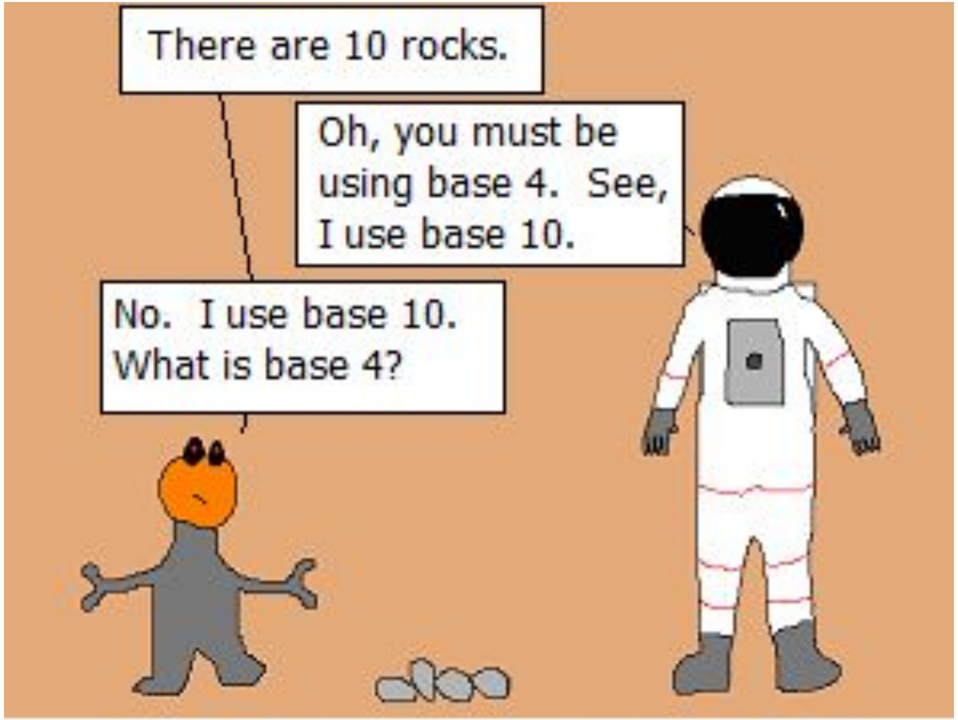
CS61C: Great Ideas in Computer Architecture
(Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Last Time...



Every base is base 10.

