

Lecture notes: Week 4

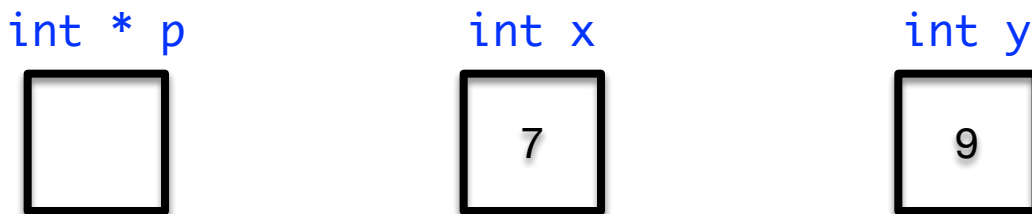
In this week, we covered:

- How to think about boxes vs. the contents of boxes
- Address-of (&) and dereference (*) operators
- Pointer types
- Pointers-to-pointers
- Pointers-to-structs
- Passing args to functions
- Void pointers
- Function pointers
- Memory allocation
- Dynamically allocating arrays
- Pointer arithmetic

Pointer details

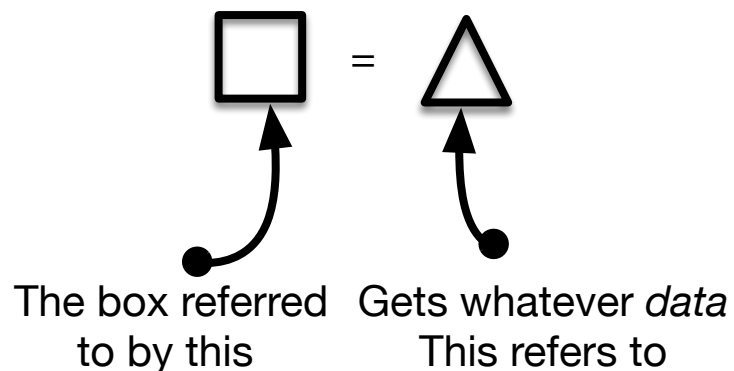
Whenever you create a variable in your program, no matter what its type, you get a box, and each box has an address:

```
int x = 7;  
int y = 9;  
int * p;
```



When we see an expression like `a = b`, what we're saying is "put a copy of whatever was *in* box `b` into box `a`". In other words, we're using the *box* called `a`, but the *value* of what was in box `b`. In compiler terms, when we're talking about the box, we're talking about an "L-value" (because it usually shows up on the left-hand-side of assignments) and when we're talking about what's *inside* the box, we're talking about an "R-value" (because it usually shows up on the right-hand-side of assignments).

This is important to keep in mind. It might seem pedantic and obvious, but it's important to know that this is true for *any* assignment expression, even if the left hand side is a complicated thing. So when we see




`p = &y`


We're putting the data (the *address of y*) in the *box* called p. P now points to y!
And now when we do something more complicated:

`*p = x`

We're finding the box referred to by `*p` (which happens to be the box called `y`) and putting the data referred to by `x` (the number 7) in it.

Address-of and dereference operators

`&`  We can think of the address-of (&) operator as an operation that says “give me the address of a box.” So if I have a box, and I take its address, I get back some location in memory. You can think of this as “Give me an arrow that points to the box.”

`*`  In contrast, the dereference (*) operator is an operation that says “instead of talking about this box, let me talk about *the location this box is pointing to*. You can think of this as “Follow the arrow, and talk about the box the arrow is pointing to.”

One thing to remember is that *every box has a type*: whenever your program is thinking about a location in memory, it is thinking it has some type. So how do * and & change types?

