The importance of CS fundamentals for Developers

Mehdi Cheracher

The "Behind the Scenes" effect

Mehdi Cheracher

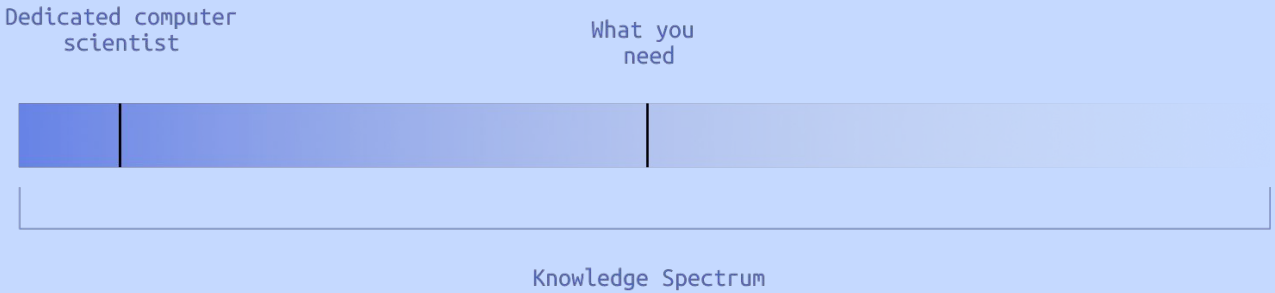
What we will cover

- CS fundamentals
- Examples
- Resources

Housekeeping

- This is an interactive talk, Please ask questions.

Setting expectations



CS Fundamentals

- The most important concepts in computer science
- They are the big pillars in every programming construct
- Allow for quickly build a mental model of things you are not familiar with
    - i.e make you learn faster (not slower as the common misconception)

CS Fundamentals

`Algorithms and Datastructures`

- An applied aspect of problem solving in computer science
- Break complex problems into manageable chunks
- Save money by making things fast / efficient
- Reorganize data in the most suitable way to be accessed by your algorithm
- Talk about performance in a way that it independent from machine specs and environment
- ...

CS Fundamentals

`Algorithms and Datastructures`

- Complexity analysis (Runtime/Memory)
- Translate text to code
- Techniques: Binary/Ternary search, Dynamic programming, Two pointers, Backtracking ...
- Famous algorithms by renowned computer scientists: Kruskal, Dijkstra, Floyd, KMP, Huffman ...
- Implementation of famous DS: BBST, HashMap, LinkedList, Heap ...
- Bonus: SSTable, LSMTree, B+Tree, HyperLogLog ...

CS Fundamentals

`Operating systems`

- Why computers behave the way they do?
- Virtualization
- Concurrency
- Persistence
- ...

CS Fundamentals

`Operating systems`

- Operating system components (mainly linux)
- Processes and Threads
  - Locks
  - Semaphores
  - Memory barriers
  - ...
- Memory management
  - Virtual Memory & Address Translation
  - Memory paging
  - Swap space
  - ...
- I/O Devices
  - Device drivers
  - Filesystems

CS Fundamentals

`Networking`

- Why systems communicate the way they do?
- What are the protocols used for communication?
- Network is unreliable, why?
- ...

CS Fundamentals

`Networking`

- TCP/IP - OSI
- Hardware components (Router, Switch, Nic ...)
- Internet protocols: HTTP, FTP, GrPC, Thrift ...
- ...

CS Fundamentals

`Compilers`

- Understand your program performance
- Easily pick up new languages
- A deeper understanding of the semantics of your preferred language
- Make your own language: (GraphQL, promQL ...)
- ...

CS Fundamentals

`Compilers`

- Lexing and Parsing
- Type checking
- IR
- Dataflow and Control Flow analysis
- Code generation
- ...

CS Fundamentals

`Distributed Systems`

We wish all of our software can fit in a single server that does not break

CS Fundamentals

`Distributed Systems`

- Load balancing
- Map-Reduce
- Sharding
- Partitioning
- Service Discovery
- CAP
- Consensus
- ...

Example: Dependency Injection

- A form of inversion of control
- Automatically manage software dependencies
- Very popular pattern in a lot of programming languages: Spring, Dagger ...

* Note: not a spring expert, whatever I say should be taken with a cup of salt

Example: Dependency Injection

With No CS background:

- Write code that uses the framework efficiently
- Understand most of the constructs that make the framework

But

- Hard to reason about problems faced holistically (i.e independent from the technology/framework used)
- Harder to understand the underlying mechanics of the framework
- Harder to extend the framework
- Harder to transfer knowledge and experience to another tool/framework

Example: Dependency Injection

With a CS background:

- Dependency injection is just a form of **Graph Theory**
- Making a dependency Tree, finding cycles (Algorithms on Graph)

Example: Async IO

- Write asynchronous code like synchronous code while keeping the async semantics.
- async/await keywords (in most programming languages: Rust, Python, Javascript ...)
- Easy to read and reason about
- Performs very well in IO bound workload

* Note: not a async io expert, whatever I say should be taken with a cup of salt

Example: Async IO

With No CS background:

- Use async/await properly
- Somewhat understand performance characteristics

But

- Hard to explain the inner workings of what makes them fast/scalable
- Hard to explain the hoops that the compiler have to jump through to make them behave the way they do.

Example: Async IO

With a CS background:

- IO at the core is **non-blocking**
- `epoll`, `kqueue` with pretty APIs
  a. `(Register interest in an IO source)`
  b. `Wait for notifications`
  c. `Read available data`
  d. `Go back to (a)`

Example: Containers

- Easily develop and deploy applications in a portable way
- Eliminates the "Works on my machine problem"

* Note: not a container expert, whatever I say should be taken with a cup of salt

Example: Containers (docker)

With No CS background:

- Write Dockerfiles compliant with best practices of the industry
- Use they CLI to interact with the docker daemon

But

- Still sees Docker as somewhat magic
- Would be clueless if he had to rewrite a similar technology

Example: Containers (docker)

With a CS background:

- Docker is not magic (Cgroups + Namespaces)
- Would have a solid idea on where to start if ever had to rewrite something similar

Resources

- Open Source Society University
- Video courses list
- Docs are good, Code is better