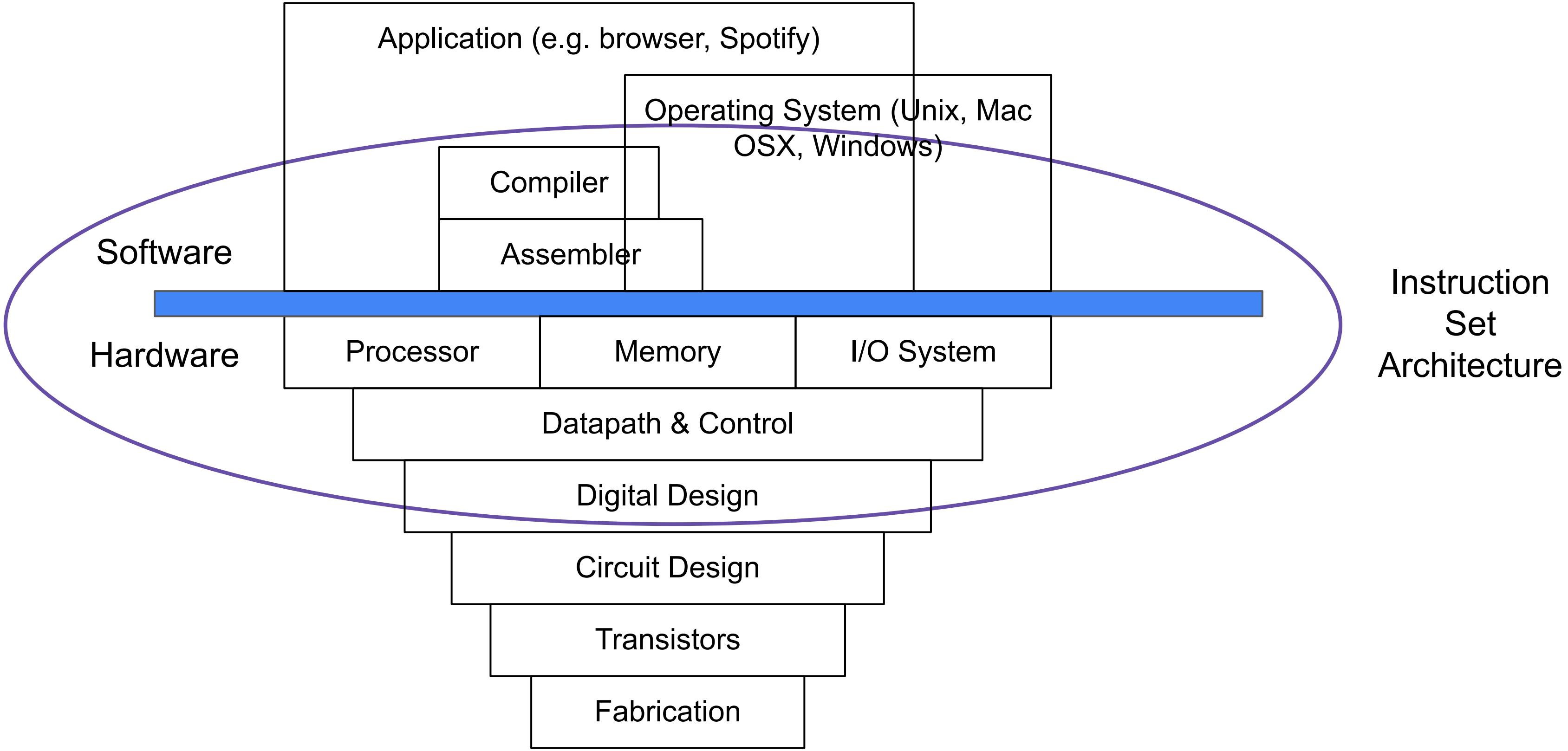
P.S. signal processor's relation to sign-and-magnitude system: No arithmetic needed, data centered around 0

Why study Computer Architecture?

61C is NOT about C programming

- It is about the hardware-software interface
  - What does the programmer need to know to achieve the highest possible performance
- Languages like C are closer to underlying hardware, unlike languages like Python or Java
  - We can talk about hardware features in higher-level terms
  - Allows programmer to explicitly hardness underlying hardware parallelism for high performance

Old School 61C



New School Machine Architecture

Software

Parallel Requests
Assigned to computer
e.g., Search “Cats”

