**// try understanding ->**

**// the next pointer points to a Node object, and that object is shown as a pointer pointing to that object**

**// head -> next ---> second(full object), pointed by a pointer named second**

**// pointer to pointer joins the objects together.**

**// if you made objects, it would have copied values from one object to the next, and they wont be separate entities**

**// head.next= second, all the values of second -> Value and Next gets copied to head.next , overwrites value of head.next**

**// pointer to pointer links objects instead of copying data**

**// pointers make lists dynamically , so no fixed size.**

**// pointers are copied... not the value**

**// transverse the next pointers to reach somewhere**

**// when you make a node's NEXT pointer to point to another node, youre not copying the node's data. Instead, you're simply making the NEXT pointer of one node point to the address of the next node**

**// to conclude, you join the next pointers to link the different nodes together.**

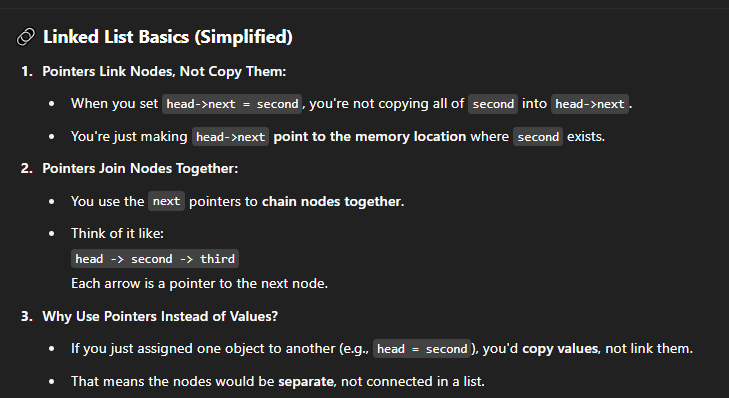
**// and**

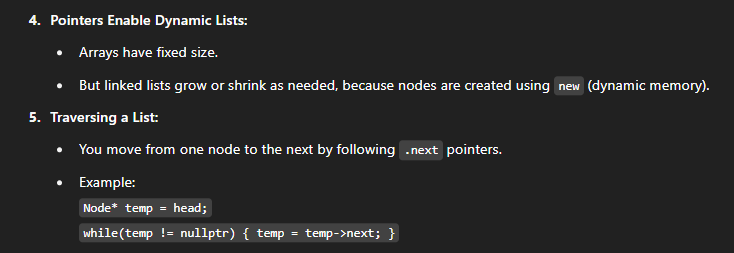
**// for more doubts..see screen recording in DSA group in phone**

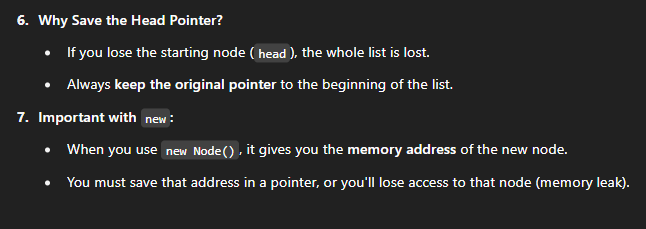
**// we transverse addresses using addresses...to save the current addresses(HEad) so they dont get lost when they point to another address (second)**

**// you need to store the pointer returned by new(), or your data vanishes**

**CHATGPT EXPLANATION TO ALL THIS 🡪**

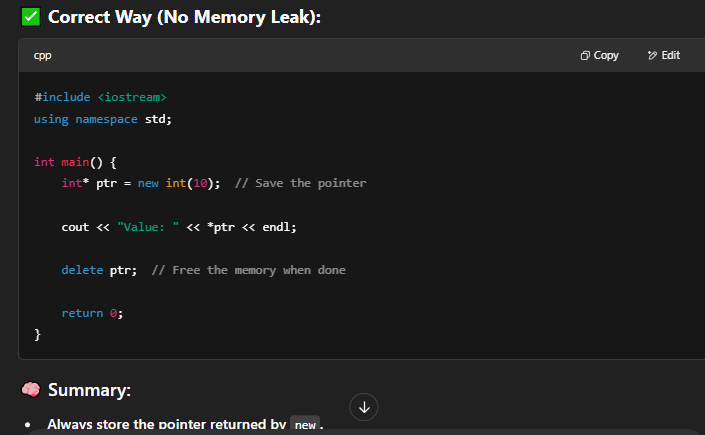






**THIS IS MEMORY LEAK 🡪**





**CODE 🡪**

#include<iostream>

using namespace std;

class Node {

public:

int Value;

Node\* Next;

};

void printlist(Node\* n) {

while (n != NULL) {

cout <<"value of Value: "<< n->Value << endl;

n = n->Next; // hai to address hi..n was pointing to the first node..now it is to the second

}

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->Value = 1;

head->Next = second;

second->Value = 2;

second->Next = third;

third->Value = 3;

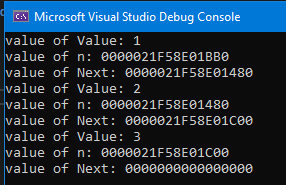
third->Next = NULL;

printlist(head);

return 0;

}

**SEE HOW THE N->Next IS THE ADDRESS OF THE VALUE OF N FOR THE NEXT NODE**

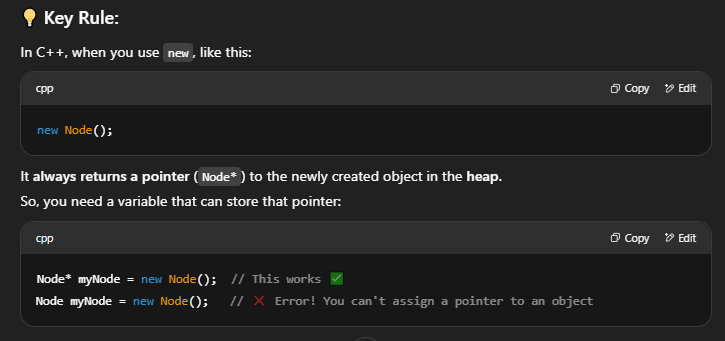


**Ab tell me… agar we didn’t make pointers pehle (here n) we couldn’t have joined all the other nodes ke next pointers with the first one agar vo hi pointer na hota**

**FINAALY I GET IT!!!!**

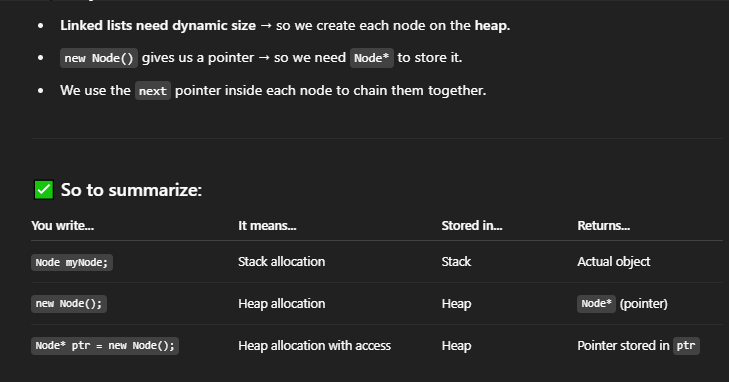
**see the function, if head wasnt a pointer, we would have no way to know the ADDRESS to the FIRST node aka the head of the linked list, and baaki sab to ho jati but there’s no Node before head jiske Next mein Head ka address store hoga**

