what is the diff between p = &n and \*p = 400 in c pionter (p vs \*p)

ChatGPT

In C programming, when dealing with pointers, it's essential to understand the difference between the address a pointer is holding and the value at that address. Let's break down your question:

p = &n;

Here, p is a pointer, and &n is the address of the variable n. When you use p = &n;, you are assigning the address of n to the pointer p. After this statement, p holds the address where n is stored in memory.

\*p = 400;

In this statement, \*p refers to the value stored at the address that p is pointing to. By using \*p = 400;, you are assigning the value 400 to the location in memory that p points to. If p was previously set to the address of n (as in the first statement), then this effectively sets n to 400.

In short:

p refers to the address the pointer holds.

\*p refers to the value at the address the pointer holds.

So, the key distinction is between the pointer itself (p) and the value at the address the pointer is pointing to (\*p).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In C, an array is NOT a pointer, the variables we declare as arrays do not hold a memory address.

When we declare an array, we use a name to refer to it, but it is only a name. Array names are identifiers that identify the entire array object. They are not pointers to anything. That is why we can not change the value of a 'variable’ which is an array.

ChatGPT

You are correct. The relationship between arrays and pointers in C is nuanced and often misunderstood.

Array is NOT a Pointer:

As you pointed out, when we declare an array, we're allocating a contiguous block of memory to hold the elements of the array. The array identifier (or the name) directly refers to this block of memory. It does not contain an address like a pointer variable does. Thus, you cannot assign a new address to an array identifier.

c

Copy code

int arr[5];

// arr = someOtherAddress; // This is ILLEGAL

Array-to-Pointer Decay:

However, in many contexts, the name of the array "decays" into a pointer to its first element. This is why arrays and pointers often seem interchangeable in C, especially when used as function arguments. But it's crucial to remember that this decay is a one-way conversion. While you can treat an array as a pointer in certain situations, you can't treat a pointer as an array.

c

Copy code

void func(int \*ptr) {

// Here ptr can point to an array's first element

}

int main() {

int arr[5];

func(arr); // Here, 'arr' decays to a pointer to its first element

return 0;

}

Size Differences:

The difference between arrays and pointers becomes apparent when using the sizeof operator. For an array, sizeof will return the total amount of memory the array occupies. For a pointer, sizeof will return the size of the pointer variable itself (which is typically 4 bytes on 32-bit systems and 8 bytes on 64-bit systems, though this can vary based on the architecture).

c

Copy code

int arr[5];

int \*ptr = arr;

printf("%zu\n", sizeof(arr)); // Prints total size of the array (5 times the size of an int)

printf("%zu\n", sizeof(ptr)); // Prints size of the pointer variable

Immutability of Array Name:

As you mentioned, you cannot change the "value" of an array, which means you can't make the array name point to a different block of memory. However, a pointer can be reassigned to point to different memory locations.

In summary, while arrays and pointers in C can sometimes be used in seemingly interchangeable ways due to array-to-pointer decay, they are fundamentally different. Understanding this distinction is crucial for C programming.

User

But, you can still use the name of the array in your code, and its value will be… the address of the first element of the array. Wait… WAT?

ChatGPT

Yes, this is one of the nuances that often confuses new C programmers.

Let's break it down:

The Name of the Array:

When you use the name of an array in most contexts, it "decays" into a pointer to the first element of the array. This essentially means it gives you the memory address of the first element.

c

Copy code

int arr[5];

printf("%p\n", arr); // Prints the memory address of the first element of 'arr'

Dereferencing:

If you dereference this "decayed" pointer, you get the value of the first element of the array.

c

Copy code

arr[0] = 10;

printf("%d\n", \*arr); // Prints '10', the value of the first element of 'arr'

Notice how \*arr gives the same value as arr[0]. This is a direct consequence of the array-to-pointer decay phenomenon.

Pointer Arithmetic:

Since the array name can decay to a pointer, you can perform pointer arithmetic on it.

c

Copy code

arr[1] = 20;

printf("%d\n", \*(arr + 1)); // Prints '20', the value of the second element of 'arr'

Here, \*(arr + 1) is equivalent to arr[1].

The Exception:

There are contexts where the array name doesn't decay to a pointer. The most notable is the sizeof operator. When you use sizeof(arr), you get the total size of the array in bytes, not the size of a pointer.

The key takeaway is that while an array is not a pointer, the array name, when used in most expressions, decays to a pointer pointing to the first element of the array. This behavior allows for the syntactic similarities we see between array indexing and pointer dereferencing/arithmetic.