Week 2 — Arrays

- What's the format into which we need to get our program that we write, just to recap?

BINARY — otherwise known as machine code.

The 0s and 1s that your computer actually does understand. So somehow we need to get to this format. And up until now, we've been using this command called MAKE which is aptly named, because it lets you make programs.

→ And the invocation of that has been pretty ~~straight~~ simple.

```
make hello
./hello
```

→ looks in your current directory or folder for a file called hello.c, implicitly, and then it compiles that into a file called hello, which itself is executable, which just means runnable, so that you can then do ./hello.

But it turns out that make is actually NOT a compiler itself. *

It does help you make programs. But make is this utility that comes on a lot of systems that makes it easier to actually compile code by using an actual compiler, the program that converts source code to the machine code, on your own Mac, or PC, or whatever cloud environment you might be using.

In fact, what make is doing for us, is actually running a command automatically known as clang, for C language.

In VS Code, is that very first program again, this time in the context of a text editor.

$ clang hello.c → it creates a file called a.out

a.out : this is a historical convention, stands for assembler output. And this is, just, the default name for a program that you might compile yourself, manually, using clang itself.

```
./a.out
hello, world
```

these are literally words or numbers that you type at your prompt after the name of a program that just influences its behavior in some way. It modifies its behavior. And it turns out, if you read the documentation for clang, you can actually pass a -o, for output, command line argument, that explicitly what do you want your outputted program to be called?

```
$ clang -o hello
hello.c
```

And then you go ahead and type the name of the file that you actually want to compile, from source code to machine code.

```
$ clang -o hello hello.c
$ ls
a.out  hello  hello.c
```

"We still have the old a.out, because I didn't delete it yet."

And I do have hello now

So ./hello runs hello, world again.

```
$ clang -o hello hello.c -lcs50
```

↳ telling clang that not only do you want to output a file called hello, and you want to compile a file called hello.c, you also want to quote-unquote link in a bunch of .os and .hs that collectively implement get string and printf.

```
./hello
what's your name? Fernanda
hello, Fernanda
```

But we'll continue to use make because it just automates that process. → I don't need to put -lcs50 and -lm, for example.

So, clang is the command that converts from source code to machine code.

*What does the -o mean?

-o is shorthand for the English word output, and so -o is telling clang to please output a file called hello, because the next thing I wrote after the command line recall was clang -o hello, then the name of the file, then -lcs50.

And this is where these commands do get and stay fairly arcane. It's just through muscle memory and practice that you'll start to remember, oh what are the other commands that you — what are the command line arguments you can provide to programs? But we've seen this before.

Technically, when you run make hello, the program is called make, hello is the command line argument. * IMPORTANT!

It's an input to the make function, albeit, typed at the prompt, that tells MAKE what you want to make.

Compiling

<stdio.h> → printf

$ clang -o hello hello.c -lcs50

When you run this command there's a few things that are happening underneath the hood, and we won't dwell on these kinds of details, indeed, we'll abstract it away by using MAKE.

But it's worth understanding from the get-go, how much automation is going on, so that when you run these commands, it's not magic. You have this bottom-up understanding of what's going on. So when we say you've been compiling your code with make, that's a bit of an oversimplification.

Technically, every time you compile your code, you're having the computer do four distinct

things for you. And this is not four distinct things that you need to memorize and remember every time you run your program, what's happening, but it helps to break it down into building blocks, as to how we're getting from source code, like C, into 0s and 1s.

It turns out, that when you compile your code, technically speaking, you're doing 4 things automatically, and all at once:

1º → preprocessing
2º → compiling
→ assembling
→ linking

(assembly)

Just humans decided, let's just call the whole process compiling

But for a moment, let's consider what these steps are. So preprocessing refers to this:

If we look at our source code earlier, that uses the cs50 library and therefore get string, notice that we have these indu de lines at top. *

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

int main(void)
{
    ...
}
```

TOP

And they're kind of special versus all the code we've written, because they start with hash symbols, specifically. And that's sort of a special syntax that means that these are, technically, called preprocessor directives. * important!

↳ Fancy way of saying they're handled special versus the rest of your code.

In fact, if we focus on cs50.h, recall from last week that he provided a hint # or to what's actually in cs50.h, among other things → "Why we were including it in the 1º place?"

Because we need to copy and paste the prototype of the function at the TOP of the file, just to teach the compiler that this function doesn't exist, yet, it does down there, but it will exist. So again, that's what these prototypes are doing for us. :)

#include <cs50.h> ↔ string get_string (string prompt);

that's the same ...

#include <stdio.h> ↔ int printf (string format, ...);

"So what does it mean to preprocess your code?"

the very first thing: the compiler, clang, in this case, is doing for you when it reads your code top-to-bottom, left-to-right, is it notices, oh, here is hash include, oh, here's another hash include ...

And it essentially, finds these files on the hard drive, cs50.h, stdio.h, and does the equivalent of copying and pasting them automatically into your code at the very top.

Thereby teaching the compiler that get string and printf will eventually exist somewhere.

So that's the preprocessing step. Literally, again, just doing a find-and-replace of anything that starts with hash include. It's plugging in the files there so that you, essentially, get all the prototypes you need automatically.

→ Compile the results (2)

Now your code looks like this in the computer's memory:

