Some quick demos for fun / preview:

```
0.1 + 0.2;
['data', 'Fullstack', 'Lecture', 'structures'].sort();
[19, 20, 1507].sort();
'🍻'.length;
```

"[The files are] in the computer... it's so simple."
MakeAGIF.com

"[The files are] in the computer... it's so simple."
MakeAGIF.com

# (Just Some) Motivations

- *Demystify foundations*
- *Gain insight & perspective*
- *Recognize solutions*
- *Write performant code*
- *Continue practicing OOP*
- *Interview well*

Contextualization. Standing on shoulders. Using the right tools. Preventing bugs. Continuing to practice object-oriented programming. Speak intelligently with other programmers. A very academic part of the curriculum, but programmers should have a conceptual grasp of this topic. Comp Sci vs. Programming.

# Coming Up

I. Information Theory & Hardware
II. Abstraction & Encoding
III. Software & APIs
IV. Abstract Data Types & Data Structures
  IV.a. Queues
  IV.b. Linked Lists
  IV.c. Hash Tables
  IV.d. Trees
V. Extras

# Part I:
# Information Theory & Hardware

Is this slide communicating anything? If everything was the same, there would be no information (how Zen: all is one, reality is an illusion). But if we add something *different* (click)… now we're cooking. In information theory, info is a sequence of *symbols*, modeling knowledge. Symbols = differentiation; this vs. that.

Is this slide communicating anything? If everything was the same, there would be no information (how Zen: all is one, reality is an illusion). But if we add something *different* (click)… now we're cooking. In information theory, info is a sequence of *symbols*, modeling knowledge. Symbols = differentiation; this vs. that.

Smallest possible unit of info is a *bit*. Not just computer science, but philosophy; no more elementary knowledge than pure differentiation. Perhaps the most profound two-valued type is *truth*. Something "is" or it "isn't". Duality gives rise to complexity.

1847 & 1854, English mathematician George Boole (1815–1864) published *The Mathematical Analysis of Logic* and *An Investigation of the Laws of Thought*. A symbolic notation for formal logic: Boolean algebra. 1/0 denote T/F. Combinatorial axioms: *not True* is False. *not False* is True. True *or* False is True, True *and* False is False. Symbolic logic combines to form complex systems (see: De Morgan's laws). How can we model this in the real world?

1847 & 1854, English mathematician George Boole (1815–1864) published *The Mathematical Analysis of Logic* and *An Investigation of the Laws of Thought*. A symbolic notation for formal logic: Boolean algebra. 1/0 denote T/F. Combinatorial axioms: *not True* is False. *not False* is True. True *or* False is True, True *and* False is False. Symbolic logic combines to form complex systems (see: De Morgan's laws). How can we model this in the real world?

1847 & 1854, English mathematician George Boole (1815–1864) published *The Mathematical Analysis of Logic* and *An Investigation of the Laws of Thought*. A symbolic notation for formal logic: Boolean algebra. 1/0 denote T/F. Combinatorial axioms: *not True* is False. *not False* is True. True *or* False is True, True *and* False is False. Symbolic logic combines to form complex systems (see: De Morgan's laws). How can we model this in the real world?

1930s, American mathematician / electrical engineer Claude Shannon (1916–2001), "the father of information theory", realized Boolean algebra can be processed using switches. He created "switching algebra" — physical logic gates (AND, OR, NOT, NOR, NAND, etc.). On-off states represent truth values.
**Note for Zeke: Shannon also juggled.*

1930s, American mathematician / electrical engineer Claude Shannon (1916–2001), "the father of information theory", realized Boolean algebra can be processed using switches. He created "switching algebra" — physical logic gates (AND, OR, NOT, NOR, NAND, etc.). On-off states represent truth values.
**Note for Zeke: Shannon also juggled.*

1930s, American mathematician / electrical engineer Claude Shannon (1916–2001), "the father of information theory", realized Boolean algebra can be processed using switches. He created "switching algebra" — physical logic gates (AND, OR, NOT, NOR, NAND, etc.). On-off states represent truth values.
**Note for Zeke: Shannon also juggled.*

1930s, American mathematician / electrical engineer Claude Shannon (1916–2001), "the father of information theory", realized Boolean algebra can be processed using switches. He created "switching algebra" — physical logic gates (AND, OR, NOT, NOR, NAND, etc.). On-off states represent truth values.
**Note for Zeke: Shannon also juggled.*

Electrical current is a remarkably useful medium for boolean algebra. Signal propagation is incredibly fast — 50% to 99% light speed. Electrons are tiny (~5 femtometers), so you can fit many signals in a small space. Electric power directly activates devices; signal becomes action. So, an electrical switch is a great model for a bit. And the smallest switch in the world…

…is the metal-oxide semiconducting field-effect transistor (MOSFET). If the CPU is a computer's brain, this is like one neuron. Above: SEM (scanning electron micrograph) of a 2013 Intel Ivy Bridge transistor. Fins only 22 nanometers thick. Viruses 20-400 nm; at 1-5 nm you get weird quantum tunneling. Thumbnail-sized chip contains 1.4 billion transistors, and ~100 km of wiring. Current from the “source” to the “drain” terminal can be switched on/off by changing voltage to the “gate”. Drain of one transistor can be routed to the gate(s) or source(s) of other(s), creating boolean logic gates. Lots of logic gates = a Rube Goldberg-esque calculating machine; a computer!