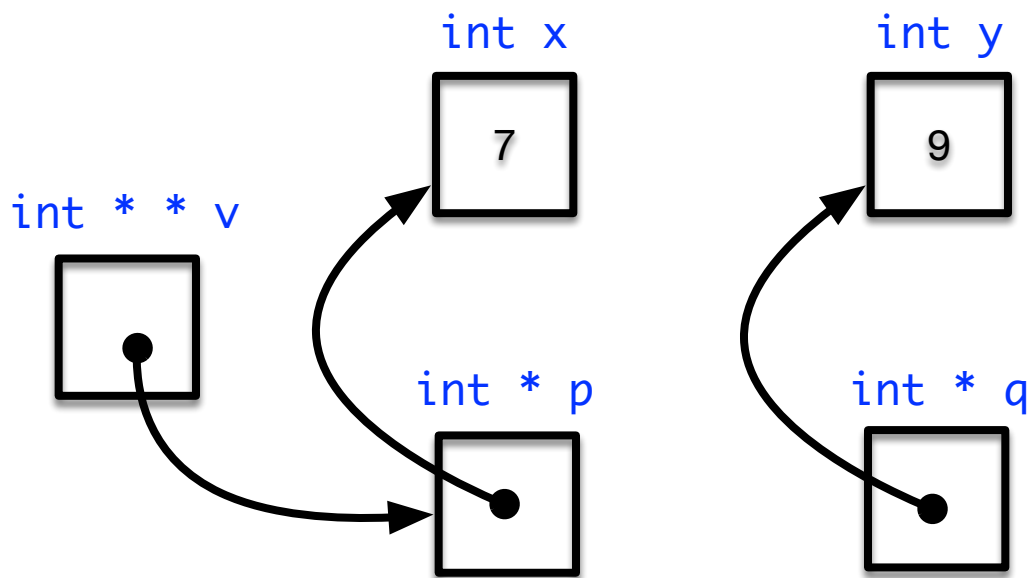
Well, if I have a `△` of type `T`, then if I take the *address of it*: `&△`, then I have an arrow pointing to a box of type `T`. I have a *pointer to T*, or a `T *`. In other words, applying an address-of operator *adds a * to the type*.

In contrast, if I have a `△` of type `T *`, then if I *dereference it*: `* △`, then I follow the arrow that `△` has in it to get to another box, and that box has type `T`. In other words, applying a dereference operator *removes a * from the type*.

