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 past week you ran against-ran up against some challenges.

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

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

Grace Hopper – the one to record 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 removing such a thing?" printf is a wonderfully useful function, not for formatting – printing formatted strings and all that, but just looking inside the

values of variables that you might be curious about to see what's going on.

With printf you can see things! You can see inside of the computer's memory by just printing stuff out like this. 😊

And now, once you've figure it out, so this should probably be less than 3 (or) I should start counting for one!

1° make main.c

debug50 → command that's representative of type of program known as a debugger!

↳ actually built into VSCode.

2° $ debug50 ./main

↳ it runs step-by-step! ♡

and click in line. → will change the interface → the line will be yellow.

→ the little red dot means BREAK there, pause execution here. And the yellow line means has (not) yet been executed.

If I do "play" → "step over"

yellow line moved from line 5 to line 7 because now it's ready but hasn't yet printed out that hash.

LOCALS

↓

there's my variable called i and it's current value is 0.

step over again → in my terminal, one of the hashes has printed. It's still 0 because the yellow highlighted line hasn't yet executed.

So with that I can see my variables changing, I can see output changing on the screen and I can just think about what should not have just happened. 😊

Rubber Duck Debugging

Talk through problems!
Talk through code with someone else!
Look at your code and talk it through.
"Ok, on line 3 I'm starting a 4 loop and

I'm initializing i to 0. Ok, then, I'm printing out a hash, just by talking through your code, step-by-step, invariably, finds you having the proverbial light bulbs go off over your head, because you realize, this is really just a proxy for any other human, teaching fellow, teacher or friend, colleague.

↳ to help you find illogic in what you think might, otherwise, be logical code.

So → printf
→ debugging
→ rubber-duck debugging

} 3 ways to get to the source of code that you will write that has mistakes.

~ ~ ~

get_negative_int

int get_negative_int(void); → I need to put this prototype on top

int → return

void → no input

1° choose my breakpoint;

2° ~~Debug~~ → debug50./buggy;

3° Step Over the line that's highlighted in yellow; → TERMINAL ok. nothing...

4° Step Into → BOOM! now the debugger jumps into that specific function and now I can step through these lines of code again and again. I can see the value of n and I can think through my logic.

Step Over → just goes over the line, but executes it; ≠

Step Into → lets you go into OTHER functions you've written.