…is the metal-oxide semiconducting field-effect transistor (MOSFET). If the CPU is a computer's brain, this is like one neuron. Above: SEM (scanning electron micrograph) of a 2013 Intel Ivy Bridge transistor. Fins only 22 nanometers thick. Viruses 20-400 nm; at 1-5 nm you get weird quantum tunneling. Thumbnail-sized chip contains 1.4 billion transistors, and ~100 km of wiring. Current from the “source” to the “drain” terminal can be switched on/off by changing voltage to the “gate”. Drain of one transistor can be routed to the gate(s) or source(s) of other(s), creating boolean logic gates. Lots of logic gates = a Rube Goldberg-esque calculating machine; a computer!

…is the metal-oxide semiconducting field-effect transistor (MOSFET). If the CPU is a computer's brain, this is like one neuron. Above: SEM (scanning electron micrograph) of a 2013 Intel Ivy Bridge transistor. Fins only 22 nanometers thick. Viruses 20-400 nm; at 1-5 nm you get weird quantum tunneling. Thumbnail-sized chip contains 1.4 billion transistors, and ~100 km of wiring. Current from the “source” to the “drain” terminal can be switched on/off by changing voltage to the “gate”. Drain of one transistor can be routed to the gate(s) or source(s) of other(s), creating boolean logic gates. Lots of logic gates = a Rube Goldberg-esque calculating machine; a computer!

…is the metal-oxide semiconducting field-effect transistor (MOSFET). If the CPU is a computer's brain, this is like one neuron. Above: SEM (scanning electron micrograph) of a 2013 Intel Ivy Bridge transistor. Fins only 22 nanometers thick. Viruses 20-400 nm; at 1-5 nm you get weird quantum tunneling. Thumbnail-sized chip contains 1.4 billion transistors, and ~100 km of wiring. Current from the “source” to the “drain” terminal can be switched on/off by changing voltage to the “gate”. Drain of one transistor can be routed to the gate(s) or source(s) of other(s), creating boolean logic gates. Lots of logic gates = a Rube Goldberg-esque calculating machine; a computer!

But what forms the inputs for all those calculations? Where do the original on-off switch states come from? Let’s back way up and talk about data storage…

But what forms the inputs for all those calculations? Where do the original on-off switch states come from? Let’s back way up and talk about data storage…

In 1880, the US census takes 8 years. Herman Hollerith (1860–1929; City College of NY, Columbia, MIT)'s doctoral thesis "An Electric Tabulating System" (1889) proposes a certain kind of data storage, plus electrical signals, to count results. The 1890 census takes one year and comes in under budget. How did he represent the data?

In 1880, the US census takes 8 years. Herman Hollerith (1860–1929; City College of NY, Columbia, MIT)'s doctoral thesis "An Electric Tabulating System" (1889) proposes a certain kind of data storage, plus electrical signals, to count results. The 1890 census takes one year and comes in under budget. How did he represent the data?

Hollerith's punchcards used electrical contact (on-off, wires brushing through holes) and position to indicate age, state, gender, marriage status, etc. He had essentially figured out the electrical bit: on-off switches. In 1911, his Tabulating Machine Company merges to become the Computing Tabulating Recording Company. Guess what the company was renamed in 1924? …International Business Machines Corporation (IBM).

Fast forward to NASA, 1957, where a card reader is still being used for this IBM 704 computer. However, by the 50s punch cards were already on their way out, rapidly being replaced by…

Fast forward to NASA, 1957, where a card reader is still being used for this IBM 704 computer. However, by the 50s punch cards were already on their way out, rapidly being replaced by…

…magnetic tape storage. Magnets have ± polarity. On magnetic tape (e.g. audio cassette, VHS) an electromagnetic read/write head magnetizes sections with one polarity or the other, forming a stream of bits. This is Grace Hopper (1906–1992, Vassar, Yale, USN Rear Admiral) working with a UNIVAC tape store. Invented the first compiler, helped popularize the idea of machine-independent programming languages, credited with the term "debugging" (removed moth from computer).

*NERSC (National Energy Research Scientific Computing Center) modern tape archive system (High Performance Storage System), in use since 1998.* Tape storage is still the most cost & space-efficient massive storage strategy, e.g. for data centers, backup. But it is terribly slow to access different sections.

In antiquity, important documents were recorded in scrolls. **What are some disadvantages of this medium?** Scrolls only offer *sequential access* — you need to *scroll* through everything to reach a specific point. Remember rewinding/fast-forwarding VHS or audio cassettes? Tape drives are essentially scrolls. Imagine the inefficiency in, say, copying data from one spot to another.

