Pointers, Arrays, and Strings in C

Shaan Fulton

July 4, 2025

Pointers, arrays, and strings are all effectively the exact same thing. Yes: there is some overhead in arrays and strings that tells C to allocate space, but we can tack this on manually to make pointers work like arrays and strings.

An array is literally just us saying: "Hey! I'd like to store 4 integers in a row. You know how long an integer is... yes, that means you'll give me 16 bytes, 4 bytes for each."

If we're storing this data as a local variable that will disappear when our function frame disappears, this means moving our stack pointer 16 bytes up so that the next things we stack don't collide. If we're storing this data in our heap, this means finding a nice spot of 16 bytes that's free for us to dump our data in and then marking it as so by updating our free space list.

Using Pointers

A pointer is just a variable that holds a memory address, typically the address of another variable. In C, you declare a pointer by putting a `*` before the variable name:

```
int *p;
```

You can point it at an integer like this (where & pulls the memory address from x):

```
int x = 10;
p = &x; // now p points to x
```

To get or set the value pointed to, use the dereference operator (`*`):

```
*p = 20; // x is now 20
```

We can also have pointers to pointers (double `**`), or even more levels if you want. This is just a pointer that stores the address of another pointer. Why? Mostly for when you want to change the pointer itself in a function (e.g., updating a pointer to point somewhere else), or for dealing with lists of pointers (like `char **argv` in `main()`).

```
int y = 5;
int *a = &y;
int **b = &a; // b points to a, which points to y
**b = 7; // y is now 7
```

Making a List in the Heap With Pointers

Arrays declared like `int arr[10];` live on the stack, but sometimes you want to create arrays that last longer, or whose size you only know at runtime. That's where heap allocation comes in, with the `malloc()` function, which looks for space in the heap and allocates it:

```
int *list = malloc(sizeof(int) * 4); // allocates space for 4 ints
```

You can use this pointer exactly like an array:

```
list[0] = 42;
list[1] = 77;
```

Under the hood, `list[1]` is just shorthand for `*(list + 1)`, which means "go to the address in `list`, move over by one `int`, or 4 bytes, and access the value there." The pointer knows that it needs to jump and `int` because we actually typed the pointer as an integer.

Don't forget to free heap memory when you're done:

```
free(list);
```

More details on the heap are found in `memory-architecture`. The heap is generally much less efficient than the stack because we actively need to search through our list of free spaces to find a slot big enough for the data we're looking to store. Only use the heap if you really need the data to last outside your function frame, or you know you'll need to grow and shrink it later.

Make Default Arrays and Strings In Stack or Static

If you know the size at compile time, C makes arrays super easy:

```
int nums[4] = {1, 2, 3, 4};
```

Strings are just arrays of `char` ending with a special zero byte called the "null terminator" (`'\0'`). You can make strings like:

```
char msg[6] = "Hello";
```

Or, let C count the length for you (it'll make the array 6 bytes to fit "Hello\0"):

```
char msg[] = "Hello";
```

When you pass arrays or strings to functions, what you're really passing is a pointer to the first element! So, this works seamlessly:

```
void printNums(int *nums, int length) {
    for (int i = 0; i < length; i++) {
        printf("%d\n", nums[i]);
    }
}
```