```
string get_string (string prompt);
int printf (string format, ...);
```

It doesn't change your file, it's doing all of this in the computer's memory, or RAM, for you.

But it, essentially, looks like this.

The next step is what's, technically, really compiling.

Compiling code, in C, means to take code that now looks like this in the computer's memory and turn it into something that looks like this (image) → CRYPTIC "0"

=it's assembly language. 😮

low level as you can get to what a computer really understands, be it a Mac, or PC, or a phone, before you start getting into actual Cs and 1s.

this is assembly language that the computer is outputting for you automatically, still for mention of main and it for mention of get string, and it for mention of printf.

So there's some relationship to the C code we saw a moment ago. And in yellow, these are what are called computer instructions.

At the end of the day, your MAC, your PC, your phone actually only understands very basic instructions, like addition, subtraction, division, multiplication, move into memory, load from memory, print something to the screen, very basic operations.

And these assembly instructions are what the computer actually feeds into the brains of the computer, the CPU, the central processing unit.

↳ And it's that Intel CPU, or whatever you have, that understands this instructions, and this one, etc. And collectively, long story, all they do is "print hello", world on the screen, but in a way that the machine understands how to do.

The commands, the instructions that those two products understand (MAC and Windows, for example), are actually different.

Now, Microsoft or any company, could genially write code in 1 language like C or another, and they can compile it twice, saying a PC version and a Mac version. !!

It's twice as much work and sometimes you get into some incompatibilities, but that's why these steps are somewhat distinct.

You can now use the SAME code and support even ≠ platforms, or systems, if you'd want.

③ Assembling

It's just happening for you every time you run make or, in turn, clang, this assembly language, which the computer generated automatically for you from your source code, is turned into 0s and 1s.

④ Linking

Even in this simple program of getting the user's name and then plugging it into printf, I'm using three different people's code; My own — which is in hello.c — some of CS50's — which is in cs50.c, which is not a file I've mentioned, yet, but it stands to reason, that if there's a cs50.h that has prototypes, turns out, the actual implementation of get string and other things are in cs50.c.

And there's a 3rd file somewhere on the hard drive that's involved in compiling even this simple program: cs50.c, stdio.c and main.c. And that's a bit of a white lie, because that's such a big, fancy library, that there's actually multiple files that compose it, but the same idea, and we'll take the simplification.

So when I have this code and I compile my code, I get those 0s and 1s that end up taking hello.c and turning it, effectively into 0s and 1s that are combined with cs50.c, followed by stdio.c as well.

that links all of these 0s and 1s together, essentially stitches them together into 1 single file called hello, or called a.out, whatever you name it.

→ that last step is what combines all of those different programmers' 0s and 1s.

When you're running make, there's some very concrete steps that are happening that humans have developed over the years, over the decades, that breakdown this big problem

of source code going to 0s and 1s, or machine code, into these very specific steps.

But henceforth, you can call all of this compiling.

> a.out → just the conventional default file name for ANY program that you compile directly with a compiler, like clang. It's a meaningless name, though.

It stands for assembler output, and assembler might now sound familiar from this assembling process. It's a lame name for a computer program, and we can override it by outputting something like hello, instead.

Debugging

We'll focus on it as a HIGHER level concept, an abstraction, known as compiling itself.

So another process that we'll now begin to focus on all the more this week because, invariably, this post week you run against — run up against some challenges.

We're all going to start to make more sophisticated mistakes. So debugging to find the mistakes in your program.

"So what are some of the techniques and tools that folks use? "

Grace Hopper — the one to credit the very 1st discovery of a geeky "actual bug in a computer" this was like a moth that had flown into, at the time, a very sophisticated system known as the Harvard Mark II computer, very large, refrigerator-sized, type systems, in which an actual bug caused an issue. The etymology of bug though, predates this particular instance, but here you have, as any computer scientist might know, the example of a 1st physical BUG in a computer.

"How do you go about resolving such a thing? "