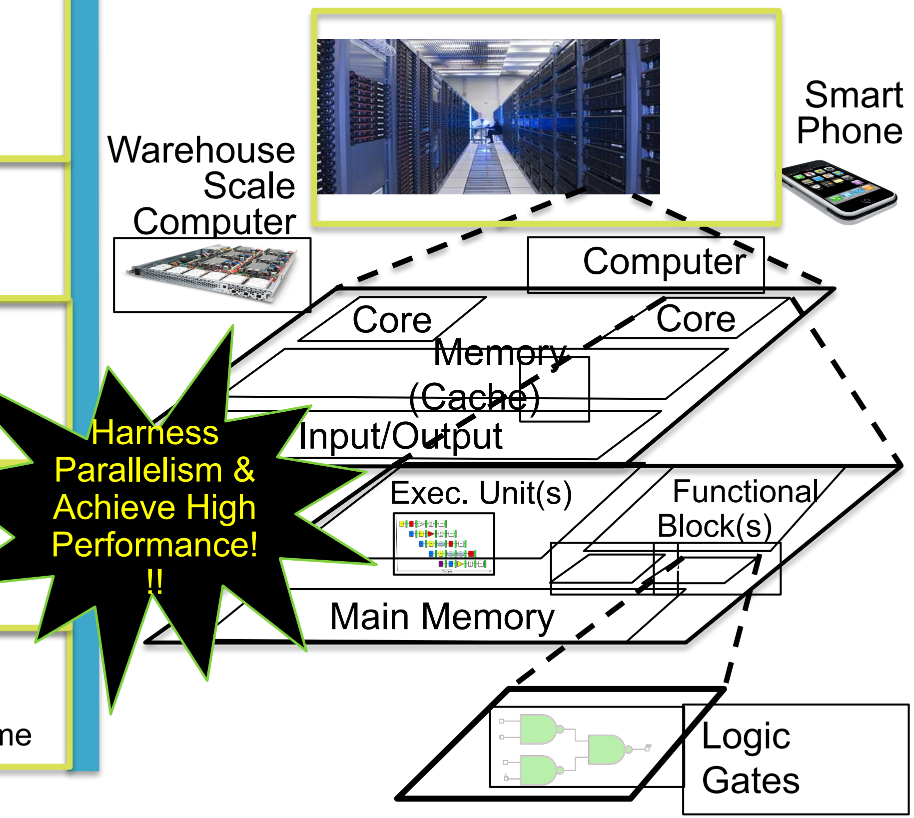
Parallel Threads
Assigned to core e.g., Lookup, Ads

Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions

Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words

Hardware descriptions
All gates work in parallel at same time

Hardware



6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via redundancy

Great Idea #1: Abstraction
(Layers of Representation/Interpretation)