**YES**

****

**LO JI**

**dynamic allocation ALWAYS returns pointer. baat hi khatam**

****

**FINALLY DUDE**

**why pointers ->**

1. **don’t get deleted when go out of scope..manual new() and delete()**
2. **dynamic allocation returns pointers**
3. **to join the head to rest of the nodes**

**add point if I missed something out**

**INSERTING 🡪**

**at front**

void InsertAtFront(Node\*\* head, int newval) { // u need address of there to put...pointing to address of address storing pointer...so \*\* (pointer to a pointer)

// 1) Make a NewNode

Node\* newnode = new Node();

newnode->Value = newval;

// 2) Put node in front of current head

newnode->Next = \*head; // as newnode is also a pointer, you point it to a pointer so \*head.... points to current first element of linked list

// 3) Move head of the list to point to the new node

\*head = newnode; // new address of head, kinda like reverse of step 2... head is position of new node

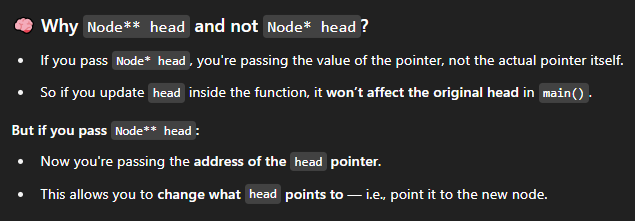
}

code🡪

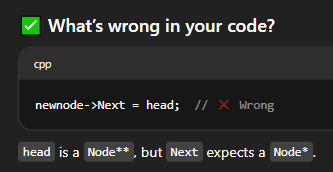
cout << "adding 0" << endl;

InsertAtFront(&head, 0);

printlist(head);

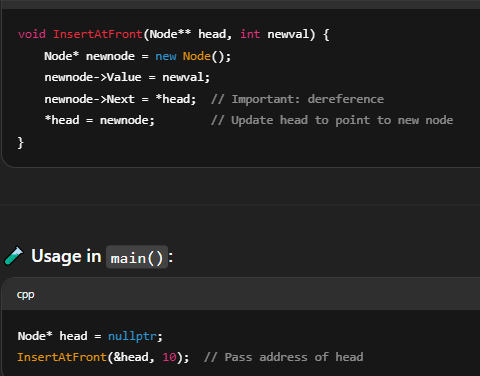


**for pointers, pass by reference is \*\* not \* (pass by value) as they are show by \***

****

**POINTER 🡪 POINTER … \*\* dereferences the pointer yk that**

**FINAL CODE🡪**

****

**CODE🡪**

#include<iostream>

using namespace std;

class Node {

public:

int Value;

Node\* Next;

};

void printlist(Node\* n) {

while (n != NULL) {

cout << n->Value << endl;

n = n->Next;

}

}

void InsertAtFront(Node\*\* head, int newval) { // u need address of there to put...pointing to address of address storing pointer...so \*\* (pointer to a pointer)..... late edit..... node\* is just the type of head, like int and stuff... you passed a pointer to that variable to pass by reference.

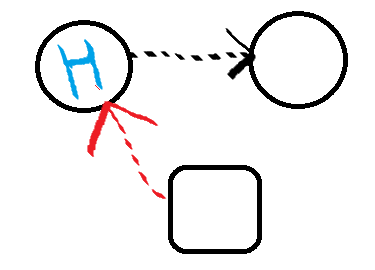
// forget for a moment ki it was a pointer, its like \* (\*)-> pointer to a pointer...second one is the type of variable.

WAIT

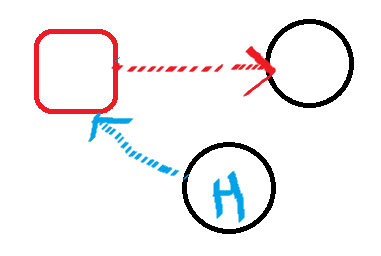
DUDE BREAKTHROUGH

WE DID \*HEAD AND NOT HEAD…CAUSE HEAD TO KUCH HAI HI NAHI…ITS JUST A POINTER TO THE FIRST ELEMENT OF THE LIST TO AVOID MEMORY LEAK…CAUSE ITS DYNAMIC ALLPCATION AND NEXT MEIN BAAKI SAB NODES KA ADDRESS HAI, U NEED A POINTER TO STORE THE ADDRESS OF THE FIRST NODE

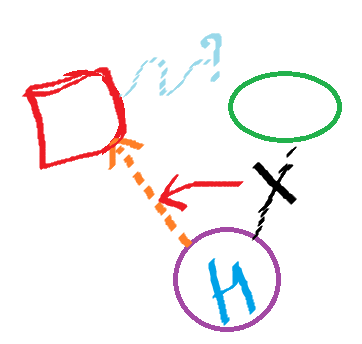
THIS IS WHAT CODEBEAUTY WAS SAYING..IF YOU MOVED HEAD N MADE IT POINT TO THE NEWNODE FIRST, JO USME ADDRESS STORED THAT WILL BE LOST !



THEN🡪>>>>>>>>>>>>



BECAUSE🡪>>>>>>>>>>>



you don’t know where to point now cause that info was stored in head…so first point to head, and then head will point to you

// 1) Make a NewNode

Node\* newnode = new Node();

newnode->Value = newval;

// 2) Put node in front of current head

newnode->Next = \*head; // as newnode is also a pointer, you point it to a pointer so \*head.... points to current first element of linked list

// 3) Move head of the list to point to the new node

\*head = newnode; // new address of head, kinda like reverse of step 2... head is position of new node

}

void InsertAtEnd(Node\*\* head, int newval) {

// 1) Prepare a NewNode.

Node\* NewNode = new Node();

NewNode->Value = newval;

NewNode->Next = NULL;

// 2) If linked list is empty,NewNode will be the head node. (else-->)

if (\*head == NULL) {

\*head = NewNode;

return;

}

// 3) Find the last node.

Node\* last = \*head; // this will traverse the list :)

while (last->Next != NULL) {

last = last->Next;

}

// 4) Insert NewNode after last node, at the end of the linked list.

last->Next = NewNode;

}

void insertAfter(Node\*Previous,int newval) {

// 1) Check if previous node is NULL.

if (Previous == NULL) {

cout << "Previous node cant be NULL." << endl;

}

// 2) Prepare a NewNode.

Node\* NewNode = new Node();

NewNode->Value = newval;

// 3) Insert NewNode after previous

// note that while inserting, before pointing to the new node, save the address of the next node in line or it will be lost forever

NewNode->Next = Previous->Next; // newnode to next node

Previous->Next = NewNode;

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->Value = 1;

head->Next = second;

second->Value = 2;

second->Next= third;

third->Value = 3;

third->Next = NULL;

printlist(head);

// INSERT AT FRONT-->

cout << "adding 0" << endl;

InsertAtFront(&head, 0);

printlist(head);

// INSERT AT END -->

cout << "\n\nadding 4" << endl;

InsertAtEnd(&head,4);

printlist(head);

// INSERT AFTER HEAD

cout << "\n\nadding negatives: \n" << endl;

insertAfter(head, -1);

insertAfter(second, -2);

printlist(head);

return 0;

}

// **everything here is a pointer...so Node\* n is default...so this is pass by value by default...if u wanna make change in list by insertion , you have to pass my ref and thus add another asterisk**