*(quote by Roman poet Martial, from his Epigrams.)*
From roughly the 100s through the 300s, the scroll was gradually replaced by the codex (i.e., books). **What are the advantages?** 1) compact/portable, both sides of page; 2) **open different sections quickly** (*random access*). Change from scroll to book was both a populist movement and the most significant leap in data storage until the printing press 1,000 years later (~1440 in the West).* After tape drives, computers needed their own "book"…
**famously used to produce this Gutenberg Bible*

The hard disk drive, invented in the 50s but common by the late 80s. A solid platter ("hard disk") spins at about 5000–10,000 RPM, and the head reads/writes magnetic ± bits in a spiral, somewhat like a record. The head can move to any point on the disk, like a finger choosing which page to open in a book.
*Show video:* https://www.youtube.com/watch?v=3owqvmMf6No

There have been a variety of more portable storage media. Portable magnetic disks include large and small floppy disks. Basically the platter of an HDD without the R/W head, motor, etc. (that stuff goes in the disk drive).

Optical disks have existed for decades (remember LaserDisc?) but didn’t become popular until the 90s. A laser reflects off a "land" surface filled with nonreflective "pits". Trick question: which are the 1s and 0s? Actually, a *change* is a 1, and no change = successive 0s. Downside: hard to change the physical holes.

As the 2000s wore on, a technique used for *volatile* memory (which stops existing after power loss) became practical for non-volatile storage: solid-state or “flash” memory. Popular in small camera storage cards & USB sticks, it is only recently that persistent flash memory has become affordable & large enough for computer drives.

Solid State Drives (SSDs) have no moving parts — they're just micro-circuitry. However, they can maintain their state even without power because they use "floating gate" transistors with electron traps, which can store charge for years. Compared to moving CDs or HDDs they are not only more shock-resistant but far faster. Downsides: poor R/W longevity, complex controller firmware, expense, etc.

So far we've been talking about storage — the most effective ways to keep big amounts of data around in a "safe" (persistent, non-volatile) way. But you wouldn't want to go to your filing cabinet every time you wanted the next page of a document, right? You'd bring that doc to your (admittedly smaller) desk for easier, faster access. This is one distinction between "storage" and "memory."

So far we've been talking about storage — the most effective ways to keep big amounts of data around in a "safe" (persistent, non-volatile) way. But you wouldn't want to go to your filing cabinet every time you wanted the next page of a document, right? You'd bring that doc to your (admittedly smaller) desk for easier, faster access. This is one distinction between "storage" and "memory."

So far we've been talking about storage — the most effective ways to keep big amounts of data around in a "safe" (persistent, non-volatile) way. But you wouldn't want to go to your filing cabinet every time you wanted the next page of a document, right? You'd bring that doc to your (admittedly smaller) desk for easier, faster access. This is one distinction between "storage" and "memory."

# "Must Go Faster"

| Medium | Typical Size (bytes) | Approx. Access Time |
|---|---|---|
| CPU Register | 64 bits = 8 | one cycle < 1 ns |
| L1 Cache | 64 000 | 4–5 cycles ≈ 2 ns |
| L2 Cache | 256 000 | ~12 cycles ≈ 5 ns |
| RAM | 8 000 000 000 | 70 ns |
| Solid State Drive | 256 000 000 000 | 100 000 ns |
| Hard Disk Drive | 1 000 000 000 000 | 12 000 000 ns |
| Optical Drive | 650 000 000 | 150 000 000 ns |
| Tape Drive | 8 000 000 000 000 | 60 000 000 000 ns |
| Network | a big truck? | 6e7 – inf∞ ns |

"Memory"
primary storage
("non-blocking")
volatile
(goes away if power off)

"Storage"
secondary storage
("blocking")
non-volatile
(persists even without power)



A modern CPU might operate at 2 to 4 GHz; that’s 0.5–0.25 ns per cycle. The storage we've talked cannot hope to keep up, so the CPU will be starved for data. The storage-memory distinction is somewhat artificial, based on speed. Same thing for "blocking" vs "non-blocking" — there is a 1:1 correspondence. The CPU can access *main memory*: the *register* directly, *caches* to improve latency for data being currently worked on, and *RAM* for most running processes.

This is a stick of DRAM (Dynamic Random Access Memory). It stores bits in tiny capacitors (not electron traps), which work as long as they have power but zero out on power loss; main difference from SSDs. Processes (running programs) are loaded into RAM to take advantage of high performance. RAM is both fast and big, which is why it's often one of the most significant performance upgrades you can get — assuming the problem is running too many programs.

RAM is pretty fast, but still hundreds of times slower than a CPU. So directly on the CPU die are a number of even *faster* data buckets.

SRAM (static RAM) caches (L3, L2, L1 -> increasing speed, decreasing size) are pure 100% transistor-based — a bunch of on-off switches. That's super fast but very expensive. Caches improve access speed for files predicted to be needed by the CPU. Finally, *registers* can hold a single "word" of data at a time. Registers are the buckets of data the CPU can actually perform operations on; they are scattered all over the CPU. The register size (+ data buses, etc.) limits how the CPU can address memory or perform math. A few years ago, CPUs were mostly 32-bit; now most are 64-bit. This bit size is called a "word" in computer architecture.

