STRING.c

We just saw strlen as an example of a function that comes in the string library.

Let's start to take more of these library functions out for a spin. So we are not relying only on the built ins that we saw last week.

If I create a file called "string.h" I need to include $\boxed{\text{cs50.h}}$ ← header ← $\boxed{\text{stdio.h}}$ and this new thing $\boxed{\text{string.h}}$ as well

And then, in the program string.c, suppose for the sake of discussion, that I didn't know about $\bigcirc\!\!\!\!\%$. for print or heck maybe early on there was no $\bigcirc\!\!\!\!\%s$ format code.

And so there was no easy way to print strings well, at least if you know that strings are just arrays of character, we could use %c as a workaround, a solution to that, sort of, contrived problem.

→ So we can ask for a string is by using get string and I'll ask the user for some input. And let me print out (output).

In pseudocode (or in English) what's the gist of how we could solve this problem, printing out the string $\boxed{s}$ on the screen without using %s?

How to make my program to uppercase everything to capital H, capital I?

ASCII chart → capital A is 65
↳ capital B is 66
↳ capital C is 67

↓
lowercase a is 97
" b" 98
" c" 99

$$\begin{array}{r} 97 \\ -65 \\ \hline \boxed{32} \end{array}$$

there's a difference of 32

So if I go from uppercase to lowercase,

I can do 65 ⊕ 32, will give me 97 and that actually works out across the board for everything else (a, b, c, d, ...).
Characters are just binary representation of numbers.

TO/UPPER → there's a function that would literally do the uppercasing thing for me, so I don't have to get into the weeds of negative 32, plus 32 like before. I don't have to consult the chart. Someone has solved that problem for me in the past. And let's see if I can actually get back to it...

Command line argument → the file you want to delete, taking command line arguments that modify the behavior of the program.

We've been typing int main(void) for the past week and just asking that you take on faith that it's just the way you do things. By default in C, at least the most recent versions thereof, there's only 2 official ways to write main functions:

ARGC and ARGV

```
int main(int argc, string argv[])
{
  ...
}
```

Argument count is going to be integer | that stores how many words the human typed at the prompt.

the C automatically gives that to you. String argv[] stands for argument vector, that's going to be an array of all of the words that the human typed at the prompt.

So with today's building block of an array, we have the ability now to let the humans type as many words or as few words, as they want at the prompt, C is going to automatically put them in an array called argv, and it's going to tell us how many words there are in an int called argc.

the int, as the return type here, we'll come back to in just a moment...

→ argv [0] is itself an array of strings, and [0] is itself a SINGLE string.

So it can be plugged into that % placeholder.

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    printf("hello, %s\n", argv[1]);
}
```

```
if (argc == 2)
{
    printf("hello, %s\n", argv[1]);
}
else
{
    printf("hello, world\n");
}
```