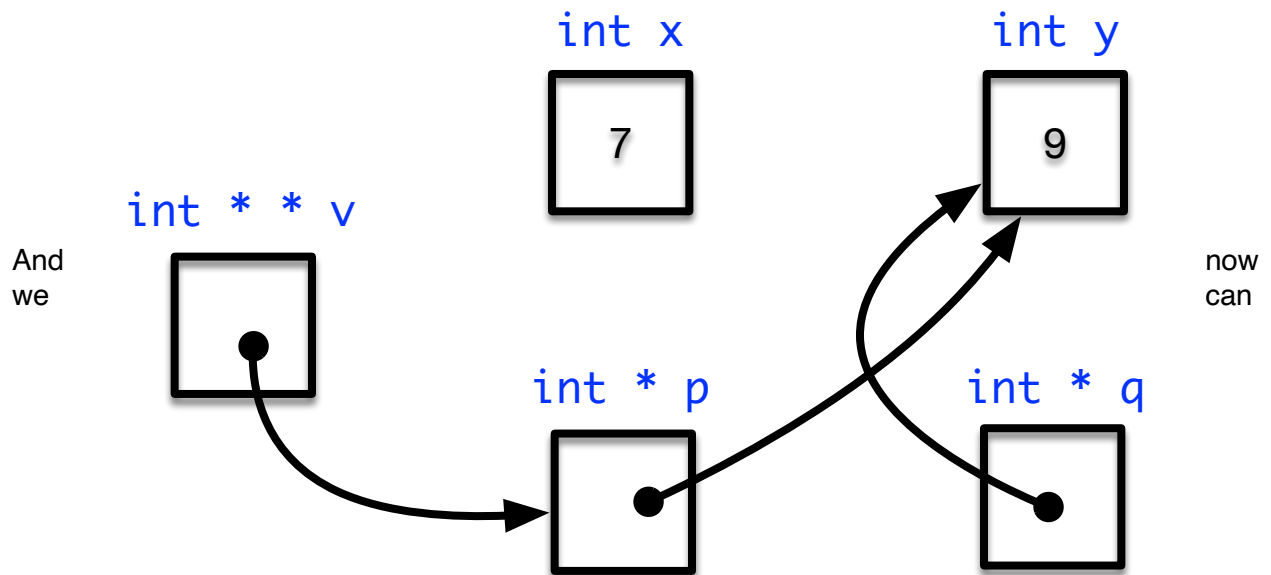
Pointers to pointers

Note that there's nothing that says that I can only point to things like ints or floats. I can point to other pointers!

```
int x = 7;
int y = 9;
int * p = &x; //p points to an int
int * q = &y; //q points to an int
int ** v = &p; //v points to an int *! We initialize it by getting an arrow to an int *!
```



This situation leaves us with the following boxes and arrows:



decode what happens as we move things around in crazy ways:

`* v = & y;`

So we *dereference* `v`, meaning we're talking about the box `v` points to (`v`), and put the address of `y` in it:

And, of course, you can have pointers-to-pointers-to ints, and so on. The main thing to remember is: *using the address-of operator gives you an arrow to the box you're talking about* and *using the dereference operator follows one arrow*.

Pointers to Structs

When we define a structure, we're defining the layout of box in memory:

```
typedef struct {  
    int x;  
    float y;  
    int z[10];  
} myStruct;
```