SRAM (static RAM) caches (L3, L2, L1 -> increasing speed, decreasing size) are pure 100% transistor-based — a bunch of on-off switches. That's super fast but very expensive. Caches improve access speed for files predicted to be needed by the CPU. Finally, *registers* can hold a single "word" of data at a time. Registers are the buckets of data the CPU can actually perform operations on; they are scattered all over the CPU. The register size (+ data buses, etc.) limits how the CPU can address memory or perform math. A few years ago, CPUs were mostly 32-bit; now most are 64-bit. This bit size is called a "word" in computer architecture.

SRAM (static RAM) caches (L3, L2, L1 -> increasing speed, decreasing size) are pure 100% transistor-based — a bunch of on-off switches. That's super fast but very expensive. Caches improve access speed for files predicted to be needed by the CPU. Finally, *registers* can hold a single "word" of data at a time. Registers are the buckets of data the CPU can actually perform operations on; they are scattered all over the CPU. The register size (+ data buses, etc.) limits how the CPU can address memory or perform math. A few years ago, CPUs were mostly 32-bit; now most are 64-bit. This bit size is called a "word" in computer architecture.

SRAM (static RAM) caches (L3, L2, L1 -> increasing speed, decreasing size) are pure 100% transistor-based — a bunch of on-off switches. That's super fast but very expensive. Caches improve access speed for files predicted to be needed by the CPU. Finally, *registers* can hold a single "word" of data at a time. Registers are the buckets of data the CPU can actually perform operations on; they are scattered all over the CPU. The register size (+ data buses, etc.) limits how the CPU can address memory or perform math. A few years ago, CPUs were mostly 32-bit; now most are 64-bit. This bit size is called a "word" in computer architecture.

# Part II:
# Abstraction & Encoding



So we have storage of physical bits. But how do bits translate to useful information? We've already talked about Boolean values, which may be encoded in a single bit. What else?

```
11101000 00011110 11010100 10010110 01000010
10101111 11000100 10001010 11011101 10011111
11101111                            11011111
01101101 There are only 10 types of 00000100
00100000 people in the world: those 10001111
00110010 who understand binary, and 11111111
01110010 those who do not.          01011100
00110111                            00110001
11000110 10100011 01101010 10010110 11000111
00001000 10010001 00101011 11011101 11100101
01101110 00101011 00011100 10011111 01101001
01010111 00100111 11111010 11111000 10111011
01111000 11000010 01000110 00100000 01111101
```

Instead of boolean logic values (T/F), bits can of course represent ones and zeroes. Ones and zeroes are not very interesting as far as data goes, but with just ones and zeros you can represent *numbers* — via binary notation.

# Using {0, 1} to Represent [0, 255]

| Physical State | OFF | ON | ON | OFF | OFF | ON | ON | OFF |
|---|---|---|---|---|---|---|---|---|
| Binary Notation | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| Order of Magnitude | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| Decimal Value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Applicable Value | 0 | 64 | 32 | 0 | 0 | 4 | 2 | 0 |
| Total Decimal Value | 102 = 64 + 32 + 4 + 2 | | | | | | | |



Why right to left? Because we took numbers from Arabic. They write 100 (show R-L, "0" "0" "1") — we borrow the same numbers, but we inscribe them as we write from left to right (going down powers). More natural to do it the Arabic way (write increasing powers) but imagine the confusion if the final visual result was opposite across cultures. [Show on whiteboard / Wacom]. This problem of order is known as “big-endian” vs “little-endian.” Anyway, 8 bits as shown here is conventionally called a “byte.” This begs the question — how many bits should we use to store a number? Can be anything, but clearly one "word" is a good size.

So we have a basic way to represent the Natural numbers {0, 1, 2 …}. But how can we deal with negative numbers, or fractions? In general, the Rational numbers?

# IEEE 754: Floating-Point Signed Double

- 64 total bits ("*double*" the previous standard word size of 32 bits)
- 52 of the bits are the *fraction* component of the *significand*. There is an implicit "1" bit. Decimal example: in 1.5204 x 10^4, the fraction is 5204.
- 11 bits are *exponent* (where the bicimal point should "*float*"). Many special rules.
  - Encoding patterns for ± 0, ± infinity, NaN, etc.
- One bit is *sign* (0 positive, 1 negative). "*Signed*" = a bit used to store ±.



**Don't panic, we are not testing you on this!** With just numerals, we can only represent integers and SOME rational numbers. But using a bicimal point and a good number of digits we can get close to any rational. **DEMO**: 0.1 + 0.2 in console. 0.1 decimal in binary is 0.0001100110011001100…. Side note: all fixed-length binary can be represented as fixed-length decimal (with enough digits), but not always reverse. Example of a base-n val that cannot be expressed in decimal: 1/3 (0.1 in base 3, 0.333333333… in base 10).

