ECE 264: Advanced C Programming

Week2 Lecture Notes 6/17/19 - 6/21/19

1 Topics

This week we discussed pointers, arrays, pointer arithmetic, and dynamic memory allocation.

- 6/17 and 6/18: Pointers
- 6/19: Pointer arithmetic
- 6/20: Arrays of strings, sizeof operator
- 6/21: Dynamic memory allocation, quick valgrind demo, quick review of PA02.

2 Pointers

What is a pointer? We've seen a bunch of different types that C has: `int, float, short`, etc. We've even seen how to create our own types by creating structures. But there is a whole category of types that we have not looked at: *pointers*. A pointer type looks like `<typename> *`, and we read it as "pointer to `<typename>`'. So, for example:

```
int * p;
```

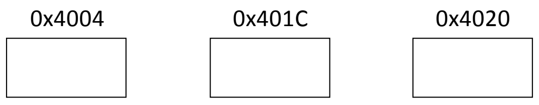
is a variable named `p` whose type is "pointer to int' (Important note: the variable we declared here is `p`, not `*p`)

So what does it mean to be a pointer to an int? To understand this, it's helpful to have a picture of what's going on in memory.

2.1 How should we think about memory?

We've already talked a bit about how programs are laid out in memory, with our discussion of the program stack. The important thing to understand about memory is that your program "thinks' in terms of memory locations, and every memory location has an *address*.

Think about memory as a bunch of boxes. Each box is a location where you can store some data. Each box has an *address* (sort of like how each house on a street has an address). A particular address refers to exactly one box (memory location). So for example, we could think of three memory locations, each with their own address:



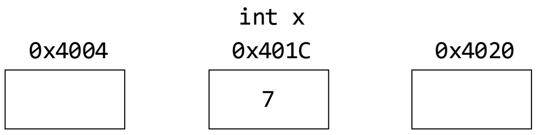
An important thing to note here is that *your computer only thinks about memory in terms of addresses*: it's like a GPS device that can only navigate to places based on their street addresses. The only thing the computer understands is memory addresses. If I want to store data in a memory location, I have to use an address. If I want to retrieve data from a memory location, I have to use its address.

2.2 What is a variable?

Humans are not so good at remembering addresses (quick: what's latitude and longitude of your hometown?) So in computer programs, we use *variables* as "handles' to let us talk about memory locations without talking about addresses. When we create a global variable:

```
int x = 7;
```

Your program chooses a particular memory location, and gives it an alternate name of `x`. So, for example, it might decide that memory location `0x401c` will be called `x`. Your program remembers this mapping, so that whenever you talk about `x`, the program generates code that talks about memory location `0x401c`. (Think of this mapping like an address book: it lets you talk about a particular street address not as, say "465 Northwestern Ave." but instead as "the EE building" — it's an alternate name for a particular location)



All variables in your program are just names given to memory locations (the details are a little trickier for variables that are local to a function, but the basic principle is the same). This means that *every variable in your program* also has an address that it is associated with.

What's interesting is that C provides a way to get at that address, using the *address of* operator, `&`. In our example:

```
&x //would return the address 0x401c
```

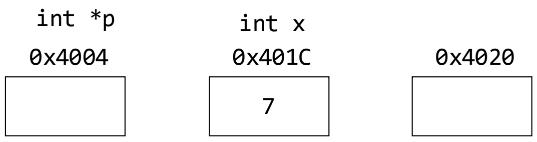
Now we're ready to answer the question: what is a pointer?

2.3 What is a pointer (take 2)?

A pointer is a data type that holds an address. The data stored for a pointer is always an address, and the type of that pointer (e.g., a pointer to an int) tells us what kind of data is stored at that address. So suppose we create our pointer again:

```
int * p;
```

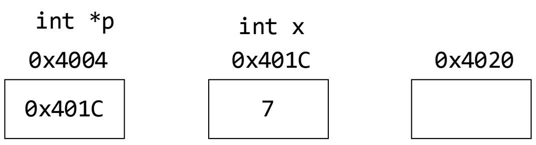
Remember, `p` is a variable, and all variables are just names given to addresses. Let's assume that our program has decided that `p` is the name of address `0x4004`.