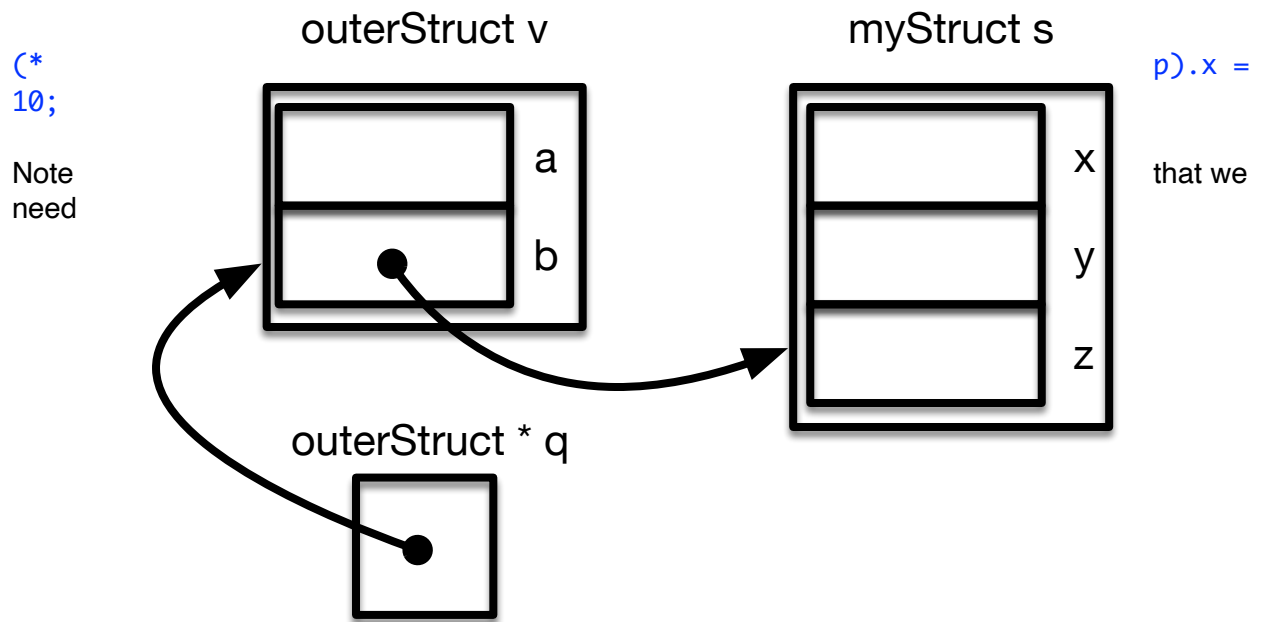
```
myStruct s;
```

This makes `s` a box in memory that has three fields next to each other: 4 bytes for the integer `x`, 4 bytes for the float `y`, and 40 bytes for the 10-integer array, `z`. Note that if we had a struct nested inside of it, there would be a “sub-box” inside the structure that is laid out like the other struct (in the same way that the array is inside the structure).

So what happens if we have a pointer to the structure?

```
myStruct * p = &s;
```

Then `p` points to `s`. If we want to access field `x` in `s` using `p`, we need to first de-reference `p` so we can follow the arrow and talk about `s`. Then we can access the field `x`:



parentheses here because the dereference operator has lower priority than the field-access operator. So if we wrote `*p.x`, it would mean “Find field `x` in structure `p`, then dereference that field.”

C provides cleaner syntax for the very common operation of “access a field in a structure pointed to by a variable”:

`p -> x = 10; //this is exactly the same as (* p).x = 10`

To see why this is useful, consider a situation where we have nested structures:

```
typedef struct {
    int a;
    myStruct * b;
} outerStruct;
```

```
outerStruct v;
```

```
v.b = &s; the b field of v points to s
```

```
outerStruct * q = &v; //q points to v
```

Now, to access field `x` from `s`, we could write:

```
(* (* q).b).x
```

So dereference `q` (giving us `v`), then access field `b` (which points to `s`), then dereference that field (giving us `s`), then accessing field `x`. As you can see, this results in a lot of parentheses and `*s`, and the `*s` are all over the place. Instead, we could write this as:

```
q -> b -> x
```

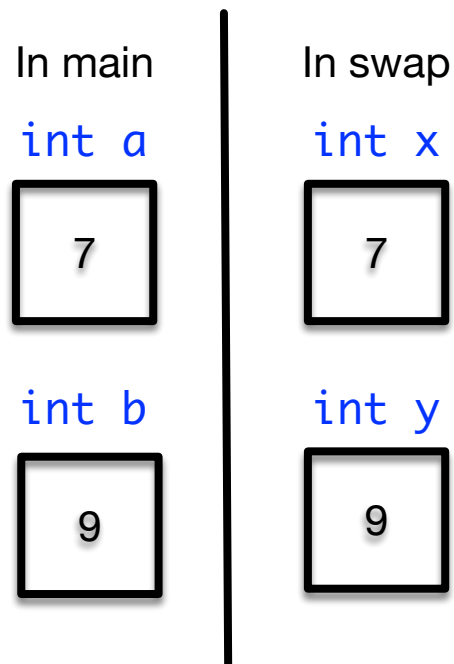
Passing arguments to functions

When we pass data to functions, we *always* pass copies of boxes: the arguments inside the function are different versions of data than the variables outside the function. So if we try to write a swap function, it may not work:

```
void swap (int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
  
int a = 7;  
int b = 9;  
swap(a, b); //inside swap, these are copies
```

So inside swap, when we change `x` and `y`, we're not really manipulating `a` and `b` at all.

So how do we deal with this? Well, the general rule of thumb is: *if you want a function to change a variable, pass the address of the variable to the function.* Remember, that the address-of operator creates an arrow. So if you pass an arrow to a function, what do you have? A pointer to the original variable!