**STOP FUCKING ASKING URSELF \*HEAD KAHA SE AA RAHA HAI WHY NOT JUST HEAD**

**WE PASSED \*HEAD THAT’S WHY… \*(\*HEAD), \* IS A SYMBOL OF IT BEING A POINTER, TELLING THE COMPILER THIS…BUT USKA NAME IS BLOODY HEAD ONLY, AND WE PASSED A POINTER TO THAT POINTER FOR PASS BY REFERENCE, TABHI \* HEAD**

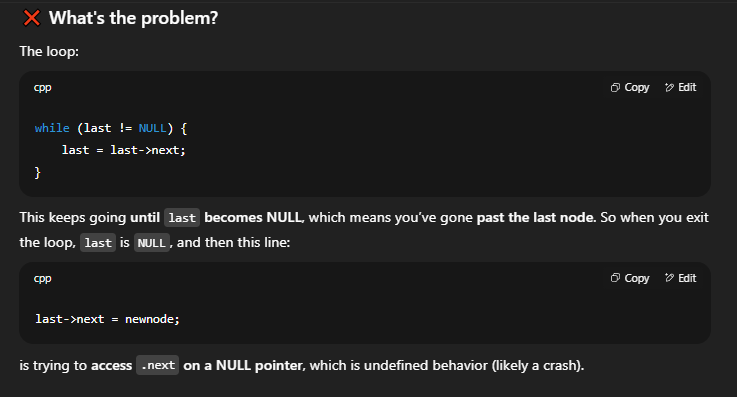
**CAUSE \* HEAD ISSSSSSSS THE ARGUMENT**

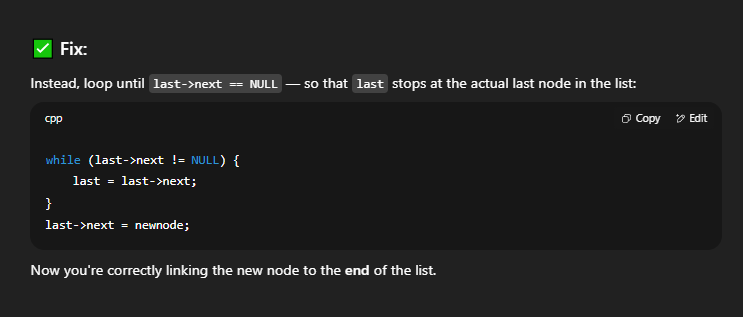
**NOT HEAD !!!**

**(also \*\* deref it…so when pointing to NULL we need to do \*Head)**

**ERROR ----🡪**

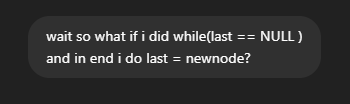
**insert in end**

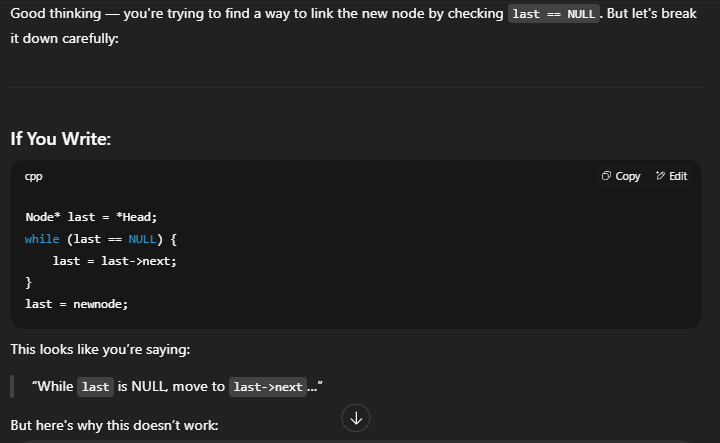
****

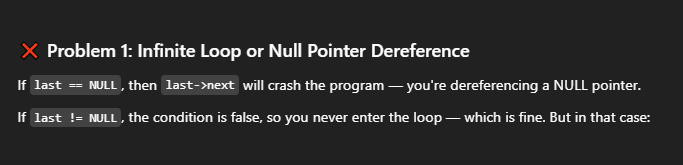
****

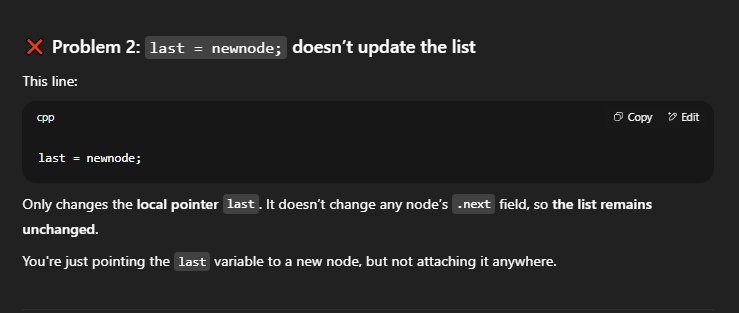
**so now last stops before newnode. . . as last -> next = newnode;**

