DEVELOPMENT 101

INNER WORKINGS AND BASIC PRINCIPLES

COMPUTERS

A computer (in the popular sense) is the combination of a processor and some memory.

A processor is, at a very high level, made up of two “components”
 1) A logic circuit that can perform a set of operations
 2) A (small) number of “registers” upon which those operations are performed.

Memory, or main memory, consists of a (large) number of addressable slots which store values.

BINARY

Computers are, at their core, electronic circuits.
They operate by essentially toggling billions of tiny switches between on and off.

Binary, or base 2, is a convenient representation of these on and off (1 and 0) states.

DECIMAL: $134 = (4 * 10^0) + (3 * 10^1) + (1 * 10^2) = 4 + 30 + 100$
BINARY: $10010000 = 0 + 0 + 0 + 0 + (1 * 2^5) + 0 + 0 + (1 * 2^8) = 16 + 128$

As you can see, we can express any arbitrary number with a sufficiently large number of switches.

Memory, like everything a computer does, is essentially a very large grid of switches.

BITS & BYTES

A bit is literally the smallest chunk of data you can have — it is a simple yes or no, on or off.

A byte, is 8 bits; it can store values between 0 and 255, or 2^8 total values.

That a byte is 8 bits is an “accident of history”; but it has stuck around as a common size.

```
0 and 1 are bits

10101010 and 11111111 are bytes

early computers were “8-bit addressable”

modern computers are “64-bit addressable”
```

BYTES AND WORDS

A word is in many ways more practical than a byte — but also more confusing.

A “word” is the largest number of bits (or bytes) that the processors’ registers can accommodate.

This means that a word can differ in size between computers.

32-bit addressable systems (shortened to “32-bit”) can operate on 32 bits, or 4 bytes.

64-bit systems can operate on 64 bits or 8 bytes.

MEMORY ADDRESSES

The biggest reason that registers kept getting larger is to address more total memory.

With 8 bits, you can address 256 locations that each store 8 bits (2048 total bits).

With 64 bits, you can address a staggering $1.8E^{19}$ locations, each storing 64 bits ($1.18E^{21}$ total bits)

In familiar numbers, the maximum system memory of different addressable sizes:

8-bit:	256 Bytes
16-bit:	128 Kbytes
32-bit:	16 Gbytes
64-bit:	More… just more.

ADDRESSING MEMORY

A memory address is just a number, something between 0 and the maximum total words available to the system.

We we store that number (address) in a variable, we call it a “pointer”, because it points to a point in memory.

Intuitively, any word of memory can be interpreted as an address; accidentally doing so is a common source of programming errors.

```
The following apply to the “C” language

*aThing     says “use aThing as an address”

bThing      says “bThing is a value”

&cThing     says “give me cThing’s address”
```

WORDS AND LETTERS

A word is, as we've seen, just a collection of bits.

We know how numbers are stored in a word.

By mapping letters to those numbers, each word can effectively store a letter.

There are many such mappings, ASCII and UTF8 being common.

Using multiple memory words, we can write actual words.

```
“Character mappings”, let us store words

a = 0
b = 1
c = 2
…
t = 20

[2] [0] [20] => cat
```

ASSEMBLY

Recall that a processor has a set of operations it can perform, these map to “assembly instructions” and “assembly” is a catch-all term for the lowest level type of programming.

So called, because you assemble a chain of instructions (add, subtract, jump to location, etc) into a program. It is extremely rare to work at the assembly level these days, but can be useful.

Assembly maps directly to “byte code”, which is what your code will eventually be compiled down to. Every programming tool and language exists to abstract away the annoyance of working in assembly.

It is, however the purest form of programming, and can be extremely useful for learning.

ASSEMBLY EXAMPLE

A simple assembly program, for a machine with four registers (0-3):
this is an oversimplification

```
1: SET [0, 1]      //set register 0 equal to 1
2: SET [1, 10]     //set register 1 equal to 10
3: ADD [0, 1]      //add 1 to the value of register 0
4: LJMP [0, 1, 3]  //if register 0 is less than register 1, jump back to line 3
```

This program simply increments register 0 from 1 to 10, but these commands map very closely to what the processor's electronics are actually doing

PROGRAMMING LANGUAGES

There are many, many, programming languages.

C is the most universal language, but it has no safety net — it is powerful but easy to mess up.

Objective-C and Java are the next most important languages, for iOS and Android respectively.

Every programming language is simply a combination of convenience layers and syntax that exist solely to make the job and life of the developer easier.

VARIABLES AND STRUCTURES

A variable is simply a named reference to some memory (instead of location 485372, it’s called “user”).

Imagine a programming language that allows you to describe the layout of some memory — a floor plan of sorts.

Then, if you were to store the “type”, and just the address of where to find it, you could store more complex structures in memory while still referring to them by simple named variables.

These of course exist in most every language, and they are surprisingly enough called “structures” or “structs”.

Structures are your friend, as are classes (a special case of structures which we will cover much later).

HEXADECIMAL

Going forward, you will see hexadecimal, which looks like this: 0xF53D

Just as binary is base 2, hexadecimal is base 16 — with the letters A-F filling in for 10-15

It is used because it is far easier to write large values: 1111000011110000 is written as 0xF0F0

The 0x that precedes a hex value simply denotes that the following value is hexadecimal — this is because some hex values (5728) are indistinguishable from regular decimal values.