`p` has type `int *`, which means that it holds the address of a memory location that stores an integer. Where might we get that sort of address from? Well, we can use the `&` operator!

```
p = &x;
```

Which stores the *address of* `x` in `p`:



Colloquially, we say that this means `p` points to `x`. Next, we'll see what we can do with this address stored in `p`.

2.4 Using pointers

We saw that we can use pointers to store addresses of locations in memory. How can we use them?

The trick to pointers is that the operator `*` (the *dereference* operator) lets us access the memory location that the pointer points to (i.e., it lets us access the memory location at the address that is stored in the pointer):

```
int x = 7;
int * p = &x; //p now points to x
*p = 10; //this is the same as x = 10
int y = *p; //this is the same as y = x
```

The expression `*p` acts just like `x` wherever we use it. In fact, one way to think about pointers is they let you give alternate names to locations in memory. If a pointer `p` stores an address, `*p` is a name for that address in exactly the same way that a variable is a name for an address!

Pointers to things other than basic data types Pointers don't have to point to ints or floats or doubles. They can also point to data types you create!

```
typedef struct {
  float x;
  float y;
} Point;
Point p = {.x = 1.5, .y = 2.5};
Point * q = &p; //now you can use * q anywhere you use p
```

Why is this useful? with regular variables, once you create them, you name a memory location, but you can never change what memory location you're talking about. Pointers give you a way of creating a "dynamic" name — a name that you can use to talk about a memory location that can change depending on what you need to use it for:

```
int x = 7;
int * p = &x; //*p is now another name for x
int y = * p; //like saying y = x
p = &y; //*p is now another name for y
*p = 8; //like saying y = 8
```

One place this is especially useful is in writing functions. C functions are *pass by value*: when you pass an argument to a function, inside the function you are working on a *copy* of that argument. If you try to change the data inside the function, you're changing the copy, not the original data. The following implementation of a swap function doesn't work:

```
int a = 8;
int b = 10;
void swap(int x, int y) {
 int tmp = x;
 x = y;
 y = tmp;
}
void main() {
 swap(a, b); //a is still 8, b is still 10
}
```

Because inside swap, `x` and `y` are names for new memory locations that are holding *copies* of the data in `a` and `b`. When we swap them, `a` and `b` don't change. But we can use pointers to get around this problem. What if `x` and `y` held the addresses of `a` and `b`? Then `*x` and `*y` would be names for the same memory locations that `a` and `b` are. Changing them would change the values of `a` and `b`!

```
int a = 8;
int b = 10;
void swap(int * x, int * y) {
 int tmp = *x; //tmp = whatever is in the location x points to
 *x = *y;
 * y = tmp;
}
void main() {
 //remember, we have to pass in addresses now, not ints
 swap(&a, &b); //a is now 10, b is now 8
}
```

2.5 Chains of pointers

Pointers can point to any data type — even pointers!

```
int x = 7;
int * p = &x; //p points to x; *p is the same as x
int * * q; //q is a pointer to a pointer to an int
q = &p; //q points to p
```

In this example, `*q` is the same as `p`. `*(*q)` is the same as `*p` which is the same as `x`.

3 Pointer arithmetic and Arrays

3.1 Pointer arithmetic

What does `*(p + 1)` mean?

If `p` is a `int *`, it means "access one integer past whatever p points to." Remember that because `p` is a pointer, it holds the address of a location in memory. Because `p` is an `int` pointer, that location in memory holds an integer. When we add 1 to the address, the compiler interprets this as "the address of the next integer." In other words, we will add 4 to the address in `p` (because integers take up 4 bytes). If `p` pointed to `doubles`, the compiler would interpret this as "the address of the next double" and add 8 to the address (because doubles take up 8 bytes).

