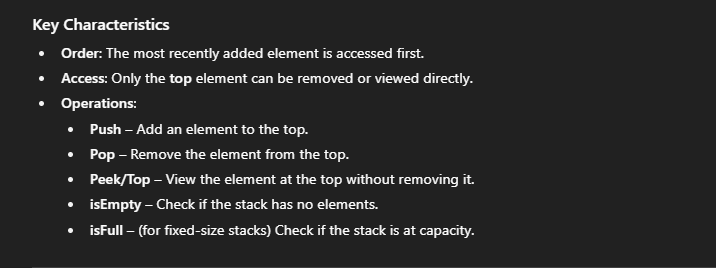


Like gym plates . . . take the lighter one out to use the heavier plate on the bottom.

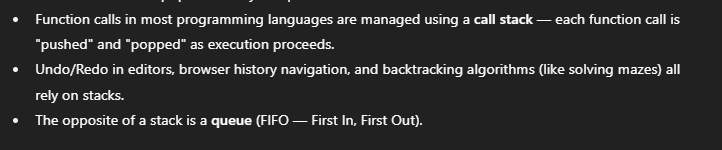


In stacks 🡪

Insertion -> Push(num)

Deletion -> Pop()

top -> index of the last element in the stack, initialized as -1 in the beginning (as top = 0 would mean there’s an element on the 0th index.

****

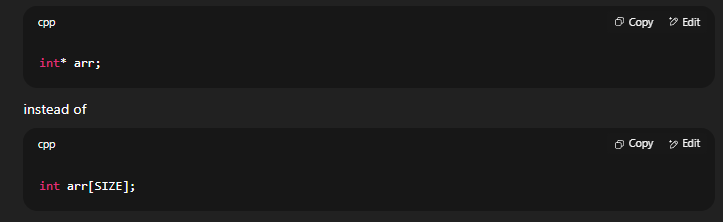
**RECURSION AND PROCEDURAL CALLS.**

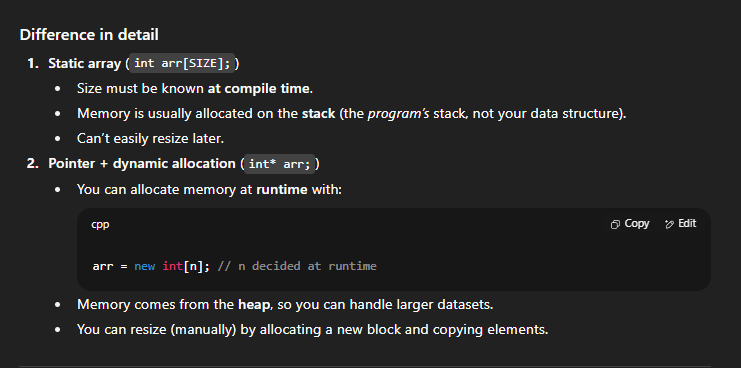
ALL DONE BY STACKS . . . as if you want to do a task in the middle of another task, that task will be implemented before the running task ( Last In First Out principle).

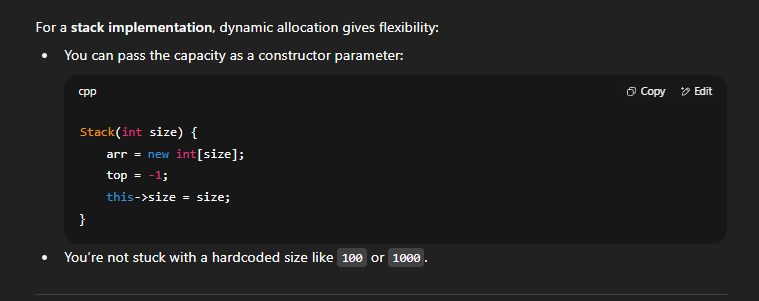
Same for Recursive functions.