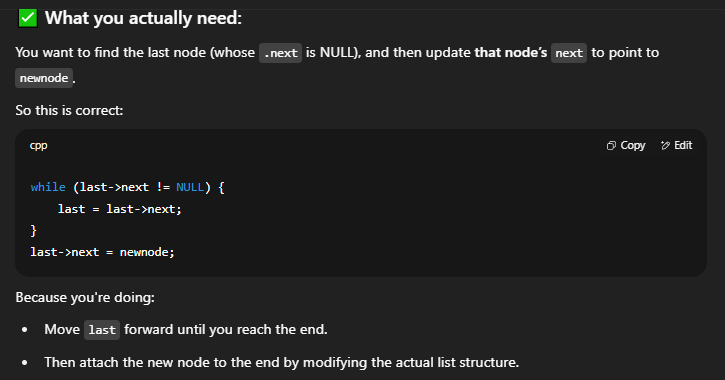
**newnode is the end node now**

****

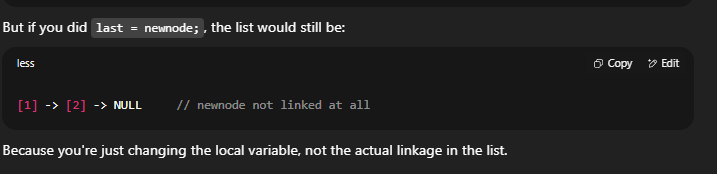
****

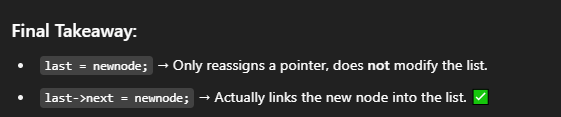
****

****

****

****

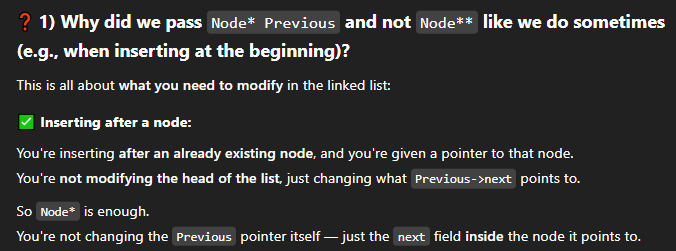
****

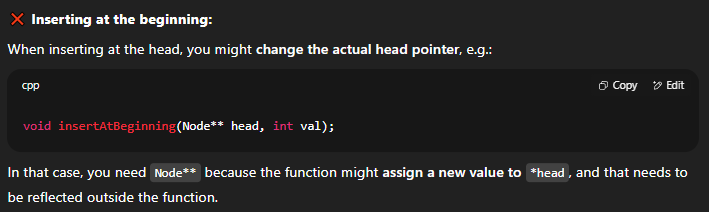
****

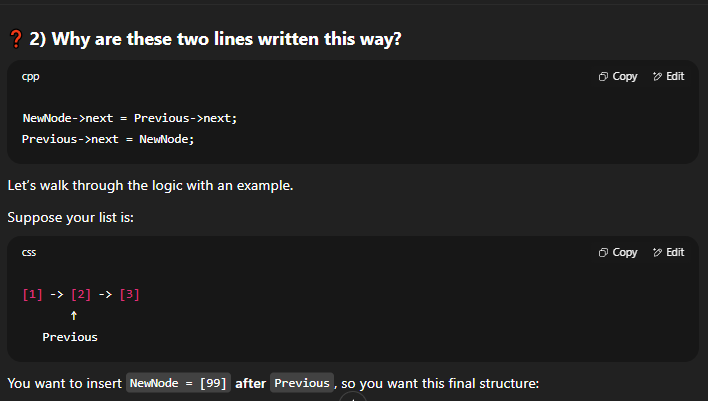
**First 3 Lines For Newnode, Uske Baad You Determine What That \*Head Points To**

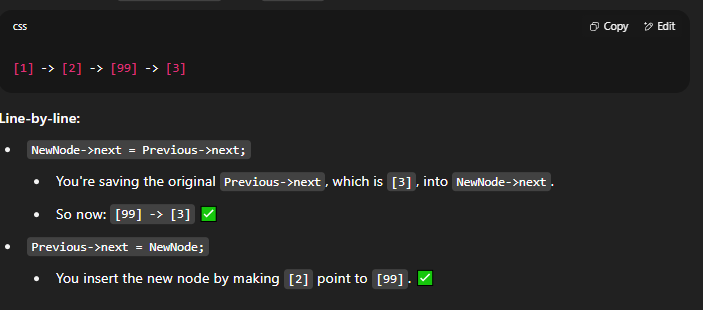
**INSERTING AFTER A NODE 🡪**

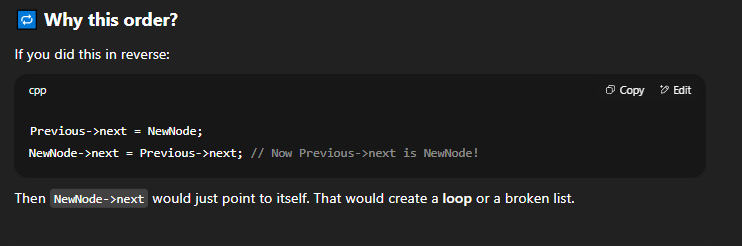
**when does head pointer change 🡪**

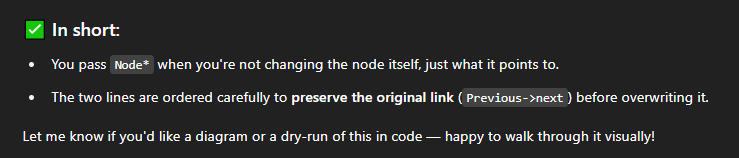
****

****

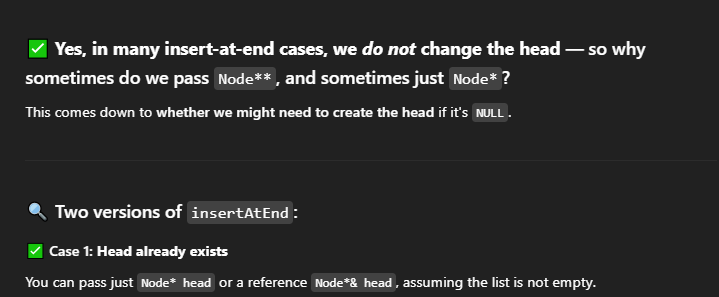
****

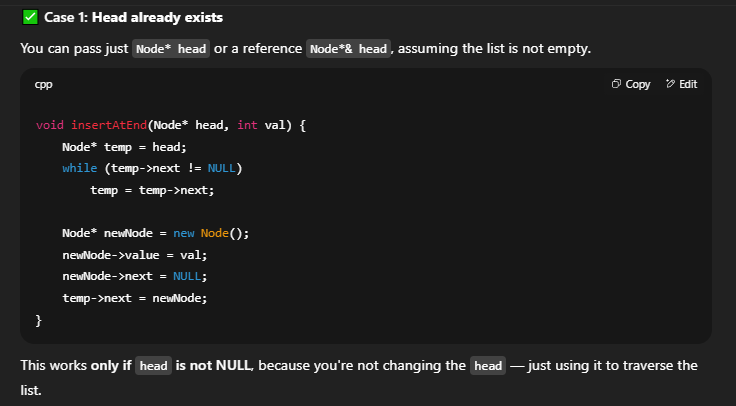
****

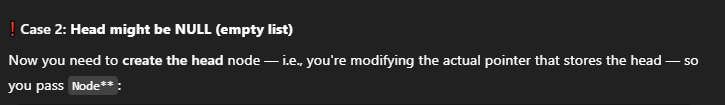
****

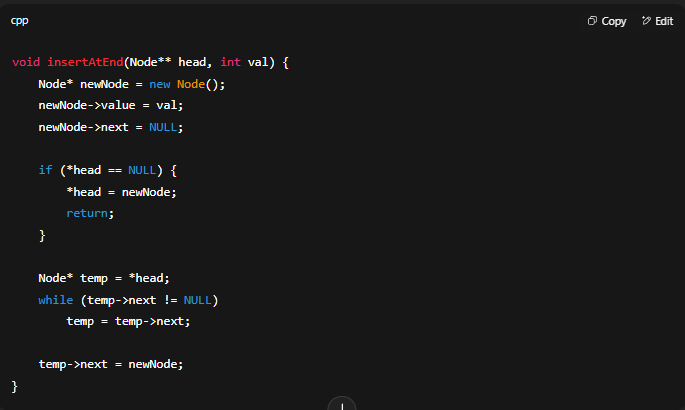
****

**2)**

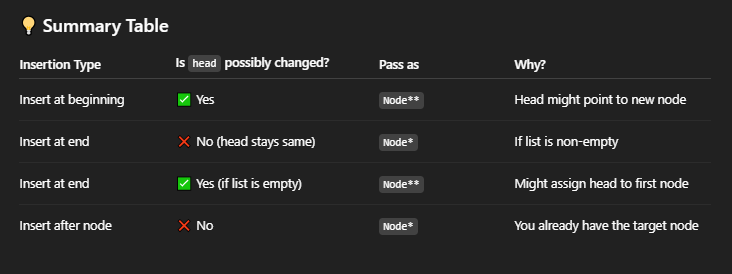
****

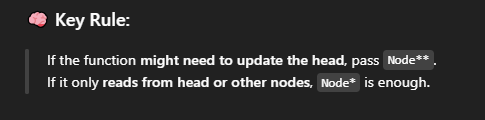
****

****

****

****

****

****

27/06

insertions completed

IMPORTANT

**Instead of Node\*\* , u can also write the c++ version Node\*& Head**

**just interchange \*Head with Head, and in parameter, don’t write &head… head only is fine.**

**see…simpler and more ‘c++’**

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* next;

};

void print\_list(Node\* n) {

while (n != NULL) {

cout << n->value << endl;

n = n->next;