We can put numbers other than 1 there. For example, `p + 3` will give the address of the integer that is three integers past whatever `p` currently points to. `p - 1` will give the address of the integer right before whatever `p` points to.

3.2 Arrays

Another data type!

Array data types

Arrays of ints, arrays of structs

Arrays of chars: the C way to represent strings

How do arrays work? They're weird — they work a little bit like pointers:

`int a[10] //a is an array of 10 integers`

Can access the first integer with `a[0]`

Can access the second integer with `a[1]`

etc.

Note that these 10 integers are *guaranteed* to be next to each other in memory

So what's `a` itself? It actually refers to the first location of the `a` array. Can see this by printing `a`, printing the address of `a`, printing the address of `a[0]`— all the same!

`a[0]` really means: `*(a + 0)`

`a[1]` really means: `*(a + 1)`

And so on.

Can use pointers to represent arrays, too: `int * p = a;`

(aside: internally, your C compiler thinks that the type of`a` is `int *`! If you try `int * p = &a`, your code will work, but the compiler will complain)

Now we can use `p` the same way we would use `a`: `p[0]` really means `*(p + 0)` which is the same as `*(a + 0)` which is `a[0]` `p[1]` really means `*(p + 1)` which is the same as `*(a + 1)` which is `a[1]`

4 Dynamic memory allocation

What if we don't know how big we want an array to be? One way to do this is to use *variable length arrays*, but those are not always supported, and using them is potentially very dangerous (we don't allow you to use them, in fact: -*Wvla* gives a warning if you try to use them)

We can instead use the *heap* — that other space in memory where we can store data. The basic way we interact with the heap is to ask the program to either "give us X bytes of data from the heap that we can use to store data" or "take back this data so that it can be reused for something else."

`void * malloc(size_t size) // give me size bytes of data from the heap, return the address of the first byte of that chunk`

`free (void * ptr) //take back the chunk of memory where ptr points to the beginning of that chunk`

So, remembering that pointers can be used as arrays, here's how we could allocate an array of `N` integers when we don't know what `N` is at compile time:

`int * arr = malloc(N * sizeof(int));`

What's happening here?

1. find 40 bytes of memory on the heap.
2. reserve it for the program's use. Means that no other call to `malloc` will return any of that part of memory unless you call `free`
3. return the address of the beginning of the chunk

Note that the chunk is guaranteed to be 40 consecutive bytes in memory. `arr` points to the beginning of the chunk. So we can treat this chunk of memory just like an array:

`arr[0], arr[5]`, etc.

When we're done with the memory, we can tell the program that we're done with it:

`free(arr);` Now when we call `malloc()` again, we might get back the same memory. [Note: `arr` still points to that location in memory — this is potentially dangerous! Always a good idea to null-out a pointer when you free the data it points to]

4.1 Memory leaks

Remember that calling `malloc` grabs a block of memory in the heap, and `malloc` guarantees that this block of memory won't be used by anyone else:

`int * p = malloc(10 * sizeof(int)); //grab 40 bytes of memory`

To give a block of memory back to the heap, so that it can be reused by something else, we call `free`:

`free(p); //release 40 bytes of memory back to the heap.`

So what happens if you call `malloc` inside a function?

```
void foo(int N) {
 int * p = malloc(N * sizeof(int)); //allocate an array of N integers
 ... //code to do something with the array
 return;
}
```

The local variable `p` points to the block of memory we grab with `malloc`. But once we return from the function, the variable `p` goes away. We no longer have any way of *reaching* the block of memory we allocated: we have no way of getting to the address of that block. If we can't get the address of the block, we can't `free` it ? we can never give the memory back to the heap. This is a *memory leak.*

If we keep calling `foo`, we'll keep allocating new blocks of memory that we lose access to. These blocks can never be given back to the heap and will just sit around taking up space. In fact, if we call `foo` too many times, we could run out of memory!