**IMPLEMENTATION USING ARRAYS 🡪**

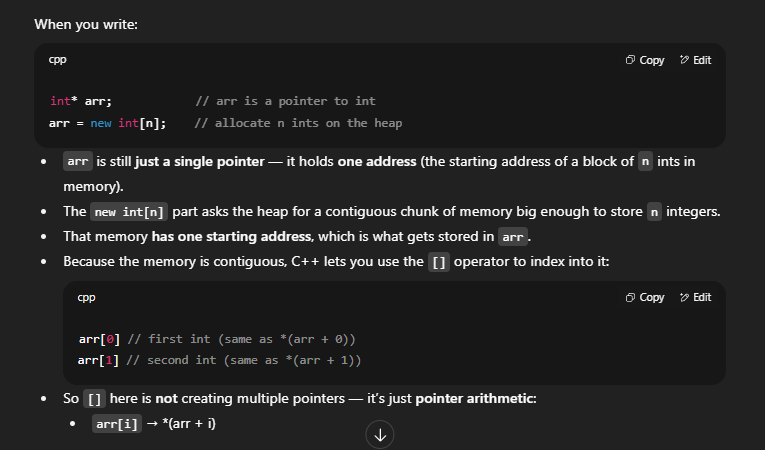
Initializing an array dynamically ->







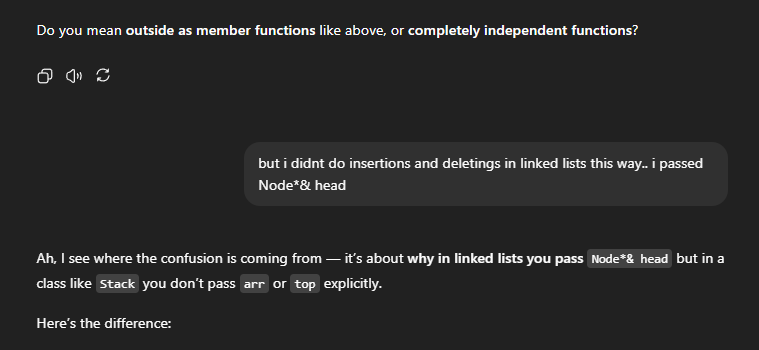
Imp 🡪

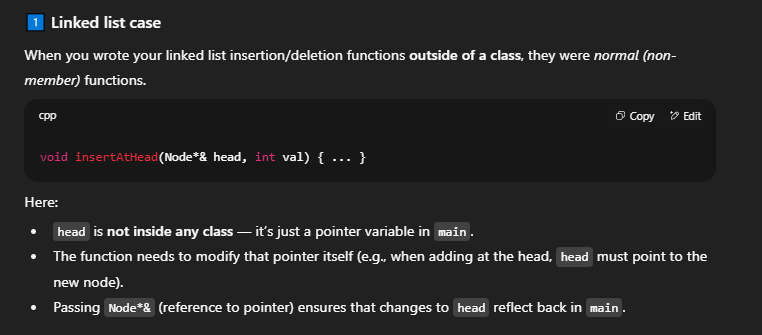


Int \* arr creates a pointer that points the beginning of the array , as we know that the name of the array arr = &arr[0].

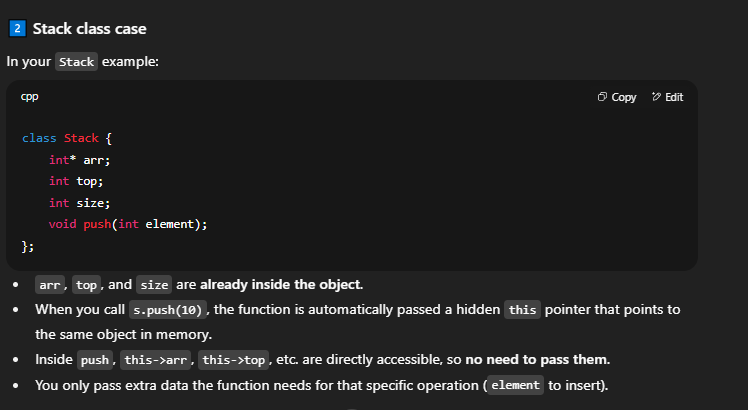
arr stores the address of the first element of the array (that’s why we could initialize the name of the array as a ptr, apart from dynamic allocation always returning a pointer, like in linked Lists)