}

}

void InsertInBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = Head;

Head = newnode;

}

void InsertInEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = NULL;

if (Head == NULL) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next != NULL) {

last = last->next;

}

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = prev->next;

prev->next = newnode;

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->value = 1;

head->next = second;

second->value = 2;

second->next = third;

third->value = 3;

third->next = NULL;

print\_list(head);

cout << "\n\n\t\t\t\t\tINSERTIONS ----->" << endl;

InsertInBeginning(head, 0);

InsertInEnd(head, 100);

InsertAfter(third, 4);

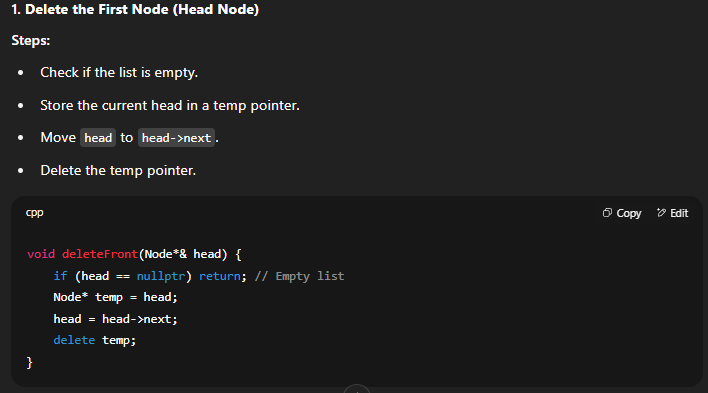
print\_list(head);

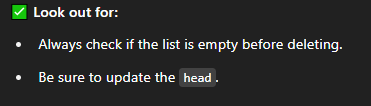
return 0;

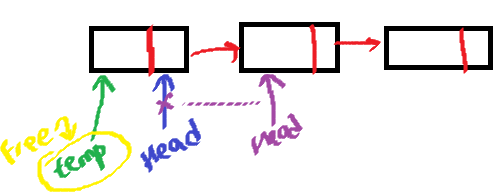
}

**DELETIONS**

1. **deleting first node**

****

****

****

void DeletingFirstNode(Node\*& Head) {

if (Head == NULL) {

cout << "Cant delete NULL" << endl;

return;

}

Node\* temp = Head;

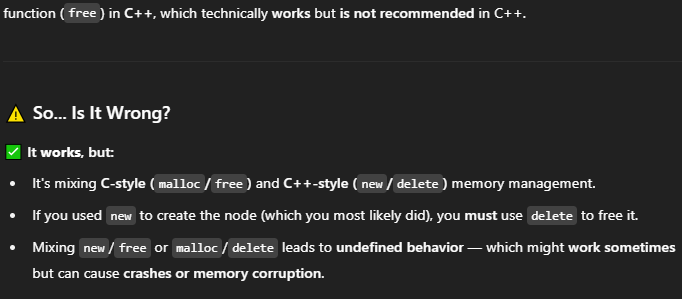
Head = Head->next;

delete (temp);

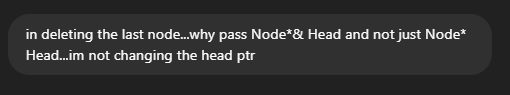
}

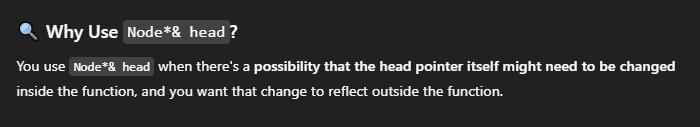
****

**you CAN … but why mix**

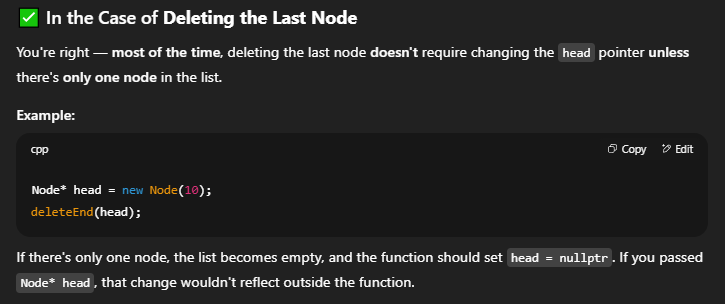
****

1. **DELETING END NODE**

****

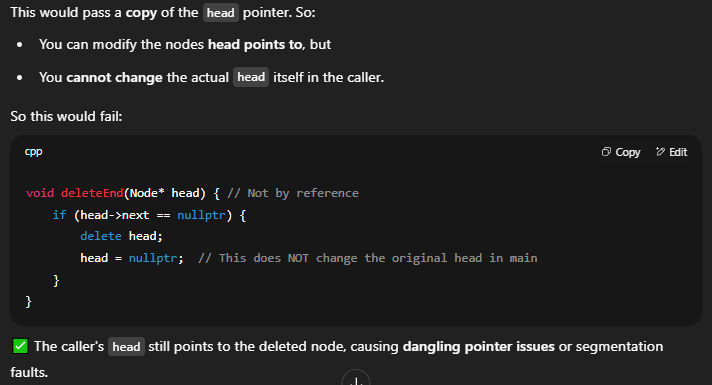
****

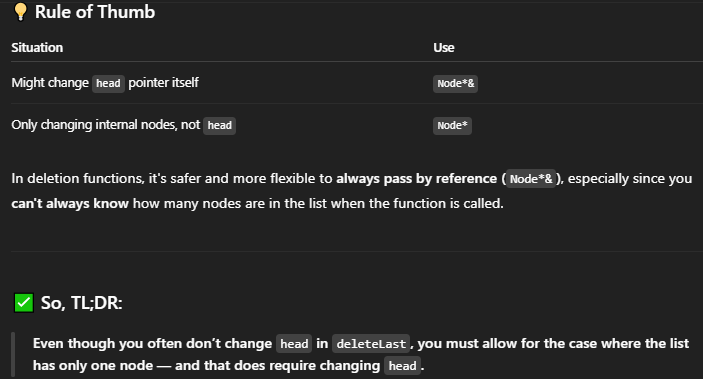
**USE Node\*& Head . . . even when there is a CHANCE to be safe**