Memory leaks are bugs in your program, though they may not always cause your program to break. What you need to do is make sure that whenever you are done using a block of memory you have allocated, you call `free` on it:

```
void foo(int N) {
 int * p = malloc(N * sizeof(int)); //allocate an array of N integers
 ... //code to do something with the array
 free(p); //free the array to avoid a memory leak
 return;
}
```

But finding memory leaks can be tricky: you have to free memory late enough that you're sure you're done with it, but *early* enough that you still have access to it. The previous example of a memory leak was an example of waiting too long to free memory: by the time we returned from `foo`, we didn't have a way of getting to the block of memory that we wanted to free. Here's an example where we need to be careful not to free memory too early:

```
int * foo(int N) {
 int * p = malloc(N * sizeof(int)); //allocate an array of N integers
 ... //code to do something with the array
 free(p); //THIS IS TOO EARLY!
 return p;
}
```

Why is it a problem to free the array in this example? Because we've returned the pointer to the array. That means we can't be sure that whoever *called* `foo` isn't going to use the array for something else. If we free the array in `foo`, we are returning memory that the program might not be done with. That's bad!

Some times it can be even harder to tell that whether it's safe to free memory:

```
int * * bar(int N) {
 int * p = malloc(N * sizeof(int)); //allocate an array of N integers
 int * * q = malloc(sizeof(int *)); //allocate space for an int*
 * q = p; //the box q points to now holds the address of the array
 return q; //return the address of the box q points to.
}
```

It's not safe to free the array(`p`) before returning from `bar`. Even though we're not returning a pointer to the array (like before, in `foo`), we're returning a pointer to a pointer to the array. Which means we can still reach the array, and freeing the array is unsafe. But it gets worse:

```
int * * i = bar(10); //i points to a box which points to the array
(* i)[5] = 12; //this sets the 6th element of the array.
(* i) = NULL; //now i points to a box which contains NULL
```

Once we execute the third line of code, we cannot reach the array any more! The array has leaked!

To fix this, we need to make sure to free the array before we make it unreachable:

```
int * * i = bar(10); //i points to a box which points to the array
(* i)[5] = 12; //this sets the 6th element of the array.
free(* i); //free the array
(* i) = NULL; //now i points to a box which contains NULL, but it?s OK
```

4.2 Valgrind

How can we find memory leaks? They're not really like other bugs. Some bugs show up at compile time because the compiler complains. Other bugs cause your program to crash. But memory leaks some times don't seem to affect your program at all! They can be hard to find. We can use a tool called *valgrind* to help find bugs. Consider the following program:

```
void main () {
 int * p = malloc(sizeof(int));
 int * q = malloc(4 * sizeof(int));
 int * r = malloc(16 * sizeof(int));
 free(p);
 free(r);
}
```

This program has a memory leak (we don't free `q` before the program exits). If we compile the program and run valgrind:

```
> gcc alloctest.c -g
```

```
> valgrind -tool=memcheck -leak-check=full a.out
```

We get the following output, telling us that we leaked 16 bytes. Valgrind also tells us which call to `malloc` caused the leak (line 5, which is the allocation for `q`)

```
==14189== Memcheck, a memory error detector
==14189== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==14189== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==14189== Command: a.out
==14189==
==14189==
==14189== HEAP SUMMARY:
==14189== in use at exit: 16 bytes in 1 blocks
==14189== total heap usage: 3 allocs, 2 frees, 84 bytes allocated
==14189==
==14189== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==14189== at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==14189== by 0x400523: main (alloctest.c:5)
==14189==
==14189== LEAK SUMMARY:
==14189== definitely lost: 16 bytes in 1 blocks
==14189== indirectly lost: 0 bytes in 0 blocks
==14189== possibly lost: 0 bytes in 0 blocks
==14189== still reachable: 0 bytes in 0 blocks
==14189== suppressed: 0 bytes in 0 blocks
==14189==
==14189== For counts of detected and suppressed errors, rerun with: -v
==14189== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```