Arrays

week 02

- An array is another type of data that allows you to store multiple values of the same type back-to-back-to-back.

↓

CONTIGUOUSLY

So an array can let you create memory for one int, or two, or three, or even more than that, but describe them all using the same variable name, the same one name.

```
int score1 = 42;
int score2 = 31;   ↔
int score3 = 2;
```

I can do this, instead:

```
int scores[3];
```

↑ is declaring for me an array of size 3; And that array is going to store 3 integers;

why? Because the type of that array is an int, here.

the square brackets tell the computer HOW many ints you want.

So if I want to now assign VALUES to this variable, called SCORES, I can do code like this:

```
int scores[3];

scores[0] = 72;
scores[1] = 23;
scores[2] = 1;
```

indexing (") starts counting at 0.

Arrays in C are ZERO INDEXED.

You can't start counting at 1 in arrays because you prefer to, you'd be sacrificing one of the elements.

I don't need to say int before these lines because that's been take care of, already, for me on line 5, where I already specified that everything in this array is going to be an int.

```
((score[0] + score[1] + score[2]) / 3.0);
```

But still not really solving all the problems we describe ...

But now I can store multiple values in the same variable! :)

...it's then, use get int to ask the user for another score.

The only thing that is changing is the number and it's just incrementing by 1.

We have all of the building blocks to do this better. So let me go ahead and improve this:

FOR LOOP

```
int scores [3];

for (int i = 0; i < 3; i++)
{
    scores [i] = get_int ("Score: ");
}
```

So this is where arrays start to get pretty powerful. You don't have to hard code, that is, literally, type in all of these magic numbers like 0, 1 and 2.

↳ I'm now, dinamically, getting 3 different scores, but putting them in these #'s locations.

And so, this program, ultimately, is going to work, pretty much, the same. But is a little better designed, too.

If I really want to nitpick, there's something that still smells, a little bit here. The fact that I have indeed, this magic number 3, that really has to be the same as this number here.

Otherwise, who knows what's going to go wrong? Ok, I can make that even more dinamically. How?

```
int main (void)
{
    int n = get_int ("How many scores?");
    int scores [n];

    for (int i = 0; i < n; i++)
    {
        scores [i] = get_int ("Score: ");
    }
```

Dinamic code.

Years ago, memory was very expensive! And very one of your instincts would have been spot on because memory is so tight.

But, nowadays, we don't worry as much about it. 😉

So now we have this ability to create an array.

And an array can store multiple values. What, then, might we do that's more interesting than just storing numbers in memory?

→ Well, let's take this one step further.

As opposed to just storing 72, 73, 33 or 100, 99, 98, at these given locations, because again, an array gives you one variable name, but multiple locations, or indices therein bracket 0, bracket 1, bracket 2 and up, if I were even bigger than that.

Let's, now, start to consider something more modest, like simple chars.

Chars, being 1 byte each, so they're even smaller, they take up much less space.

And, indeed, if I wanted to say a message like, "hi I could use 3 variables? If I wanted a program to print hi

H - I - !

literally, I could store those in 3 variables, like c1, c2, c3.

And let's, for the sake of discussion... let's whip this up real quickly.

↳ Printf is able to tolerate printing them as integers.

If I really want it to be pedantic, I could use this technique, again, known as TYPECASTING,

↓

where I can actually convert 1 data type to another, if it makes logical sense to do so.

* ↳ But we saw in week 0, that chars, too, are just NUMBERS, like 72, 73 and 33. ↳ ASCII table!

So I can use this parenthetical expression to CONVERT, incorrectly, 3 chars to 3 integers, instead.

So that's mean TYPE. So (int) says "take whatever variable comes after this c1, c2 or c3 and convert it to an int.

↳ the effect is going to be NO different, make hi, and then rerunning whoops: 72, 71, 73. ✓

I'm explicitly converting chars to ints.
— An int has NO decimal point. —
But, for now, I'm going to rewind to the version of this implicit casting, just to demonstrate that we can, indeed, see the values underneath the hood.

→ There's clearly an equivalence, then, between sequences of chars and strings. And if you do it the real pedantic way, you have 3 different variables c1, c2, c3, representing H-I-! or you can just treat them all together like this: hi!

```
int main(void)
{
    string s = "Hi!";
    printf("%s\n", s);
}
```

But it turns out that strings are actually implemented by the computer in a pretty new familiar way. 😊

"What might a string actually be or of this point in the story?"

So, string like this is an array of chars? ö YES!
which actually opens up some interesting possibilities for us.