****

**ONLY TIME YOU CHANGE HEAD PTR 🡪**

**so if u pass Node\* Head, the changes wont be implemented in the real LL**

****

****

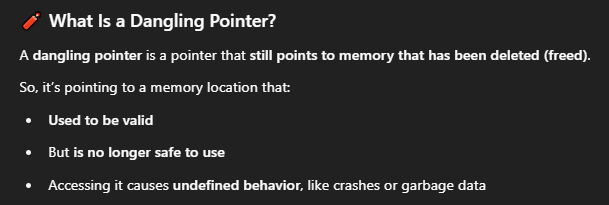
**UNDERSTAND?**

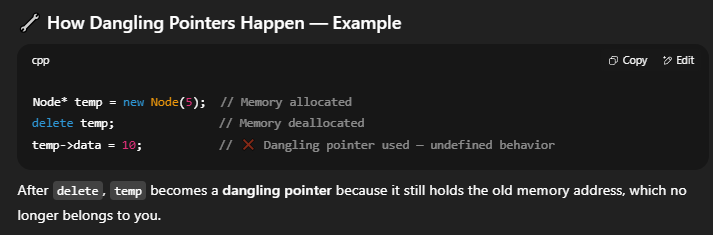
**no u didn’t**

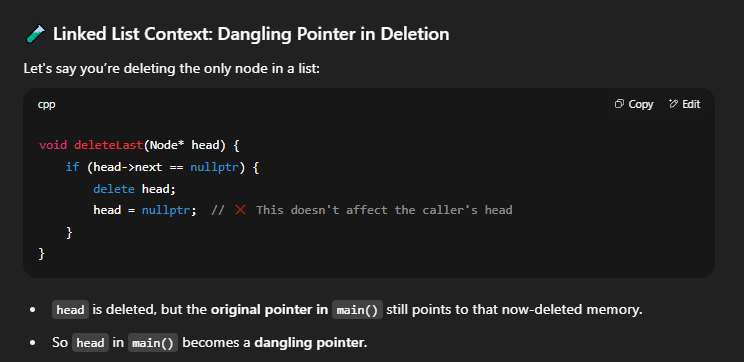
**read this again 🡪**

****

**DANGLING POINTER ISSUES**

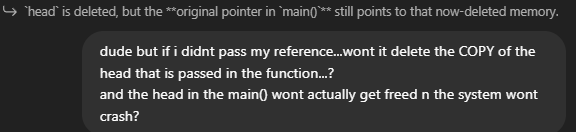
****

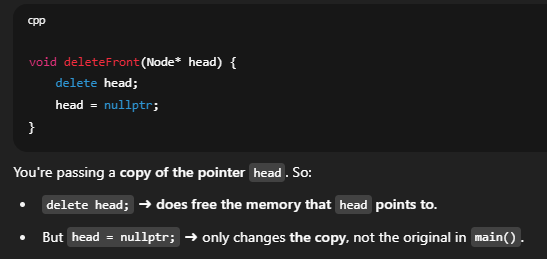
****

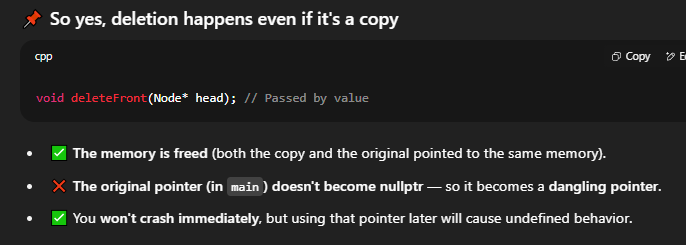
****

**SO THE MEMORY GOT DELETED. . . YOU LOST CONTROL. . . BUT BECAUSE YOU PASSED BY VALUE. . . THE HEAD STILL POINTS TO (whatever it did to) NOT NULL…SO ITS ‘ DANGLING’.**

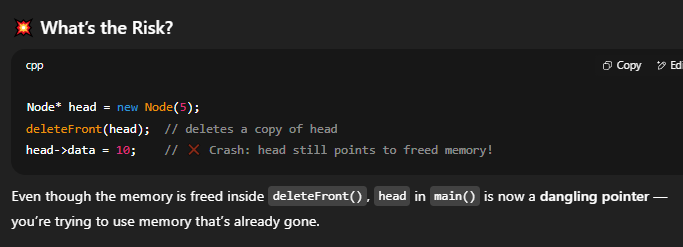
**But then you might ask. . I passed by value. . how did it delete in the main function ?**

****

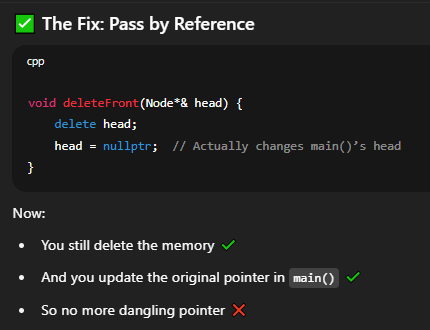
****

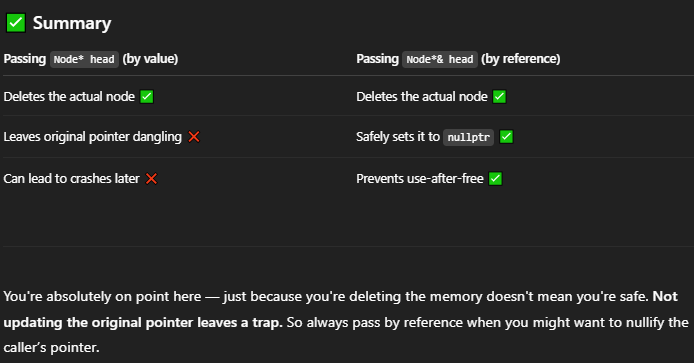
****

**what this means is… deletion does happen, copy or by ref, but by copy “Head=NULL” doesn’t work and THAT is a problem.**