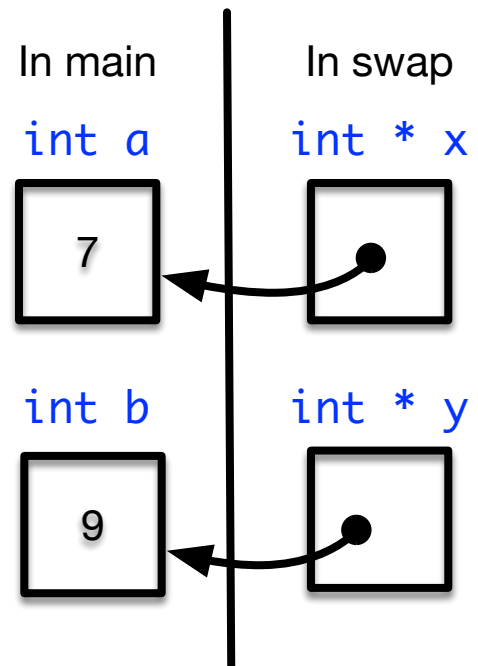


```
void swap (int * x, int * y) {
    int t = * x;
    * x = * y;
    * y = t;
}
```

```
int a = 7;
int b = 9;
swap(&a, &b); //actually swaps!
```

And now that we have pointers to the original variables, we can use the dereference operator (*) to access the original boxes, and change their values!

Note that this trick of passing addresses to variables is why `fscanf` and `scanf` take the addresses of variables that you want to set: that way you can change the variable itself instead of a *copy* of the variable:



```
scanf("%d", &x); //read an integer and store it in x
```

Void pointers

Some times, you want to store an address in a variable (or pass it to a function, or return it to a function) without knowing *what*, exactly, that address refers to. C lets you do this using *void pointers*.

A void pointer is declared with a special data type, `void *`:

```
void * ptr; //pointer to _something_
```

All we know, now, is that pointer holds an address. But it could be an address to an integer, it could be an address to a float, or it could even be the address of a complicated custom struct! To tell our program what we're pointing to, we need to *cast* it to the pointer type we care about:

```
int * iptr = (int *) ptr; //iptr points to an integer
* iptr = 7
```

```
float * fptr = (float *) ptr; //fptr points to a float
* fptr = 10.4;
```

```
myStruct * sptr = (myStruct *) ptr; //sptr points to a myStruct
sptr->x = 9;
```

C does not try to save you from yourself. You can cast a `void *` pointer to whatever other kind of pointer you want, and C will assume that the address holds that thing. It will not try to check that the address *actually* holds that type of data. So you can accidentally (or on purpose) do things like have a `float *` point to an integer, and C will interpret the data as a floating point value (remember that data types tell C how to interpret bits)

So what do we use void pointers for? Well, they let us write very generic code that uses void pointers as arguments. We'll see some examples of this when we get to memory allocation and function pointers.

Function pointers

One way that you can make code more generic is to be able to invoke functions without knowing ahead of time what they are. Normally, when we invoke a function like `foo`, we use the name `foo`, and the compiler generates code that invokes that function, passing the necessary arguments and collecting the return value. Note that this requires that the compiler knows exactly what function you're going to call when you call it.

But what if we want to call a function in a piece of code without knowing ahead of time what function we're going to call? Remember that all functions actually have addresses in memory (in the `.text` section). So what if we could put the address of a function in a pointer. As long as the compiler knew what the arguments to the function were, and what the return type was, it could generate code to call the function *without knowing ahead of time what function was being called* — it could just get the address from the pointer!

Compare this to how we use pointers to data. Rather than knowing the name of the variable we want to read from or write to (i.e., knowing ahead of time what box we're talking about), a pointer lets us access “whatever box has this address,” letting us change the address in our code and write more reusable code. Function pointers do the same thing: “call whatever function has this address” rather than “call `foo`”

The syntax for function pointers is a little weird. The *type* of the pointer has to encode both the types of the arguments to the function and the type of the return value (because that's how the compiler knows how to generate the right code). So if we want to create a function pointer `ptr` that can point to *any* function that takes an `int` and a `double` as an argument and returns a `float`, we would write this:

```
float (* ptr) (int, double);
```

