pointer

delete ptr; // Free allocated memory
```

- **Explanation**: In this case, `ptr` is a pointer to a `Point` object. The `->` operator is used to access the members `x` and `y` through this pointer.

Differences Between `.` and `->`

1. **Type of Access**:
   - `.` is used for accessing members of an object or struct when you have the actual instance.
   - `->` is used for accessing members of an object or struct through a pointer to that instance.
2. **Syntax and Usability**:
   - With `.` you directly access the member from the instance.
   - With `->`, the compiler automatically dereferences the pointer to access the member. It is equivalent to `(*pointer).member`.
3. **Memory Management**:
   - Instances accessed with `.` are typically allocated on the stack unless declared otherwise.
   - Instances accessed with `->` are often dynamically allocated on the heap and require explicit memory management (`new` and `delete`).
4. **Safety and Convenience**:
   - The dot operator (`.`) is straightforward and used with non-pointer types, safer in terms of dereferencing errors.
   - The arrow operator (`->`) simplifies accessing members through pointers without manually dereferencing them with `*`.

Examples to Illustrate the Differences

Example 1: Using `.` with Struct

```cpp
struct Rectangle {
    int width, height;
};

Rectangle rect;
rect.width = 50;  // Directly accessing and setting width
rect.height = 100; // Directly accessing and setting height
```

Example 2: Using `->` with Struct Pointer

```cpp
struct Rectangle {
    int width, height;
};

Rectangle* rectPtr = new Rectangle;
rectPtr->width = 50;  // Accessing and setting width through a pointer
rectPtr->height = 100; // Accessing and setting height through a pointer

delete rectPtr; // Clean up dynamically allocated memory
```

Example 3: Comparing `.` and `->` with Class Objects

```cpp
class Car {
public:
    void start() {
        // Implementation
    }
};

Car myCar;
myCar.start(); // Using dot operator to call start method

Car* carPtr = new Car;
carPtr->start(); // Using arrow operator to call start method through a pointer

delete carPtr; // Clean up dynamically allocated memory
```

Summary

- Use `.` when dealing with direct instances of an object or structure.
- Use `->` when dealing with pointers to objects or structures.

Understanding these operators is crucial for proper memory management and writing clear, bug-free C++ code, especially when dealing with pointers and dynamic memory allocation.

Doubly Linked List (DLL) in C++

A Doubly Linked List (DLL) points to both the next and the previous node.

Basic Structure of a Doubly Linked List:

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

    Node(int data1, Node* next1 = nullptr, Node* back1 = nullptr) {
        data = data1;
```

```
            next = next1;
            back = back1;
        }
    };
```

Converting an Array to a Doubly Linked List:

cpp

```cpp
Node* ConvertToLL(vector<int> &arr) {
    Node* head = new Node(arr[0]);
    Node* prev = head;
    for (int i = 1; i < arr.size(); i++) {
        Node* temp = new Node(arr[i], nullptr, prev);
        prev->next = temp;
        prev = temp;
    }
    return head;
}
```

Deleting the Head of a Doubly Linked List:

When the DLL has many elements, move the head to the second element and isolate the first element by setting its `next` and `back` to null.

cpp

```cpp
Node* deleteHead(Node* head) {
    if (head == NULL || head->next == NULL) {
        return NULL;
    }
    Node* prev = head; // Store current head
    head = head->next; // Move head to the next node
    head->back = nullptr; // Nullify the back pointer of the new head
    prev->next = nullptr; // Nullify the next pointer of the isolated node
    delete prev; // Delete the isolated node
    return head;
}
```

Deleting the Tail of a Doubly Linked List:

Traverse to the last node and remove it.

cpp

```cpp
Node* DeleteTail(Node* head) {
    if (head == NULL || head->next == NULL) {
        // If the list is empty or has only one node, return NULL
        if (head != NULL) {
            // If there's a single node, free it
            delete head; // Assuming nodes are allocated with 'new'
        }
        return NULL;
    }

    // Traverse to the second-to-last node
    Node* tail = head;
    while (tail->next->next != NULL) {
```

```
        tail = tail->next;
    }

    // Now 'tail' points to the second-to-last node
    Node* lastNode = tail->next; // The node to delete

    // Update the second-to-last node to be the new tail
    tail->next = NULL;

    // Free the memory of the last node
    delete lastNode; // Assuming nodes are allocated with 'new'

    return head;
}
```

Deleting the Kth Element of a Doubly Linked List:

Depending on the position and connections of the kth element, different cases are handled.

cpp

```
Node* DeleteKthElement(Node* head, int k) {
    if (head == NULL) {
        return NULL;
    }
    int cnt = 0;
    Node* kNode = head;
    while (kNode != NULL) {
        cnt++;
        if (cnt == k) break;
        kNode = kNode->next;
    }
    Node* prev = kNode->back;
    Node* front = kNode->next;

    if (prev == NULL && front == NULL) {
        return NULL;
    } else if (prev == NULL) {
        // Deleting the head
        return deleteHead(head);
    } else if (front == NULL) {
        // Deleting the tail
        return DeleteTail(head);
    }

    prev->next = front;
    front->back = prev;
    kNode->next = nullptr;
    kNode->back = nullptr;
    delete kNode;
    return head;
}
```

Insertion of Elements in a Doubly Linked List

Inserting Before the Head:

cpp

```
Node* InsertBeforeHead(Node* head, int val) {
    Node* newHead = new Node(val, head, nullptr);
```

```
        head->back = newHead;
        return newHead;
    }
```

Insertion at the Kth Position:

cpp

```
Node* InsertatKth(Node* head, int k, int val) {
    if (k == 1) {
        return InsertBeforeHead(head, val);
    }
    Node* temp = head;
    int cnt = 0;
    while (temp != NULL) {
        cnt++;
        if (cnt == k) break;
        temp = temp->next;
    }
    Node* prev = temp->back;
    Node* newNode = new Node(val, temp, prev);
    prev->next = newNode;
    temp->back = newNode;
    return head;
}
```

Inserting Before the Tail:

cpp

```
Node* InsertB4Tail(Node* head, int val) {
    if (head->next == NULL) {
        return InsertBeforeHead(head, val);
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    Node* prev = temp->back;
    Node* newNode = new Node(val, temp, prev);
    prev->next = newNode;
    temp->back = newNode;
    return head;
}
```