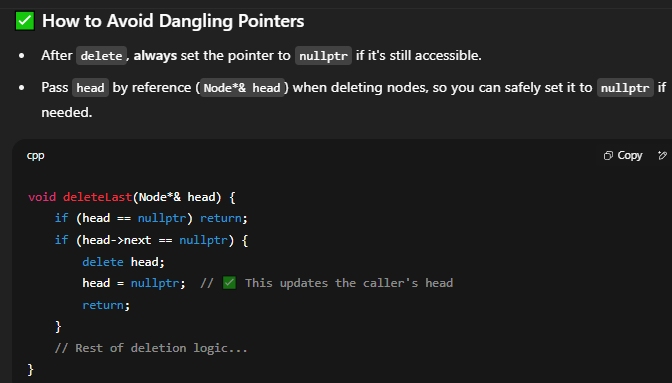
****

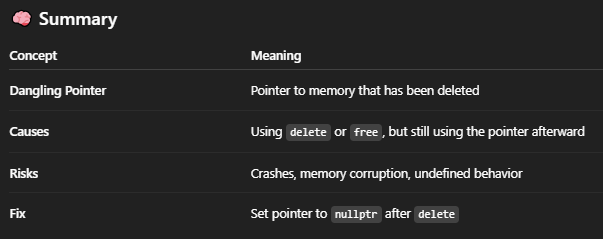
**SO WHAT TO DO?**

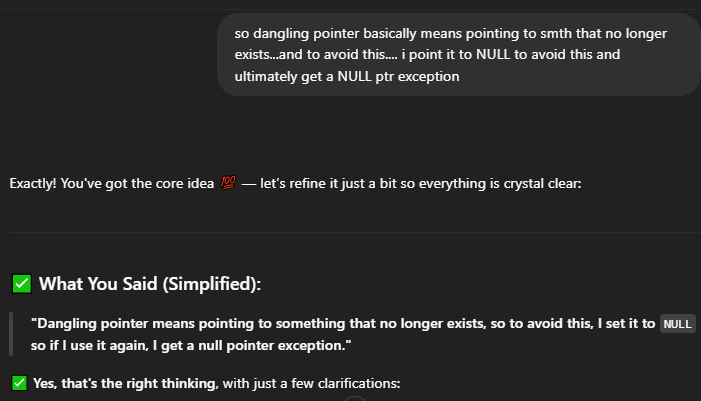
****

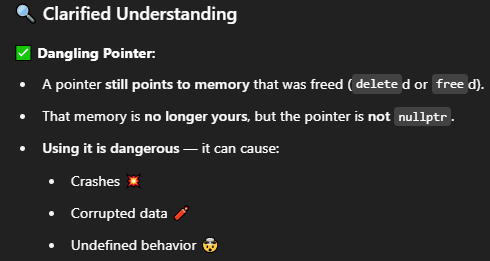
****

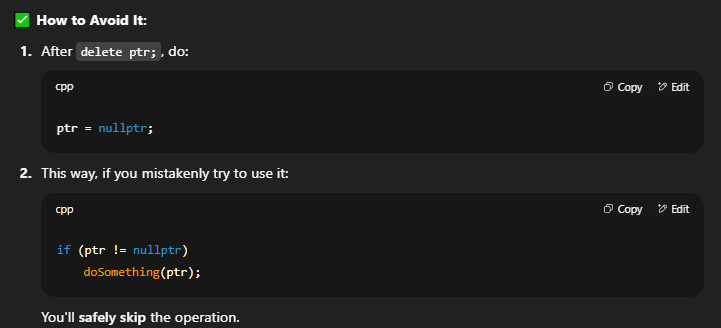
**Continuing 🡪**

****

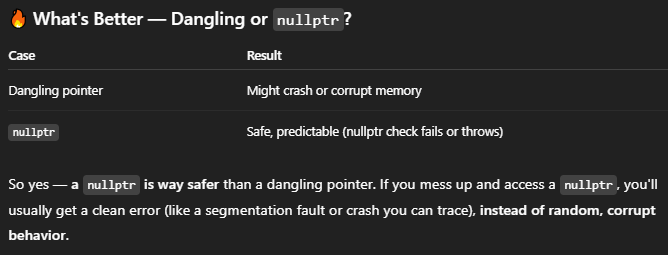
****

****

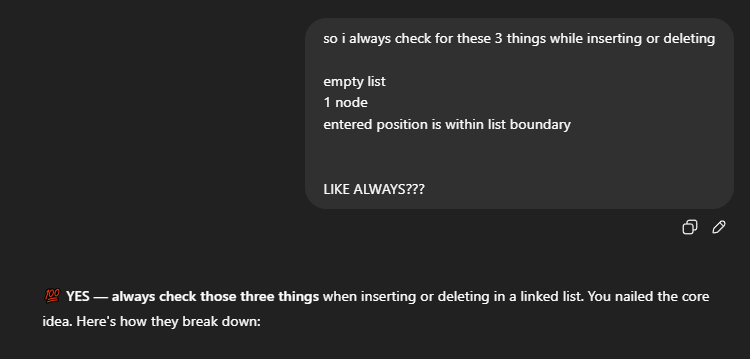
****

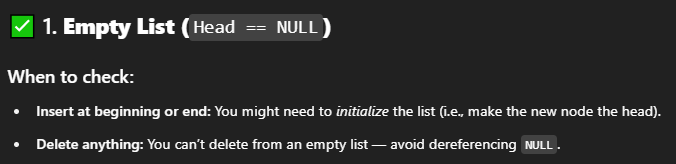
****

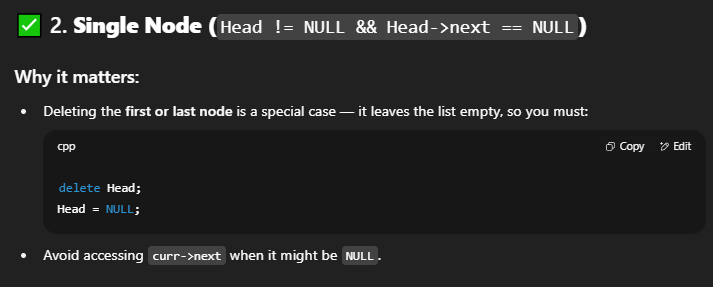
**THIS IS IMPORTANT**

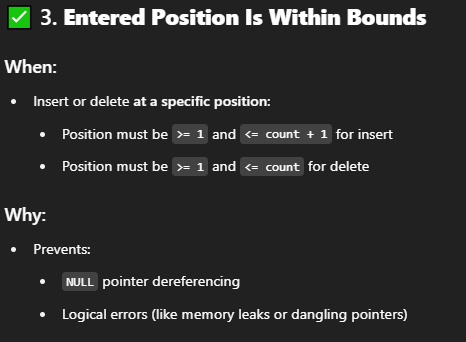
****

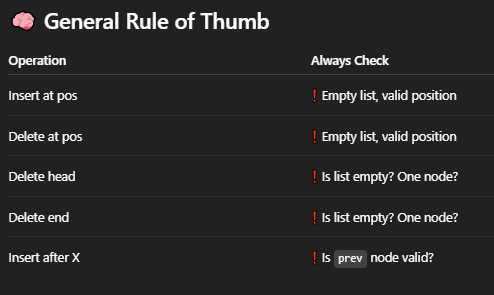
FINAL CHATPGT APPROVED CODE ---🡪











CODEEEEEEEEEEEEEEEEEEEE 🡪

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* next;

};

void print\_list(Node\* n) {

cout << "List: ";

while (n != NULL) {

cout << n->value;

if (n->next) cout << " -> ";

n = n->next;

}

cout << endl;

}

int count\_list(Node\* n) {

int count = 0;

while (n != NULL) {

count++;

n = n->next;

}

return count;

}

void InsertInBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = Head;

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = NULL;

if (Head == NULL) {

Head = newnode;

return;

}

Node\* last = Head;

while (last->next != NULL) {

last = last->next;

}

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

if (prev == NULL) {

cout << "prev cant be NULL !" << endl;

return;

}

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = prev->next;

prev->next = newnode;

}

void DeleteFirst(Node\*& Head) {

if (Head == NULL) { // no Node

cout << "NULL cant be deleted !" << endl;

return;

}

if (Head->next == NULL) { // one Node

delete(Head);

Head = NULL;

return;

}

Node\* temp = Head;

Head = Head->next;

temp->next = NULL;

delete(temp);

}

void DeleteEnd(Node\*& Head) {

if (Head == NULL) { // no node

cout << "NULL cant be deleted !" << endl;

return;

}

if (Head->next == NULL) { // one node

delete(Head);

Head = NULL;

return;

}

Node\* prev = NULL;

Node\* curr = Head;

while (curr->next != NULL) {

prev = curr;

curr = curr ->next;

}

prev->next = curr->next;

curr->next = NULL;

delete(curr);

}

void DeleteNode(int pos, Node\*& Head) {

int len = count\_list(Head);

if (Head == NULL) {

cout << "head cant be NULL" << endl;

return;

}

if (pos<1 or pos>len) {

cout << "Can't delete outside the linked list !!!" << endl;

return;

}

Node\* prev = NULL;

Node\* curr = Head;

if (pos == 1) {

DeleteFirst(Head);

return;

}

else {

for (int i = 1; i < pos; i++) {

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

curr->next = NULL;

delete(curr);

}

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->value = 1;

head->next = second;

second->value = 2;

second->next = third;

third->value = 3;

third->next = NULL;

InsertInBeginning(head, 0);

InsertAtEnd(head, 100);

InsertAfter(third, 4);

print\_list(head);

cout << "\n\n\t\t\t\t\tDELETING" << endl;

cout << "deleting first Node: " << endl;

DeleteFirst(head);

print\_list(head);

cout << "\n\ndeleting 2nd Node: " << endl;

DeleteNode(2, head);

print\_list(head);

cout << "\n\ndeleting last Node: " << endl;

DeleteEnd(head);

print\_list(head);

return 0;

}

psstt !

