MAIN
PREVIOUS
NEXT
HELP
Follow

Exercise 15: Pointers Dreaded Pointers

Pointers are famous mystical creatures in C that I will attempt to demystify by teaching you the vocabulary used to deal with them. They actually aren't that complex, it's just they are frequently abused in weird ways that make them hard to use. If you avoid the stupid ways to use pointers then they're fairly easy.

To demonstrate pointers in a way we can talk about them, I've written a frivolous program that prints a group of people's ages in three different ways:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // create two arrays we care about
    int ages[] = {23, 43, 12, 89, 2};
    char *names[] = {
        "Alan", "Frank",
        "Mary", "John", "Lisa"
    };

    // safely get the size of ages
    int count = sizeof(ages) / sizeof(int);
    int i = 0;

    // first way using indexing
    for(i = 0; i < count; i++) {
        printf("%s has %d years alive.\n",
                names[i], ages[i]);
    }

    printf("---\n");

    // setup the pointers to the start of the arrays
    int *cur_age = ages;
    char **cur_name = names;

    // second way using pointers
    for(i = 0; i < count; i++) {
        printf("%s is %d years old.\n",
                *(cur_name+i), *(cur_age+i));
    }
```

```
    printf("---\n");

    // third way, pointers are just arrays
    for(i = 0; i < count; i++) {
        printf("%s is %d years old again.\n",
                cur_name[i], cur_age[i]);
    }

    printf("---\n");

    // fourth way with pointers in a stupid complex way
    for(cur_name = names, cur_age = ages;
            (cur_age - ages) < count;
            cur_name++, cur_age++)
    {
        printf("%s lived %d years so far.\n",
                *cur_name, *cur_age);
    }

    return 0;
}
```

Before explaining how pointers work, let's break this program down line-by-line so you get an idea of what's going on. As you go through this detailed description, try to answer the questions for yourself on a piece of paper, then see if what you guessed was going on matches my description of pointers later.

ex15.c:6-10
: Create two arrays, `ages` storing some `int` data, and `names` storing an array of strings.

ex15.c:12-13
: Some variables for our `for-loops` later.

ex15.c:16-19
: You know this is just looping through the two arrays and printing how old each person is. This is using `i` to index into the array.

ex15.c:24
: Create a pointer that points at `ages`. Notice the use of `int *` to create a "pointer to integer" type of pointer. That's similar to `char *`, which is a "pointer to char", and a string is an array of chars. Seeing the similarity yet?

ex15.c:25
: Create a pointer that points at `names`. A `char *` is already a "pointer to char", so that's just a string. You however need 2 levels, since `names` is 2-dimensional, that means you need `char **` for a "pointer to (a pointer to char)" type. Study that too, explain it to yourself.

ex15.c:28-31
: Loop through `ages` and `names` but instead use the pointers *plus an offset of i*. Writing `*(cur_name+i)` is the same as writing `name[i]`, and you read it as "the value of (pointer `cur_name` plus i)".

ex15.c:35-39
: This shows how the syntax to access an element of an array is the same for a pointer and an array.

ex15.c:44-50
: Another admittedly insane loop that does the same thing as the other two, but instead it uses various pointer arithmetic methods:

    ex15.c:44
    : Initialize our `for-loop` by setting `cur_name` and `cur_age` to the beginning of the `names` and `ages` arrays.

    ex15.c:45
    : The test portion of the `for-loop` then compares the *distance* of the pointer `cur_age` from the start of `ages`. Why does that work?

    ex15.c:46
    : The increment part of the `for-loop` then increments both `cur_name` and `cur_age` so that they point at the *next* element of the `name` and `age` arrays.

    ex15.c:48-49
    : The pointers `cur_name` and `cur_age` are now pointing at one element of the arrays they work on, and we can print them out using just `*cur_name` and `*cur_age`, which means "the value of wherever `cur_name` is pointing".

This seemingly simple program has a large amount of information, and the goal is to get you to attempt figuring pointers out for yourself before I explain them. *Don't continue until you've written down what you think a pointer does.*

What You Should See

After you run this program try to trace back each line printed out to the line in the code that produced it. If you have to, alter the `printf` calls to make sure you got the right line number.