We can then assign it to a function that we want to call, and then call it:

```
ptr = foo;  
float f1 = ptr(3, 2.5); //calls foo
```

```
ptr = bar;
```



```
float f2 = ptr(3, 2.5); //calls bar
```

Note that you can also call the function using the function pointer by more explicitly dereferencing the pointer (this may help you keep straight what's actually happening — you're not calling a function named `ptr`, you're calling the function that `ptr` points to):

```
float f2 = (* ptr)(3, 2.5);
```

Now we can write generic code that uses a function pointer and pass in different kinds of functions depending on what we want the function to do! In HW3, you use this ability to create an array of function pointers and call the same integrate function on different functions.

The syntax for function pointers can be a little tedious, so we can also use typedef to create a new name for the type:

```
typedef float (* funcPtr_t) (int, double);  
funcPtr_t ptr = foo; //same as float (* ptr) (int, double) = foo;
```

That will let us more easily create things like arrays of function pointers.

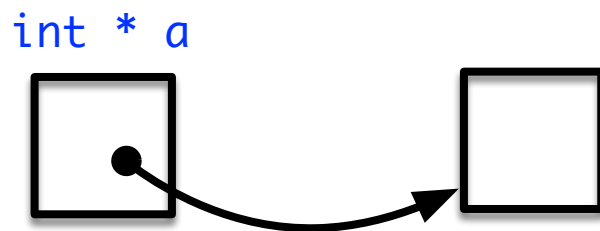
Memory allocation

Up until now, we have had our pointers point to other variables. The box the pointer points to is already named by another variable. But we can also ask C to give us an “anonymous” box: a box that we can store data in, but that doesn't have a name. Thus, there is no way to access that box *except* by having a pointer point to it.

These anonymous boxes are located on the *heap*, another part of your program's layout (see Week 2 notes). You can create anonymous boxes by using a special function called `malloc` (which stands for “memory allocate”):

```
int * a;  
a = (int *) malloc(sizeof(int));
```

And now `a` points to an anonymous box in memory. So what is going on here?



malloc

The signature of `malloc` is:

```
void * malloc(size_t size);
```

What this means is that you tell `malloc` *how big* you want the box to be in bytes (`size_t` is a special type that C uses to represent sizes of data types). Note that `malloc` doesn't know

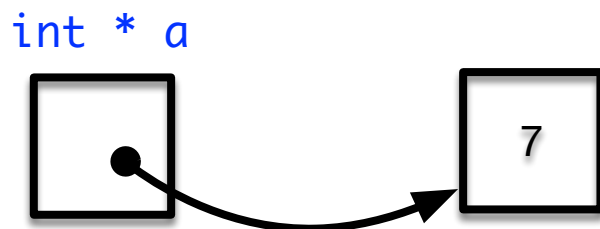
anything about data types. All it wants to know is how big the box should be. In this case, we want the box to be big enough to hold an `int`, so we tell `malloc` that we want the box to be, well, the size of an `int`! `sizeof` is a special function the compiler uses to figure out how big (in bytes) any data type is. You can also tell the compiler to make the box “as big as the thing `a` should point to” and get the same effect:

```
a = (int *) malloc(sizeof(* a));
```

Note that if you said `malloc(sizeof(a))`, the compiler would interpret this as “as many bytes as `a` takes up” — and `a` is a pointer, not an `int`, so it would create a box that is 8 bytes big.