I did it ... full code no mistakes… 9/7/25 11:00pm

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* next;

};

void Print\_list(Node\* n) {

while (n != NULL) {

cout << n->value << endl;

n = n->next;

}

}

int Count\_List(Node\* n) {

int count = 0;

while (n != NULL) {

count++;

n = n -> next;

}

return count;

}

void InsertAtBeginning(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = Head;

Head = newnode;

}

void InsertAtEnd(Node\*& Head, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = NULL;

if (Head == NULL) {

cout << "Head cant be NULL !" << endl;

return;

}

Node\* last = Head;

while (last->next != NULL) {

last = last->next;

}

last->next = newnode;

}

void InsertAfter(Node\* prev, int newval) {

if (prev == NULL) {

cout << "prev cant be NULL !" << endl;

return;

}

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = prev->next;

prev->next = newnode;

}

void DeleteFirst(Node\*& Head) {

if (Head == NULL) {

cout << "Head cant be NULL !" << endl;

return;

}

Node\* temp = Head;

Head = Head->next;

delete(temp);

temp = nullptr;

}

void DeleteEnd(Node\*& Head) {

if (Head == NULL) {

cout << "Head cant be NULL !" << endl;

return;

}

if (Head->next == NULL) {

DeleteFirst(Head);

return;

}

Node\* prev = NULL;

Node\* curr = Head;

while (curr->next != NULL) {

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

delete(curr);

curr = nullptr;

}

void DeleteThisNode(int pos, Node\*& Head) {

int len = Count\_List(Head);

if (Head == NULL) {

cout << "Head cant be NULL !" << endl;

return;

}

if (pos > len or pos <= 0) {

cout << "enter position within the range of the limit !!!" << endl;

return;

}

if (pos == 1) {

DeleteFirst(Head);

return;

}

Node\* prev = NULL;

Node\* curr = Head;

for (int i = 1; i < pos; i++) {

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

delete(curr);

curr = nullptr;

}

int main() {

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

head->value = 1;

head->next = second;

second->value = 2;

second->next = third;

third->value = 3;

third->next = NULL;

cout << "og list: " << endl;

Print\_list(head);

cout << "\n\nInsertions --> " << endl;

InsertAtBeginning(head, 0);

InsertAtEnd(head, 100);

InsertAfter(third, 4);

cout << "list -->" << endl;

Print\_list(head);

cout << "\n\nDeletions --> " << endl;

DeleteFirst(head);

DeleteEnd(head);

DeleteThisNode(3, head);

cout << "list -->" << endl;

Print\_list(head);

return 0;

}

15/7

a little side note

void InsertAfter(Node\* prev, int newval) {

Node\* newnode = new Node();

newnode->value = newval;

newnode->next = prev->next;

prev->next = newnode;

}

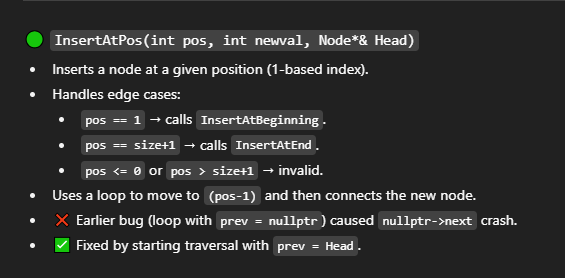
in this..if u pass head as argument..it will always print at the second position..

but if u pass after a node u declared, like second or third, then it will actually insert after that NODE

maybe the insertion after a node in DLL…by position is actually better cause of this

7 September 2025

Implementing new functions n adding to github



void InsertAtPos(int pos, int newval, Node\*& Head) {

int size = Count(Head);

if (pos > size || pos <= 0) {

cout << "enter pos within range of the list !" << endl;

return;

}

if (pos == 1 || !Head) {

InsertAtBeginning(Head, newval);

return;

}

if (pos == size + 1) {

InsertAtEnd(Head, newval);

return;

}

Node\* newnode = new Node();

newnode->value = newval;

Node\* prev = Head;

for (int i = 1; i < pos - 1; i++) {

prev = prev->next;

}

/\*

Node\* prev = nullptr;

for (int i = 1; i < pos; i++) {

prev = prev->next;

this is showing exception throw for trying to insert at pos = 2, cause prev = prev-> next means nullptr = nullptr

this causes exception throw. Declaring it with Head, and doing < pos - 1 to balance the loop rightens the situation

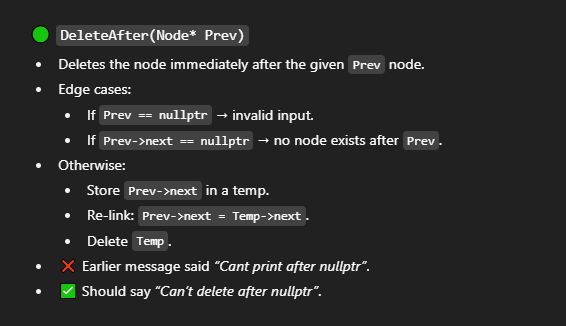
}

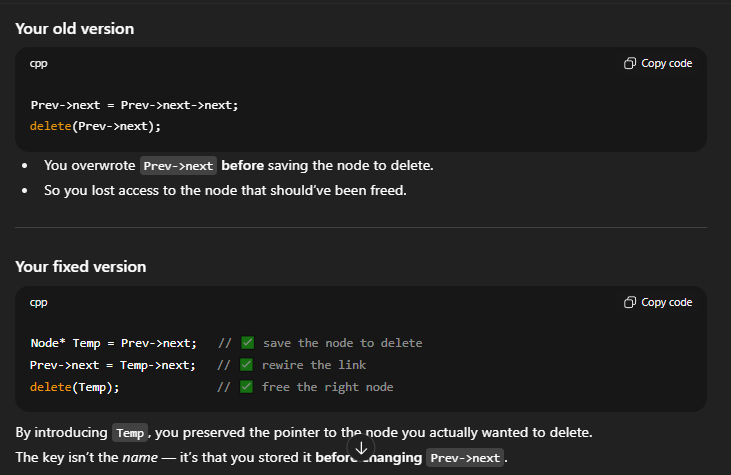
\*/

newnode->next = prev->next;

prev->next = newnode;

}





void DeleteAfter(Node\* Prev) {

if (!Prev) {

cout << "Cant delete after nullptr" << endl;

return;

}

Node\* Temp = Prev->next;

if (!Temp) {

cout << "cant delete NULL after single node" << endl;

return;

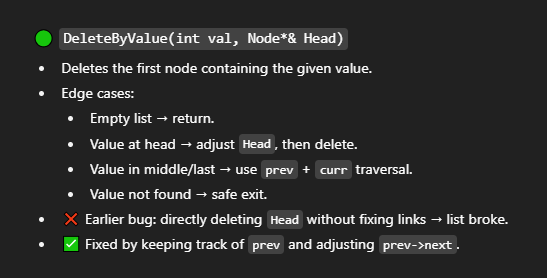
}

Prev->next = Temp->next;

delete(Temp);

// temp removes the confusion between Prev->next address and the node after Prev

}



void DeleteByValue(int val, Node\*& Head) {

if (!Head) {

return;

}

// Case 1: head node has the value

if (Head->value == val) {

Node\* temp = Head;

Head = Head->next;

delete temp;

return;

}

// Case 2: search further

Node\* curr = Head;

while (curr->next && curr->next->value != val) {

curr = curr->next;

}

// if not found

if (!curr->next) {

cout << "Node not found in list " << endl;

return;

}

Node\* temp = curr->next;

curr->next = temp->next;

delete temp;

}