# Hexadecimal Notation

| Hex. digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dec. notation | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Bin. notation | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

- Base-16: uses sixteen different digits per "place" (order)
- To avoid ambiguity: prefaced 0x or U+ or #, or subscript$_{16}$
- Much easier to convert b2 to/from b16 than to/from b10
- Student question: what is 0x2a (i.e. $2a_{16}$, U+2a) in b10?



**Don't panic, we are not testing you on this!** For more compact representations of values, e.g. memory addresses, programmers often use base-16. The advantage vs. base-10 is that conversion to and from base-2 ends up being much easier. A quick exercise for you and your neighbor: what is 0x2a in base-10? Solution: 2 sixteens + "a" ones = 32 + 10 = 42 (aka the answer to everything). What about base-2? Answer: just concat binary "quartets": 0010 followed by 1010 => 101010.

Numbers are great, but what if we want to store something a little more communicative… like text?

At least one solution from the 1840s: Morse. An electronic, binary representation of text. Patterns of dots and dashes correspond to letters. Based more on the physical demands of hitting a switch (common letters used fewer bits; E is just "dot") than on anything more meaningful.

Now that we have binary integers, it's far more convenient to think in terms of numbers rather than arbitrary bits. We can use an agreed-upon *encoding scheme* that maps letters to/from numbers.

# ASCII

- American Standard Code for Information Interchange (1960s)
- 7-bit unsigned integers (stored in 8 bits) = 128 chars… +?

| Binary | Decimal | Glyph | Binary | Decimal | Glyph |
|---|---|---|---|---|---|
| 0010 1110 | 46 | . | 0011 1010 | 58 | : |
| 0010 1111 | 47 | / | 0011 1011 | 59 | ; |
| 0011 0000 | 48 | 0 | 0011 1100 | 60 | < |
| 0011 0001 | 49 | 1 | 0011 1101 | 61 | = |
| 0011 0010 | 50 | 2 | 0011 1110 | 62 | > |
| 0011 0011 | 51 | 3 | 0011 1111 | 63 | ? |
| 0011 0100 | 52 | 4 | 0100 0000 | 64 | @ |
| 0011 0101 | 53 | 5 | 0100 0001 | 65 | A |
| 0011 0110 | 54 | 6 | 0100 0010 | 66 | B |
| 0011 0111 | 55 | 7 | 0100 0011 | 67 | C |
| 0011 1000 | 56 | 8 | 0100 0100 | 68 | D |
| 0011 1001 | 57 | 9 | 0100 0101 | 69 | E |



**Don't panic, we are not testing you on this!** ASCII from teleprinter code used at Bell. Lots of obsolete control characters now. Spec'd for 7 bits, but stored in 8-bit bytes; the "extra" bit has been used for myriad purposes (language-specific, extra glyphs, checksums), causing compatibility problems. Side note, number glyphs are 011 + binary representation, caps are 100 + binary order, lowercase are 110 + binary order. DEMO: `['data', 'Fullstack', 'Lecture', 'structures'].sort()`. Is JS using ASCII? Well, sort of, but not quite; we'll get to this a bit later.

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.
*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*
*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|
| I | U+0049 | 0100 1001 | | | |



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.
*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*
*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|
| I | U+0049 | 0100 1001 | | | |
| ❤ | U+2764 | 1110 0010 | 1001 1101 | 1010 0100 | |



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.

*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*

*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|
| I | U+0049 | 0100 1001 | | | |
| ❤ | U+2764 | 1110 0010 | 1001 1101 | 1010 0100 | |
| J | U+004A | 0100 1010 | | | |



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.

*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*

*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|
| I | U+0049 | 0100 1001 | | | |
| ❤ | U+2764 | 1110 0010 | 1001 1101 | 1010 0100 | |
| J | U+004A | 0100 1010 | | | |
| S | U+0053 | 0101 0011 | | | |



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.

*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*

*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-8

- *Universal Coded Character Set* + *Transformation Format* – 8-*bit*
- Handles every *code point* of **Unicode** (1.1M+; ~260K assigned)
- *Variable-width*: one to four 8-bit unsigned integers (*code units*)
- Superset of ASCII — backwards-compatible (best for web)

| Glyph | Unicode Code Point | Code Unit 1: *singleton* or *lead* | Code Unit 2: *trail 1* | Code Unit 3: *trail 2* | Code Unit 4: *trail 3* |
|---|---|---|---|---|---|
| I | U+0049 | 0100 1001 | | | |
| ❤ | U+2764 | 1110 0010 | 1001 1101 | 1010 0100 | |
| J | U+004A | 0100 1010 | | | |
| S | U+0053 | 0101 0011 | | | |
| 😄 | U+1F604 | 1111 0000 | 1001 1111 | 1001 1000 | 1000 0100 |



**Don't panic, we are not testing you on this!** 7-bit ASCII and even extended 8-bit ASCII are obviously not enough for really rich character encoding. Enter UTF-8, which encodes Unicode characters as one to four 8-bit *code units* per *code point*. This is the preferred scheme for web because it is backwards-compatible with ASCII; a UTF-8 file with only ASCII values is identical to an ASCII file. Variable-width encoding is more space-efficient than fixed-length.
*Code point: an entity from unicode. For example, LATIN CAPITAL LETTER A. Listed as a hex value.*
*Code unit: a byte which may make up all or part of the actual binary encoding for a code point.*

# UTF-16

- Variable-width: one or two 16-bit unsigned integers
- Incompatible with ASCII (no 8-bit blocks)
- byte vs. word stream issues
- byte order ambiguity
- space inefficiency for most languages, even C/J html
- (Sort of) predated UTF-8, which is why stuff inherits it



UTF-16 has almost no advantage over UTF-8. When you have a choice, use the latter.