It then returns the address of the newly allocated box. Because `malloc` doesn't know about data types, this address is a void pointer. But we can cast it to a pointer to an `int`, and now we can dereference that to put values in the box!

```
*a = 7
```



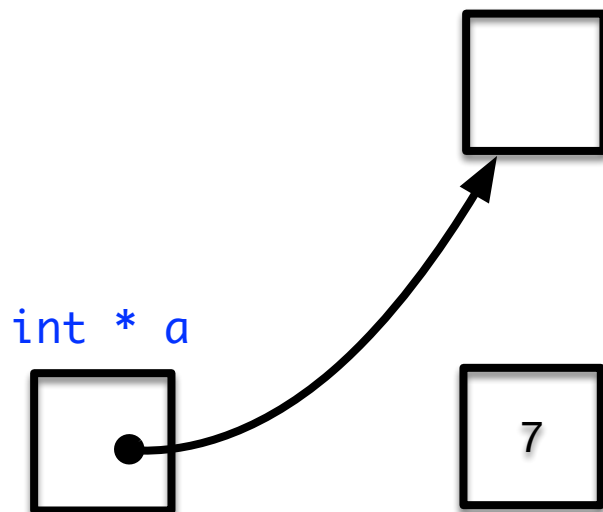
You can also use `malloc` to allocate structures in memory. `sizeof` works on any data type, including custom structs:

```
myStruct * s = (myStruct *) malloc(sizeof(myStruct))
```

Note that we can call `malloc` again to make `a` point to a different box in memory. `malloc` guarantees that we will get back a different box:

```
a = (int *) malloc(sizeof(int));
```

But now look what happened! The box that holds 7 is anonymous: it doesn't have a name. So there is no way to access it except through a pointer that holds its address. Unfortunately, `a` no longer points to that box, so it seems like there is no way to access that 7 anymore. The box is just floating in the heap, taking up space, and serving no purpose. We have a memory leak!



Memory leaks

A memory leak happens when we use `malloc` to allocate a box on the heap, and then lose the ability to access it anymore through a pointer (or a chain of pointers). Once that happens, the box sits on the heap and `malloc` won't let us use that space anymore (`malloc` will always give you a new box as opposed to any box that you've allocated before). If we do this in, say, a loop, we can create a huge number of boxes on the heap, and eventually your program can crash!

```
for (int x = 0; x < BIG_NUMBER; x++) {  
    a = (int *) malloc(sizeof(int)); //creates a new box each time  
}
```

To avoid memory leaks, we need to tell `malloc` that we're done with a box before we lose access to it. Once we know we don't need the box anymore, we can *free* it, releasing the memory and letting `malloc` use it again—a future call to `malloc` may return the old box's address.

Note that if `malloc` cannot find space for a box, it will return `NULL`. You should always check to make sure `malloc` has given you a valid address; if it returns `NULL`, that means the heap is out of space.

free

The function that releases memory is called `free`, and it has the following signature:

```
void free(void * addr);
```

You pass `free` a pointer that points to a box, and `free` will *deallocate* that box, returning it to the heap so `malloc` can use it again.

Note that you can only `free` boxes that were originally `malloced`. If you pass `free` the address of a box on the stack, your program will crash. And you can only `free` boxes by pointing to the address at the beginning of the box. If you allocate a large box and try to call `free` on an address in the middle of the box, your program will crash.

You need to use `free` carefully: if you deallocate a box too early, while you still need it, `malloc` could re-allocate that box, and you might accidentally overwrite important data. But if you wait too long to `free` a box, you may no longer have a way of pointing to it, and you'll get a memory leak.

Dynamic arrays

Now that we have the ability to dynamically allocate memory using malloc, it turns out we can use it to allocate arrays at runtime. Normally, you have to know the size of an array when you allocate it:

```
int a[5]; //5 element array
int a[] = {5, 2, 4}; //3 element array
int a[]; //this doesn't work!
```

But we can instead use `malloc` to allocate an array of variable size. The key here is that an array places all of its elements right next to each other in memory. So if we want a 10-integer array, we can allocate a box big enough to hold 10 integers:

```
int * a = (int *) malloc(10 * sizeof(int));
```

```
int * a
```



Note that even though `a` is an integer pointer, it points to a 40-byte box. C has one more trick up its sleeve: an `int *` pointer can be treated as though it's an array of `ints`!

```
a[0] = 7; //assigns the first element of the array
a[7] = 10; //assigns the eighth element of the array
```

```
int * a
```