Reason for encapsulating functions in stack class, instead of making them non-member-functions like in linked lists ->





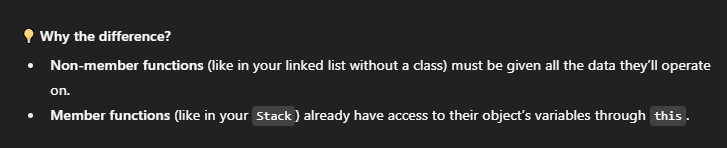
Remember head was a node defined in the main ()



Read this part again

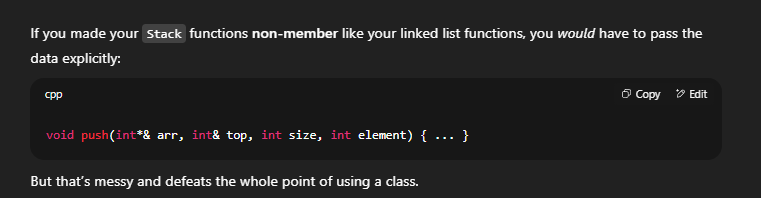


Head wasn’t defined in the Node class. So it had to be constantly passed around. Here, the values of the variables getting changed (top and size) are already in the class, and don’t need to be passed every time if the function is inside that class ( or implemented after declaring as Stack::Push(int element) )



Encapsulated functions access the data using the this -> pointer, that’s inbuilt/default.

Sir method ->



#include<iostream>

using namespace std;

class Stack {

public:

int\* arr; // dynamic allocation

int top;

int size;

// Making a Constructor

Stack(int size) {

this->size = size; // that object's size = size passed as argument

arr = new int[size];

top = -1;

}

// behavior

void push(int element) {

// check overflow

// note : you can also check overflow as if (top == size)

// but best for all conditions is -->

if (size - top > 1) {

top++; // top now points one index ahead

arr[top] = element; // as empty, 'element' is added at that index

}

else {

cout << "Exception Thrown. Stack Overflow" << endl;

return; // not really required to write, as end of function after this line anyways

}

}

void pop() {

// check underflow

if (top >= 0) {

cout << "popped element: " << arr[top] << endl;

top--;

}

else {

cout << "Exception Thrown. Stack Underflow" << endl;

return;

}

}

void peek() {

// check for both underflow and overflow

if (top >= 0) {

cout << "top most element : " << arr[top] << endl;

}

else {

cout << "Stack is empty" << endl;

return;

}

}

bool isEmpty() {

if (top == -1) {

return true;

}

else {

return false;

}

}

};

int main() {

Stack st(5); // size = 5, top = -1, array[5] created

st.push(1);

st.push(2);

st.push(3);

st.peek(); // shows 3

st.pop(); // removes 3

cout << endl;

st.peek(); // shows 2

st.pop(); // removes 2

cout << endl;

st.peek(); // shows 1

st.pop(); // removes 1;

cout << endl;

st.peek(); // "stack is empty"

st.pop(); // Exception thrown

/\*

repeats same message as above

cout << endl;

st.peek();

st.pop();

\*/

cout << endl;

if (st.isEmpty()) {

cout << "Stack is Empty At the moment" << endl;

}

else {

cout << "there are elements present in the stack" << endl;

}

cout << "\n### Attempting Stack overflow ###\n" << endl;

st.push(1);

st.push(2);

st.push(3);

st.push(4);

st.push(5);

st.push(6);

cout << "\n### Peeking at topmost element ###\n" << endl;

st.peek(); // displays 5, as 6 not added.