We have booleans (1 bit), signed floating-point numbers (including 64-bit), variable-length string encodings (e.g. UTF-8). What else? How about images?

The human retina has three types of color-sensing cone cells, each with peaks at different wavelengths. Three separate colors added together (i.e. too close for the human eye to differentiate) are all that is needed to stimulate all three types of cells to variable degrees — producing the sensation of any color in the visible spectrum. Computer monitors are grids of pixels which each consist of RGB sub-pixels that can be adjusted to different brightnesses.

The human retina has three types of color-sensing cone cells, each with peaks at different wavelengths. Three separate colors added together (i.e. too close for the human eye to differentiate) are all that is needed to stimulate all three types of cells to variable degrees — producing the sensation of any color in the visible spectrum. Computer monitors are grids of pixels which each consist of RGB sub-pixels that can be adjusted to different brightnesses.

The human retina has three types of color-sensing cone cells, each with peaks at different wavelengths. Three separate colors added together (i.e. too close for the human eye to differentiate) are all that is needed to stimulate all three types of cells to variable degrees — producing the sensation of any color in the visible spectrum. Computer monitors are grids of pixels which each consist of RGB sub-pixels that can be adjusted to different brightnesses.

# Image Encoding

- Simple *bitmaps* are triads of 8-bit unsigned ints for RGB values
- 8 bits makes 256 possible brightnesses per *channel*
- "24-bit" (8 + 8 + 8) monitors have 256^3 = ~16.8M RGB distinct channel combinations ("colors").
- Since grayscale means RGB channels have to be the same, that's still only 256 possible B&W values.



When monitors talk about "bit-depth" they mean how many bits per color channel (R, G, and B); standard modern consumer monitor uses 8-bits per channel, resulting in ~16.8 million color combos. Surprise: many laptop screens are actually 6-bit, but use clever flickering mechanism (FRC) to simulate 8 bits by jumping back and forth between colors. This can result in odd patterns sometimes.

# Audio Encoding

- Simplest possible is raw PCM (pulse-code modulation)
- 44,100 *samples* per second; each sample 16-bit *amplitude*
- Different amplitudes over time models an audio wave
- DAC converts values to speaker cone analog signal



705.6 Kbps mono, 1.411 Mbps stereo uncompressed

# Part III:
# Software & APIs



[BREAK?]. We've established how bits are stored, how the CPU accesses them, and what kinds of things we can encode with them. But what's doing the encoding, and how can we take advantage of this?

# Machine Code & Assembly

- Every CPU has its own *instruction set* — numerical (binary) codes that tell the CPU to perform operations on registers.

| MIPS example | operation code | register source | register target | register destination | shift amount | function type |
|---|---|---|---|---|---|---|
| instruction | 000000 | 00001 | 00010 | 00110 | 00000 | 100000 |
| hex | 0x0 | 0x1 | 0x2 | 0x6 | 0x0 | 0x20 |
| decimal | 0 | 1 | 2 | 6 | 0 | 32 |
| meaning | arithmetic | register 1 | register 2 | register 6 | no offset | addition |
| English | *"add the value in register 1 to the value in register 2 and place the result in register 6"* | | | | | |
| assembly | `add $t6, $t1, $t2` | | | | | |



Raw numerical machine code is what ultimately drives all CPUs. Streams of binary words trigger a CPU to follow certain instructions (based on that CPU's instruction set). For example, in the MIPs architecture, this 32-bit code (00000000001000100011000000100000) tells the CPU to add up the values in registers 1 and 2 and place the result in register 6. Writing programs as a sequence of 1s and 0s is exactly as painful as you imagine, so people came up with *Assembly* languages — strings that "assemble" directly into machine instructions, with a 1:1 correspondence.

# Compilers & Interpreters

- **First compiler written in 1952 by Grace Hopper, for A-0.**
- **She** *"helped teach the machines a language, stopped them from speaking in undecipherable numbers."* — 60 minutes
- **Followed by Mark 1, FORTRAN, COBOL, etc.**

Assembly is still extremely laborious and error-prone — human beings don't think like abacuses. Compilers let programmers use human syntax which may then have to be converted into a series of dynamic machine codes (no more 1:1 correspondence). The distinction between compilation and interpretation is complex and blurry, but in general people think in terms of compiling ahead of time (resulting in a pre-optimized machine code file) vs. interpreting at run time (translating the code on demand). Compilers & interpreters allowed for the type of declarative programming languages we now all use.