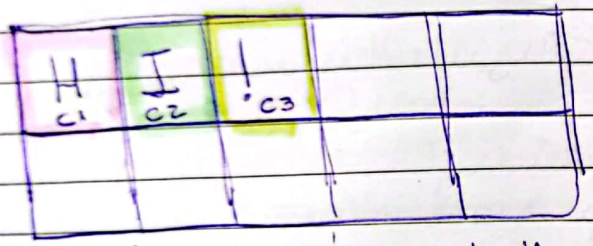
So maybe I can start poking around inside of strings, even though we didn't do this last week, so I can get at those individual values.

```
int main(void)
{
    string s = "Hi!";
    printf("%c %c %c\n", s[0], s[1], s[2]);
}
```

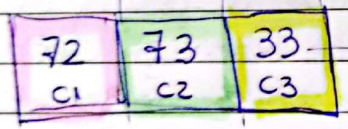
A string is just an array of characters, and arrays, you can index into them using this new square bracket notation.

↳ So I can get at any one of these individual characters, and convert it to an integer like we did in week 0.

What else might be in the computer's memory?



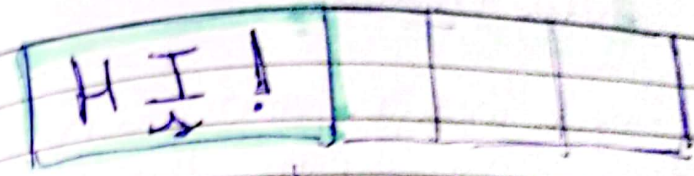
the map to these decimal digits or equivalent these binary values.
→ Ascii table



But what is this looking at memory?

string s = "Hi!"

When you create a string in memory, string s equals "Hi!". Let's consider what's going on



big box of size 3.
But technically, ~~or~~ a string is an array.

↳ So if the string is called s, s[0] will give you the 1st character, s[1] the 2nd, the s[3] the 3rd.

But if this is the only thing in my computer memory and the ability, like a camera to draw 0s and 1s, or numbers, or characters, or whatever on it... but that's it, like this is your mac, PC and phone ultimately reduced to.

Suppose that I'm running a piece of software, like a text messenger, and now I write down "bye!".
Well, where might that go in memory?

How does the computer know if B-Y-E-! is right AFTER H-I-! where 1 string ends and the next one begins?

NUL Character

All we have are bytes, or 0s and 1s. So if you were designing this, how would you implement some kind of delimiter between the two? Or figure out what the length of a string is?

Special Character → Sentinel character

Humans decided some time ago that you know what, if we want to delineate where one string ends and where the next one begins, we just need some special symbol.

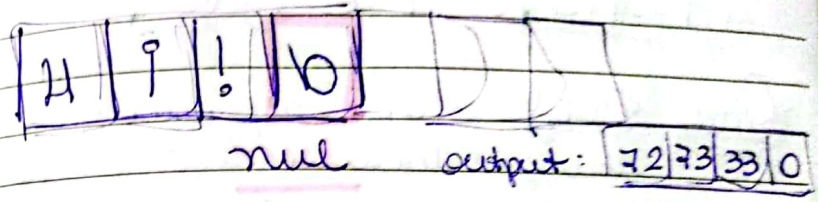
↳ the symbol they'll use is generally written as backslash 0. this is just shorthand notation for literally eight 0 bits.

0 0 0 0 0 0 0 0 '\0'

N-U-L it's the nickname.

the computer is apparently using not 3 bytes to store a word like hi, but 4 bytes.

whatever the length of the string is, +1 for this special sentinel value that demarcates the END of the string.



" How do we distinguish 1 string from another? "

↳ malloc(sizeof(int) * 4 (+1))

↳ 1 byte

If I were to make 1 other variant of this program, let's get rid of just this word and let's have e.

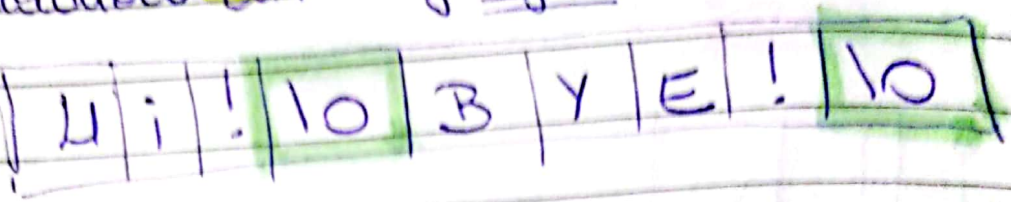
```
int main(void)
{
    string s = "Hi!";
    string t = "BYE!";

    printf("%s\n", s);
    printf("%s\n", t);
}
```

Output:

Hi!
BYE!

So if (s) is string hi and apparently 1 bonus byte that demarcates the end of that string,

byte is apparently going to fit into the location directly after.



Only strings are accompanied by nuls at the end because every other data type we've talked about thus far is of well defined finite length.

important!

1 byte — char
4 bytes — ints
"