Yes — **a reference to a pointer** (TreeNode\*& root) means you’re giving the function **access to the original pointer**, not a copy. It’s like giving someone your actual address, not just a copy of your key. So if they change the pointer, it changes **outside** the function too.

Let’s go step by step with a **memory analogy. We have to only pass root not &root**

If we’re passing an array to a function like search(int arr[], int n, int target), is it **call by reference**? And if so, why don’t we use a pointer explicitly like int\* arr?

**✅ Short Answer:**

Yes! When you pass an array like int arr[], **it is actually passed as a pointer** — so it's already "call by reference" for the array's elements.

You can think of this:

cpp

CopyEdit

int arr[] ≡ int\* arr

So both:

cpp

CopyEdit

int search(int arr[], int n, int target)

and

cpp

CopyEdit

int search(int\* arr, int n, int target)

are **exactly the same in C++** under the hood.

**💡 Why we don’t always write \*arr?**

* Writing int arr[] just makes it **clearer** that you're working with an array, not just any pointer.
* It's about **readability**, especially in function declarations.
* But functionally, arr[] and \*arr are treated **identically** by the compiler.

**📌 Important Notes:**

| **Topic** | **Explanation** |
| --- | --- |
| Array passed to function | Passed by reference (via pointer) |
| Change in function reflects outside | ✅ Yes, because it uses the original memory |
| Do we need & or \*? | ❌ No need — arrays automatically decay to pointers |

**🧠 Example to prove it:**

cpp

CopyEdit

void change(int arr[], int n) {

arr[0] = 100; // Modifies the original array

}

int main() {

int a[] = {1, 2, 3};

change(a, 3);

cout << a[0]; // Output: 100

}

Even without using & or \*, the original array **is modified** — proving it's call by reference via pointer.

**✅ Summary:**

| **Your Doubt** | **Answer** |
| --- | --- |
| Is arr[] passed by reference? | ✅ Yes, passed as a pointer (reference) |
| Do we need \*arr instead? | ❌ Not needed — both are same in function parameter |
| Why doesn’t it need &? | Arrays **automatically decay to pointers** when passed to functions |

## ✅ Full C++ Code (Dynamic Array + Pointers + Search)

cpp

CopyEdit

#include <iostream>

using namespace std;

class Solution {

public:

int search(int\* arr, int n, int target) {

for (int i = 0; i < n; i++) {

if (\*(arr + i) == target) { // same as arr[i]

return i;

}

}

return -1;

}

};

int main() {

int n;

cout << "Enter size of array: ";

cin >> n;

// Dynamically allocate array using new

int\* arr = new int[n];

cout << "Enter " << n << " elements:\n";

for (int i = 0; i < n; i++) {

cin >> \*(arr + i); // same as arr[i]

}

int target;

cout << "Enter number to search: ";

cin >> target;

Solution sol;

int result = sol.search(arr, n, target);

if (result != -1)

cout << "Element found at index: " << result << endl;

else

cout << "Element not found." << endl;

// Free the dynamically allocated memory

delete[] arr;

return 0;

}

## 🧠 What You’re Learning Here:

| **Concept** | **Explanation** |
| --- | --- |
| int\* arr = new int[n]; | Dynamically allocates an array |
| \*(arr + i) or arr[i] | Access elements via pointer |
| delete[] arr; | Frees dynamically allocated memory |
| int\* arr as parameter | Accepts pointer to array (call by reference) |
| No STL used | Like in C, but in C++ style |

## 🧾 Sample Output:

yaml

CopyEdit

Enter size of array: 6

Enter 6 elements:

7 8 3 6 45 8

Enter number to search: 45

Element found at index: 4

### 💡 Tips:

* In C, you'd use malloc() and free() instead of new and delete[]
* Pointers help in understanding memory deeply — very useful in interviews!

Would you like this same version:

* 🧠 In pure **C language using malloc()**?
* 🧠 With visualization of how memory looks using pointers?

Let me know, I’ll draw a full picture of what’s going on!

### Twoptr

**💡 Two Pointer in for Loop:**

cpp

CopyEdit

for (int i = 0, j = n - 1; i < j; ) {

// logic

if (arr[i] + arr[j] < target)

i++;

else

j--;

}

Let’s break it into 3 parts:

| **Part** | **Code** | **Meaning** |
| --- | --- | --- |
| Initialization | int i = 0, j = n-1 | Starts i from beginning and j from end |
| Condition | i < j | Loop continues while i < j |
| Update | *empty!* | Update is done **inside the loop body** |

So yes, it's **still a valid for loop**, just that we decided to leave the **update part empty** and control pointer movement from **inside** based on conditions. ✅

**🧠 Why use this format?**

* for loop groups all control variables (i, j) in one line
* Makes it clear what your loop is doing (two pointers: start and end)
* Keeps logic clean and easy to understand in some cases

**✅ Equivalent while version:**

cpp

CopyEdit

int i = 0, j = n - 1;

while (i < j) {

if (arr[i] + arr[j] < target)

i++;

else

j--;

}

Same logic! It’s just your choice — both are fine.