Tip 16: All the pointer arithmetic you need to know

Ben Klemens

2 November 2011

level: basic
purpose: save you reading and cognitive effort

[A tip every other day on POSIX and C. Start from the tip intro page[1].]

Here's my theory of why I like C despite the common wisdom that it is terrible: I didn't learn C in a classroom. When you learn this stuff on the streets, you skip the parts of the textbook that that aren't necessary for survival, at which point you have a pretty lean and fun language.

Kernighan & Ritchie (and by extension, lots of standard C textbooks) use a lot of paper expressing love for how pointer arithmetic works. If you're reading this blog, then you need none of it.

It's really amusing stuff. An array element consists of a base position plus an offset. This was all designed in the 1970s, so implementing an array as a block of memory and its elements as offsets made sense to the sort of people who spend their mornings writing assembly code. But it was also sort of mathematically clean and appealing. You could declare a pointer `double *p`; then that's our base, and you can use the offsets from that base as an array: the contents of the first element is `p[0]`, the contents of the second `p[1]`, et cetera. So we've implemented the distinction between data and the location of data, and got arrays for free in the process.

Or you could just write the base plus offset directly and literally, via a form like `(p+1)`. As your textbooks will tell you, this is valid C, and in fact `p[1]` is exactly equivalent to `*(p+1)`, which explains why the first element in an array is `p[0] == *(p+0)`. K & R spend about six pages on this stuff [sections 5.4 and 5.5].

This is a bit like how Latin is taught, versus every other language. In your Spanish class, you start off with usage. For arrays, something like:

- Declare pointers either via the dynamic form, `double *p` or the static form like `double p[100]`. We'll worry about the distinction later.

- In either case, the nth array item is `p[n]`. Don't forget that the first item is zero, not one; it can be referred to with the special form `p[0] == *p`.

[1] `http://modelingwithdata.org/arch/00000049.htm`

- If you need the address of the nth element (not its actual value), use the ampersand: `&p[n]`. Of course, the zeroth pointer is just `&p[0] == p`.

Weeks later, when you have the basic forms down, your Spanish class teaches you about the difference between different types of future tense. Meanwhile, in Latin class, you *start* with learning about the ablative case, and then learn Latin as an application of all that grammar you saw. Jumping the metaphor again, your average C textbook opens the section on pointers with a diagram showing a series of memory registers, while your typical Python textbook never gets into the details of implementation at all.

Since this is a tip-a-day blog, and you're probably reading *don't read the section on pointer arithmetic* as more a rant than a tip, I'll throw in one nice trick: you don't need an index for `for` loops that step through an array. Here, we use a spare pointer that starts at the head of a list, and then step through the array with `p++` until we hit the `NULL` marker at the end.

```
#include <stdio.h>

int main(){
    char *list[] = {"first", "second", "third", NULL};
    for (char **p=list; *p != NULL; p++){
        printf("%s\n", p[0]);
    }
}
```

It's nice that we don't have to bother with a counter, as we would in any other language, but then, it's not much of a payoff to six pages of pointer arithmetic lessons. Exercise: how would you implement this if you didn't know about `p++`?

Oh, and as for bit-shifting operators, like bitwise XOR and shift-register-left, I have written tens of thousands of lines of C code and used one maybe once. Just skip those sections entirely.