// time complexitiy

// O(1) push, pop, peek, empty , as checking by if ( ) , thats constant

return 0;

}

REVISITED ON 15 SEPTEMBER 2025

FINISHED PAST MIDNIGHT

Added functions – Display, Count, SearchFor, ClearStack, etc.

// Good Morning at 8:00 pm

// Array implementation of Stack

#include<iostream>

using namespace std;

class Stack {

public:

int\* Arr; // cant do int\* Arr[] now, later dynamically allocate as new int[] instead of new int()

int Size;

int Top;

// Constructor to :- Initialise Size to argument, array size = Size, Top = -1

Stack(int s) {

this->Size = s;

Arr = new int[s];

Top = -1;

}

void push(int element) {

// Checking if Stack is full; Overflow condition

if (Top == Size - 1) {

cout << "ERROR. OVERFLOW EXCEPTION. " << element << " caused overflow." << endl;

return;

}

Top++;

Arr[Top] = element;

}

void pop() {

if (Top == -1) {

cout << "ERROR. UNDERFLOW EXCEPTION. "<< endl;

return;

}

int temp = Arr[Top];

Top--;

cout << temp << " has been popped from the stack" << endl;

}

void peek() {

if (Top == -1) {

cout << "ERROR. Can't perform peek as stack is Empty" << endl;

return;

}

cout << "Topmost element: " << Arr[Top] << endl;

}

// Displaying all elements top to bottom

void Display() {

if (Top == -1) {

cout << "ERROR. Cant display elements as stack is empty" << endl;

return;

}

cout << "Displaying all elements top to bottom" << endl;

for (int i = Top; i >= 0; i--) {

cout << Arr[i] << " ";

}

cout << endl;

}

void Count() {

int Size = Top + 1;

cout << "Size of Stack: " << Size << endl;

}

void SearchFor(int key) {

for (int i = Top; i >= 0; i--) {

if (Arr[i] == key) {

cout << "Element found at position: " << i + 1 << endl;

return;

}

}

cout << "Element not found in the stack" << endl;

}

bool isEmpty() {

return Top == -1;

}

// Making a helper function

void checkEmpty(bool isEmpty) {

// type name --> no () as that would result in function calling

if (isEmpty) {

cout << "Stack has no elements" << endl;

return;

}

else

cout << "Stack is not Empty" << endl;

}

void ClearStack() {

Top = -1;

cout << "Stack has been reset" << endl;

}

~Stack() {

delete[] Arr;

cout << "Stack Memory freed from Heap" << endl;

}

};

int main() {

Stack s(3);

cout << "Operations on Empty Stack -->" << endl;

s.pop();

s.peek();

s.checkEmpty(s.isEmpty());

s.Display();

s.Count();

cout << endl;

cout << "Inserting elements and checking for overflow" << endl;

s.push(1); // 1

s.push(2); // 2 1

s.push(3); // 3 2 1

s.push(4); // OVERFLOW CONDITION

s.peek(); // should show 3

s.checkEmpty(s.isEmpty()); // false

s.Display();

s.SearchFor(2);

s.SearchFor(10);

s.Count();

cout << endl;

cout << "Deleting elements and checking for underflow" << endl;

s.pop(); // 2 1

s.pop(); // 1

s.pop(); // Empty

s.pop(); // UNDERFLOW CONDITION

s.peek(); // doesnt work on empty stack

s.checkEmpty(s.isEmpty()); // true

s.Display();

s.Count();

cout << endl;

cout << "Checking reset function" << endl;

s.push(1);

s.push(2);

s.push(3);

s.Display();

s.Count();

cout << "\n";

s.ClearStack();

s.checkEmpty(s.isEmpty());

s.Display();

s.Count();

cout << endl;

return 0;

}