```
$ make ex15
cc -Wall -g    ex15.c   -o ex15
$ ./ex15
Alan has 23 years alive.
Frank has 43 years alive.
Mary has 12 years alive.
John has 89 years alive.
Lisa has 2 years alive.
---
Alan is 23 years old.
Frank is 43 years old.
Mary is 12 years old.
John is 89 years old.
Lisa is 2 years old.
---
Alan is 23 years old again.
Frank is 43 years old again.
Mary is 12 years old again.
John is 89 years old again.
Lisa is 2 years old again.
---
Alan lived 23 years so far.
Frank lived 43 years so far.
Mary lived 12 years so far.
John lived 89 years so far.
Lisa lived 2 years so far.
$
```

Explaining Pointers

When you type something like `ages[i]` you are "indexing" into the array `ages`, and you're using the number that's held in `i` to do it. If `i` is set to 0 then it's the same as typing `ages[0]`. We've been calling this number `i` an "index" since it's a location inside `ages` that we want. It could also be called an "address", that's a way of saying "I want the integer in `ages` that is at address `i`".

If `i` is an index, then what's `ages`? To C `ages` is a location in the computer's memory where all of these integers start. It is *also* an address, and the C compiler will replace anywhere you type `ages` with the address of the very first integer in ages. Another way to think of `ages` is it's the "address of the first integer in ages". But, the trick is `ages` is an address inside the *entire computer*. It's not like `i` which was just an address inside `ages`. The `ages` array name is actually an address in the computer.

That leads to a certain realization: C thinks your whole computer is one massive array of bytes. Obviously this isn't very useful, but then C layers on top of this massive array of bytes the concept of *types* and *sizes* of those types. You already saw how this worked in previous

exercises, but now you can start to get an idea that C is somehow doing the following with your arrays:

> - Creating a block of memory inside your computer.
> - "Pointing" the name `ages` at the beginning of that block.
> - "Indexing" into the block by taking the base address of `ages` and getting the element that's `i` away from there.
> - Converting that address at `ages+i` into a valid `int` of the right size, such that the index works to return what you want: the int at index `i`.

If you can take a base address, like `ages`, and then "add" to it with another address like `i` to produce a new address, then can you just make something that points right at this location all the time? Yes, and that thing is called a "pointer". This is what the pointers `cur_age` and `cur_name` are doing. They are variables pointing at the location where `ages` and `names` live in your computer's memory. The example program is then moving them around or doing math on them to get values out of the memory. In one instance, they just add `i` to `cur_age`, which is the same as what it does with `array[i]`. In the last `for-loop` though these two pointers are being moved on their own, without `i` to help out. In that loop, the pointers are treated like a combination of array and integer offset rolled into one.

A pointer is simply an address pointing somewhere inside the computer's memory, with a type specifier so you get the right size of data with it. It is kind of like a combined `ages` and `i` rolled into one data type. C knows where pointers are pointing, knows the data type they point at, the size of those types, and how to get the data for you. Just like `i` you can increment them, decrement them, subtract or add to them. But, just like `ages` you can also get values out with them, put new values in, and all the array operations.

The purpose of a pointer is to let you manually index into blocks or memory when an array won't do it right. In almost all other cases you actually want to use an array. But, there are times when you *have* to work with a raw block of memory and that's where a pointer comes in. A pointer gives you raw, direct access to a block of memory so you can work with it.

The final thing to grasp at this stage is that you can use either syntax for most array or pointer operations. You can take a pointer to something, but use the array syntax for accessing it. You can take an array and do pointer arithmetic with it.

Practical Pointer Usage

There are four primary useful things you do with pointers in C code:

- Ask the OS for a chunk of memory and use a pointer to work with it. This includes strings and something you haven't seen yet, `structs`.
- Passing large blocks of memory (like large structs) to functions with a pointer so you don't have to pass the whole thing to them.
- Taking the address of a function so you can use it as a dynamic callback.
- Complex scanning of chunks of memory such as converting bytes off a network socket into data structures or parsing files.

For nearly everything else you see people use pointers, they should be using arrays. In the early days of C programming people used pointers to speed up their programs because the compilers were really bad at optimizing array usage. These days the syntax to access an array vs. a pointer are translated into the same machine code and optimized the same, so it's not as necessary. Instead, you go with arrays every time you can, and then only use pointers as a performance optimization if you absolutely have to.

The Pointer Lexicon

I'm now going to give you a little lexicon to use for reading and writing pointers. Whenever you run into a complex pointer statement, just refer to this and break it down bit by bit (or just don't use that code since it's probably not good code):

`type *ptr`
: "a pointer of type named ptr"

`*ptr`
: "the value of whatever ptr is pointed at"

`*(ptr + i)`
: "the value of (whatever ptr is pointed at plus i)"

`&thing`
: "the address of thing"

`type *ptr = &thing`
: "a pointer of type named ptr set to the address of thing"

`ptr++`
: "increment where ptr points"

We'll be using this simple lexicon to break down all of the pointers we use from now on in the book.

Pointers Are Not Arrays

No matter what, you should never think that pointers and arrays are the same thing. They are not the same thing, even though C lets you work with them in many of the same ways. For example, if you do `sizeof(cur_age)` in the code above, you would get the size of the *pointer*, not the size of what it points at. If you want the size of the full array, you have to use the array's name, `age` as I did on line 12.

TODO: expand on this some more with what doesn't work on both the same.

How To Break It

You can break this program by simply pointing the pointers at the wrong things:

- Try to make `cur_age` point at `names`. You'll need to use a C cast to force it, so go look that up and try to figure it out.
- In the final `for-loop` try getting the math wrong in weird ways.
- Try rewriting the loops so they start at the end of the arrays and go to the beginning. This is harder than it looks.

Extra Credit

- Rewrite all the array usage in this program so that it's pointers.
- Rewrite all the pointer usage so they're arrays.
- Go back to some of the other programs that use arrays and try to use pointers instead.
- Process command line arguments using just pointers similar to how you did `names` in this one.
- Play with combinations of getting the value of and the address of things.
- Add another `for-loop` at the end that prints out the

addresses these pointers are using. You'll need the `%p` format for `printf`.
- Rewrite this program to use a function for each of the ways you're printing out things. Try to pass pointers to these functions so they work on the data. Remember you can declare a function to accept a pointer, but just use it like an array.
- Change the `for-loops` to `while-loops` and see what works better for which kind of pointer usage.