On a simple computer, pure object code might suffice. But that is somewhat restrictive. What if you want to manage running multiple programs simultaneously? Or access various 3rd-party hardware devices (like hard drives or graphics cards)? An *operating system* abstracts away and standardizes accessing hardware and running multiple programs by providing an API of *system calls* that binary code can use.

# More nuanced / complex...

based on *Advanced Linux Programming*

# Languages & Memory Management

- Languages are a *specification*
- Compilers generate the real instructions to alter registers
- Some languages let you instruct the compiler to attempt to alter memory (C)
- Others only provide abstract *data types*, and it is up to the compiler/interpreter to figure out what to do (JS)

```
char ch = 'c';
char t;
char *chptr = &ch;
t = *chptr;
```

```
var ch = 'c';
var t = ch;
```



Note: even C programs may not be able to directly alter memory — if they are compiled to an executable for an operating system, what they really do is ask the OS to do various memory operations and the OS will oblige.

# Part IV:
# Abstract Data Types
# & Data Structures



[BREAK?]. We've established how bits are stored, how the CPU accesses them, and what kinds of things we can encode with them. But what's doing the encoding, and how can we take advantage of this?

Data structures can typically be stored either in one big block (contiguous) or in many small blocks, each with a memory address for the next block (linked). What are some pros and cons of each approach? Contiguous pros: good physical locality of memory means faster in practice. No wasted space for memory addresses. Linked pros: infinite adding so long as there are open spaces big enough for element + address.

# ADTs vs DSs

| Common Abstract Data Types | (Some) Data Structures |
|---|---|
| Set | List, Linked Tree, Trie, Hash Table, etc. |
| List | Linked List, Array |
| Stack | Array, Linked List |
| Queue | Linked List, Deque |
| Map (Associative Array / Dictionary / etc.) | Hash Table, Association List, Red-Black Tree |
| Graph | Adjacency List, Adjacency Matrix |
| Tree | Linked Tree, Heap |



An ADT is a description of:
1. how information is related (e.g. ordered elements, connected elements)
2. operations we can perform on that information (e.g. add, remove, find)

A DS is a concrete, programmatic implementation of an ADT that determines:
1. how performant the operations actually are

# The Stack ADT

- Collection of elements
- Ordered
- Elements can repeat (not a set)
- Some example operations:
  - Make a new stack
  - Add an element to the stack ("push")
  - Retrieve an element from the list ("pop") - *must be LIFO (Last In, First Out)*
  - Check if stack is empty
  - Look at the top element without removing it ("peek")
  - Clear the stack



Stacks are useful for a couple key applications in programs. Remember the RPN calculator? Also very useful for backtracking (command-z, anybody?), holding old values waiting to be processed (*ahem*, call stack or recursion), etc.

# The Array Data Structure (non-JS*)

| Pseudocode | Compiler/Intepreter | Memory |
|---|---|---|
| vals = int8 Array(5) | `vals` is 5 bytes at 0x3A6 | _ _ _ X X X X X _ _ _ _ |
| vals[0] = 9 | put `9` at 0x3A6 + 0 = 0x3A6 | _ _ _ 9 X X X X _ _ _ _ |
| vals[3] = 4 | put `4` at 0x3A6 + 3 = 0x3A9 | _ _ _ 9 X X 4 X _ _ _ _ |
| vals[2] // undef. | what is at 0x3A6 + 2 = 0x3A8 | _ _ _ 9 X X 4 X _ _ _ _ |
| vals[3] // 4 | what is at 0x3A6 + 3 = 0x3A9 | _ _ _ 9 X X 4 X _ _ _ _ |



Now we'll take a look at a specific implementation of the stack — the fixed, typed array. In true arrays, data is stored sequentially in physical memory. Here we see a hypothetical array of 8-bit (1-byte) integers. Every time we access an index of the array, what the interpreter does is simply add that index arithmetically to the starting memory address of the array. Then it can go straight to that memory address. Notice: no matter which space we want to access, it takes the same ("constant") amount of time. Also, arrays have good "locality" — the data is all side-by-side in the physical machine. This aids in performance, e.g. for iterating through the array.

# The Stack ADT & Array DS

| Stack Feature / Operation | Array Implementation with `top` |
|---|---|
| Collection of elements | Store values at memory addresses |
| Ordered | Sequential addresses maintain ordering |
| Create new stack | Initialize a new array |
| Push onto stack | Insert value at `top` var (next index to use) |
| Pop off of stack (LIFO) | Use `top` index to return latest value |
| Check if stack is empty | Return whether `top` variable is 0 |



By doing some minimal accounting using a variable (like `top`), we can implement the Stack ADT using a true array fairly easily. In fact, the `push` and `pop` methods of a JavaScript array basically use its `.length` property in a similar way. Thought: how performant is pushing and popping using an array to implement our stack? Well, we know that accessing an array element by index (either reading or writing) is a constant-time operation (i.e., it doesn't matter which index we use, it will always take the same time give or take). That's excellent!