```
$ make argv
$ ./argv David
$ hello, David
```

argument

→ PROGRAMMING DEFENSIVELY

Now I have some error handling here in
(argc == 2)
Because, again, argc is argument count, the number of words, total, typed at the command line.
I would need to alter my logic to support more than just 2 words after the prompt.

~ So what's the point of this? ~

If you had to use get string every time you compile your code, it'd be kind of annoying, right?
In this command-line-interface world, if you support command line arguments, then you can use these little tricks.
↳ that's why we had (void) initially, but what more we can now put in main? Context matters!

Exit Status

- Usefully to automating tests of your own code;
- When it comes to figuring out if a program ~~suc~~ succeeded or failed;
- It turns out that main has $\boxed{1}$ more feature we haven't leveraged; an ability to SIGNAL to the user ~~whether~~ whether something was successful or not.

And that's by ~~the~~ way of main's return value.

So I'm going modify this program as follows, like this:

- Suppose I want to write a similar program that requires that the user type a word at the prompt, so that argc has to be 2 for whatever design ~~ott~~ purpose.

- If argc does NOT equal 2, I want to quit out of my program prematurely, because I want to insist that the user operate the program correctly.

- So I might give them an error message like "missing command line argument \n".

But now I want to quit out of the program. Now HOW can I do that?

↓

the right way to do that is to return a value from main. Now it's a little weird because no one called main yet, main just gets called automatically, but the convention is anytime something goes wrong in a program you should return a non-zero value from main.

↓

1 is fine as a go-to

If you return 1, that is a clue to the system, the Mac, the PC, the cloud device that's something went wrong, Because 1 is not 0.

If everything works fine, let's go ahead and print out("hello, %s\n", argv[1]). If works fine, we will return 0.

Main has always defined by us as taking on int as a return value. By default, main automatically sort of secretly, returns 0 for you.

If you've never once use the return keyword, which you probably haven't in main, it just automatically returns 0 and the system assumes that all went well. But now that we're starting to get a little more sophisticated with our code, and you know, the programmer, something went wrong, you can abort programs early.

You can exit out of them by returning some other value, ~~bt~~ besides 0, from main.

And this is fortuitous that it's an int, right?

$\boxed{\text{0 — everything worked}}$

Unfortunately, in programming, there are seemingly, an infinite number of things that can go wrong. And it gives you 4 billion possible codes that you can use, exit statuses, to signify errors.

So if you've ever on your Mac or PC gotten some weird pop up that an error happened, sometimes, there's a cryptic number in it.

→ Maybe it's ⊕, maybe it's ⊖

It might say "error code 123", or negative -49 or something like that.

What you're generally seeing, are these EXIT STATUSES, these return values from main in a program that someone at Microsoft or Apple or somewhere else wrote, something went wrong, they are unnecessarily showing you, the user ~~the~~ what the error code is.

If only, so that when you call customer support or submit a ticket, you can tell them what exit status you encountered, what error code you encounter.

Readability

When analyzing text, well if the words are kind of short, everything's very simple, that's probably a very young, or early, grade ~~[illegible]~~ level.

- "What makes this text assume an older audience, a more mature audience, a higher grade level, would you think?"

SOPHISTICATION

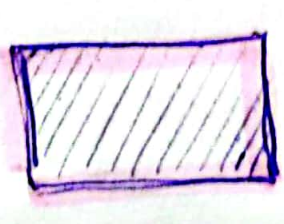
CRYPTOGRAPHY this notion of scrambling information in such a way that you can hide the contents of a message from someone who might otherwise intercept it.

the earliest form of this might also be when you're younger, and you're in class, and you're passing a note from one person to another, from yourself to someone else.

You don't want to necessarily write a note in English or some other written language you might want to scramble it somehow, or encrypt it.

Maybe you change the As to a B, and the Bs to a C, so that if the teacher snaps it up and intercepts it, they can't actually understand what it is you've written because it's encrypted.

So long as your friend, the recipient of this note, knows how you manipulated it, how you added or subtracted letters to each other, they can decrypt it, which is to reverse that process. So formally, in the world of cryptography, and computer science, this is another problem to solve.

input →  → output

Your input, though, when you have a message you want to send securely, is what's generally known as plain text.

There's some algorithm that's going to then encipher, or encrypt that information, into what's called ciphertext, which is the scrambled version that theoretically can get safely intercepted and your message has NOT been spoiled, unless that intercept actually knows what algorithm you used inside of this process.

$$\begin{tikzcd}\text{plaintext} \arrow[r] & \boxed{\text{cipher}} \arrow[r] \arrow[dl, bend right] & \text{ciphertext} \\ & & \end{tikzcd}$$

So that would be generally known as a cipher. The ciphers typically take, though, not 1 input, but 2.

If, for instance, your cipher is a simple as A becomes B, B becomes C, C becomes D, ..., Z becomes A, you're essentially adding 1 to every letter and encrypting it.

You and the recipient both have to agree, presumably, before class, in advance, what number you're going to use that day to rotate or change ALL of these letters by.

Because when you add 1, they upon receiving your ~~[illegible]~~ ciphertext, have to subtract 1 to get back the answer.

For instance, if the input, plaintext is hi, as before, and the key is 1, the ciphertext using this simple rotational algorithm, otherwise known as the Caesar cipher, might be ij exclamation point.

So it's familiar, but it's at least scrambled at first glance. Unless the teacher really cares to figure out what algorithm are they using today, or what ~~[illegible]~~ key are they using today, it's probably sufficiently secure for your purposes.

How do you reverse the process?

Well, your friend gets this and reverses it by negative 1.