And there is no reason that we had to know the size of the array to begin with. Suppose we have a variable `size`, and we want to create an array of `size myStructs`:

```
myStruct * s = (myStruct *) malloc(size * sizeof(myStruct));
```

So what's happening here? Well, it turns out that arrays and pointers are very closely related in C. In fact, under the hood, arrays are basically pointers that point to the first element of the array. There are some subtleties regarding how, exactly, the types work, but the following code is valid:

```
int x[] = {0, 1, 2}; //3 element array
int * a = x; //a points to the same 3-element array
a[2] = 4; //x is now {0, 1, 4}
```

So if an array is just a box in memory of a bunch of elements next to each other, and the array variable points to the same element, then we can just as easily allocate that box on the heap and have a pointer point to it!

The mechanics by which this work involve *pointer arithmetic*

Pointer Arithmetic

If `p` is a pointer integer, then `p` holds an address. An address is just a number, representing a location in memory, so we can add or subtract from it!

```
int * p = ...
int * q = p + 1;
```

But arithmetic on pointers is tricky. `p + 1` doesn't mean "add 1 to the address in `p`." Because `p` is an `int *`, `p + 1` means "add *one int's worth of size* to the address in `p`." In other words, add 4 to the address in `p`. If `p` points to an `int`-sized box, `q` points to an `int`-sized box *right next to the box `p` points to*.

If `p` pointed to `doubles` instead, 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.

Pointer arithmetic and arrays

So what happens if we think about pointer arithmetic and arrays?

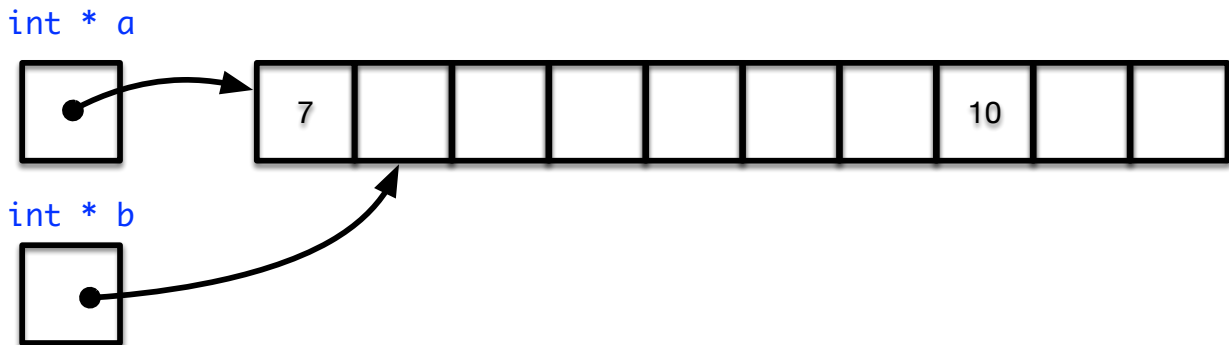
```
int * a = (int *) malloc(10 * sizeof(int));
```

What does `a` point to? It points to the first element of the array.

```
int * a
```



And what about $a + 1$? Well, that's the address 4 bytes past the first box. So if $b = a + 1$, then b holds the address of the second box!



And $a + 2$ holds the address of the third box, and so on!

And now comes the punchline: $a[i]$ means *exactly the same thing* as $*(a + i)$. Every time we access the i th element of a , we're actually adding i to a (using pointer arithmetic) and dereferencing that location! There is no difference between an array of `ints` (or `floats` or `structs`), and a pointer to a box containing a bunch of `ints` (or `floats` or `structs`) in a row!

Of course, once we're done with our `malloced` array, we still have to remember to free it:

`free(a)`

(Remember that `free(b)` will crash your program: `b` doesn't point to the beginning of a box that was `malloced`)