WHAT ABOUT JS?

## JavaScript Primitive (Immutable) Data Types

- `Undefined` (default for unassigned vars / nonexistent props)
- `Null` (`typeof` is wrong! Null is a primitive, not "object").
- `Boolean` (could be 1 bit, but may not be)*
- `String` (ES: engines use UTF-16)†
- `Number` (ES: 64-bit signed double float)‡
- `Symbol` (ES6: a unique token, can be key for objects).



* often stored in a byte

† Spec can produce surprising results: `'🍺'.length === 2` . Astral plane characters are "surrogate pairs" of 16-bit code units: `'🍺' === '\uD83C\uDF7A'` . Mitigated in ES6 with Unicode point notation: `` `\u{1f37a}` ``.

‡ JS numbers are specified as 64-bit signed floats; IEEE 754 except that all distinct NaN are treated as a single NaN. HOWEVER, in V8 they are pairs of 32-bit numbers. Also, when using bitwise operators, spec says they should be treated as signed 32-bit twos-complement values.

° **Main point, the interpreter may be implemented in a variety of ways, so long as the behavior "seems" right; if someone asks you how JS data types are implemented, "it depends" is a pretty good answer.**

# The JavaScript Non-Primitive Data Type

- `Object`
  - Mutable (can change)
  - Collection of *properties* – key-value pairs (map!)
  - Variable assignment copies reference, not value; it "points to" the object
  - Other types are actually objects. Examples:
    - **Function** (special: can be invoked)
    - "**Array**" (sort of — can be true array or hash table in V8)
    - **Typed Array** (ES6-only, & can be implemented by engine as a bit buffer)
    - **Regular Expression**, **Date**, others
    - Constructed primitives (**String**, **Number**, **Boolean**, etc.)

Demo — copying object refs, changing object, reflected in different places (vs. same for primitives). Also, demo constructed primitives.

QUEUES &
LINKED LISTS

# The Queue ADT

- Like Stack, except FIFO (First In, First Out)
- Collection of elements
- Ordered / sequential
- Operations:
  - Enqueue (add)
  - Dequeue (remove)
  - Peek
  - Clear
  - IsEmpty… etc.

Queues are a natural fit for handling incoming values in a fair way (buffering). In the workshop today you will be implementing the Queue ADT, initially using JavaScript's "array" fundamental type. However, for the academic practice, treat those arrays like "true" arrays and do not use any of the built-in JS helper functions on `Array.prototype`. That means no push, pop, shift, unshift; also, do not use `.length`. The only thing you really have access to is indices. Note that since Queues are an ADT, not a DS, that implies there is more than one way to skin a cat.

# The Linked List DS

- Data structure used for *list, stack, queue, dequeue* ADTs etc.
- Uses *nodes* which encapsulate a value and pointer(s)
- Main entity holds reference(s) to just a head and/or tail node
  - the "handle(s)"
- Each node then *points* to the next and/or previous node
  - "singly-linked" (unidirectional) vs. "doubly-linked" (bidirectional)

# Linked List

Head:



When it comes to implementations, the LinkedList is often considered synonymous with its handle or handles (in this case just an empty "head" reference).

This is a Linked List containing one node. The node has a value (Omri) and an empty `next` pointer. In this case it only has one pointer (no `previous`) so this is going to be a singly-linked list.

Partway through adding to the tail — we have created a new Ayana node, but it isn't connected to our list yet.

Set the Omri node's `next` pointer to the Ayana node, and we would say that Ayana is the new tail node. Can I get to Omri from Ayana? If I want to reach Ayana, I have to start at `head` and then follow `next`.

What if I want to add something in the middle of the list?

The best thing about Linked Lists is that *insertion* (adding in the middle) always takes the same number of steps, regardless of how big the list is. Again, we call that "constant time." First, change the previous node's `next` to point to the new node.

Then, point the new node's `next` to the old later node.

Then, point the new node's `next` to the old later node. That's it!

Here is an example of a doubly-linked list with head and tail pointers. Think about the pros and cons of this approach.

# (some) pros/cons: Linked Lists vs Arrays

| Operation | Linked List | Typed Array |
|---|---|---|
| Reach element in middle | Must crawl though nodes | Constant time |
| Insert in middle or start | Constant time (if we have ref). | Must move all following elements |
| Add element to end | With handle, constant time | Constant time |
| Space per element | Container + element + pointer(s) | Just element! |
| Total space | Grows as needed | Pre-reserved & limited* |
| Physical locality | Not likely | Best possible |



**Dynamic* arrays keep track of their limit, and if it is exceeded, the array is copied into a new bigger array (which is expensive but hopefully happens infrequently).

Part 1 of the workshop! Do Queues & Linked Lists. The next two specs — Hash Tables and Binary Search Trees — are part of the next phase, which we'll be doing tomorrow.

Part 1 of the workshop! Do Queues & Linked Lists. The next two specs — Hash Tables and Binary Search Trees — are part of the next phase, which we'll be doing tomorrow.