CS61C

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw   x3, 0(x10)
lw   x4, 4(x10)
sw   x4, 0(x10)
sw   x3, 4(x10)
```

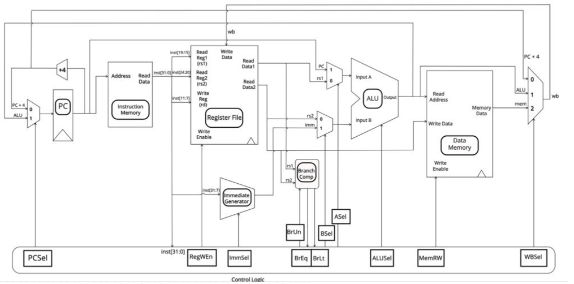
Assembler

Machine Language Program (RISC-V)

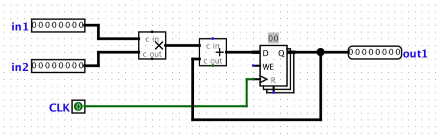
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Anything can be represented as a number, i.e., data or instructions

Hardware Architecture Description (e.g., block diagrams)

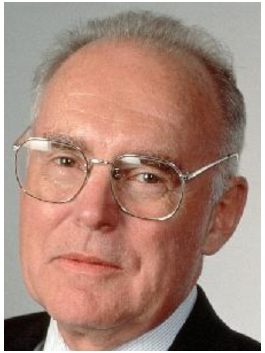


Logic Circuit Description (Circuit Schematic Diagrams)



Great Idea #2: Moore’s Law

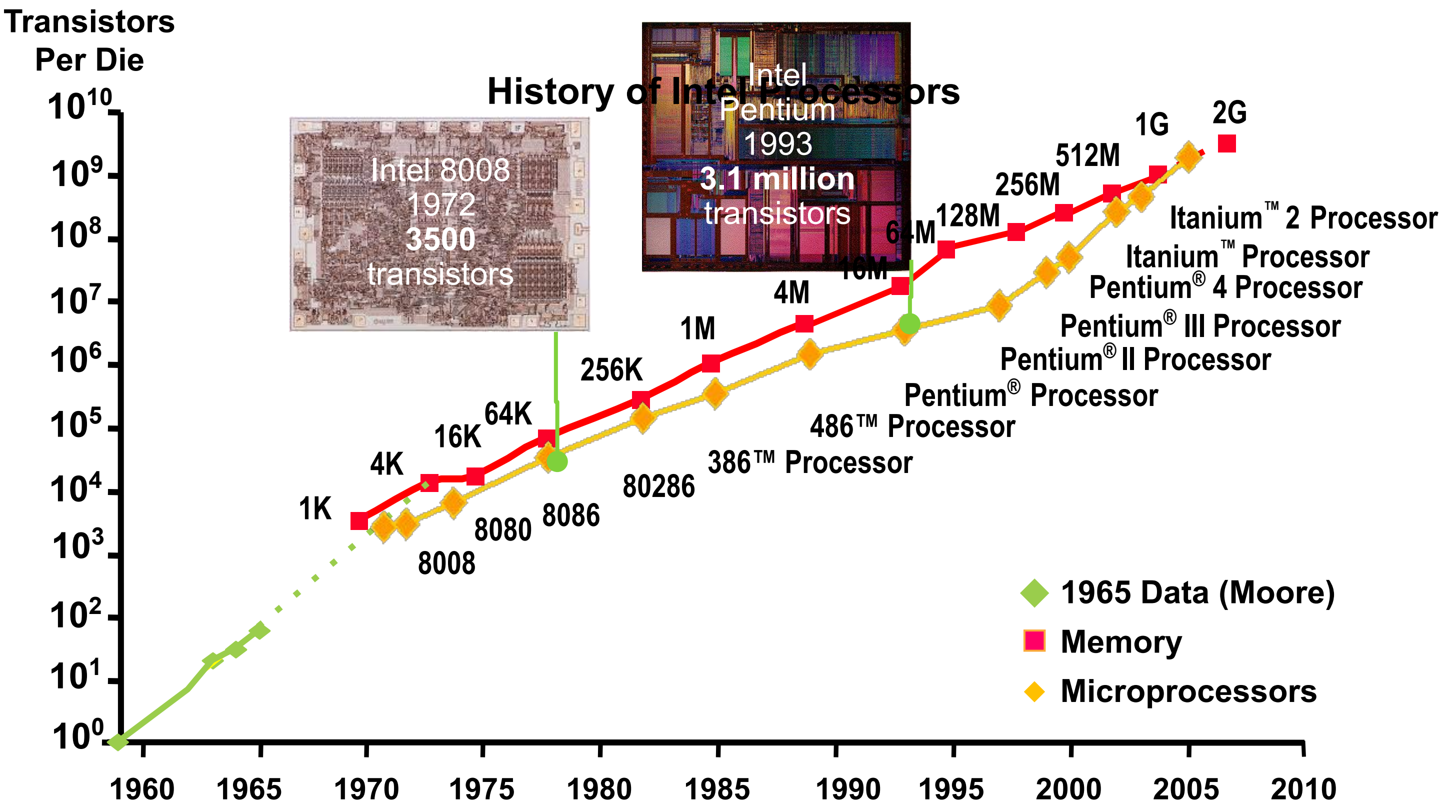
- Moore's law: idea that # of transistors (a base hardware component of integrated circuits) doubles every 2 years
  - # of transistors in an IC increases exponentially
- Why is this useful?
  - Transistors are used everywhere, from fundamentally building lightning fast on-chip memory, to main memory
  - More transistors ⇒ faster, more compact memory
    - For now...
- Is this sustainable and infinite?
  - We... don't know!
  - Doubling has slowed down since 2010 though!



Gordon Moore
Intel Cofounder
B.S. Cal 1950!

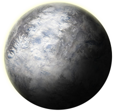
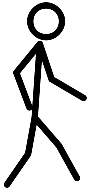
Great Idea #2: Moore's Law

i8008 (1972)
i8080 (1974)
i8086 (1078)
i80286 (1982)
i80386 (386)
i80486 (486)
Pentium (1993)
Pentium II (1997)
Pentium III (1999)
Pentium 4 (2000)
Itanium (2001)
Itanium 2 (2002)

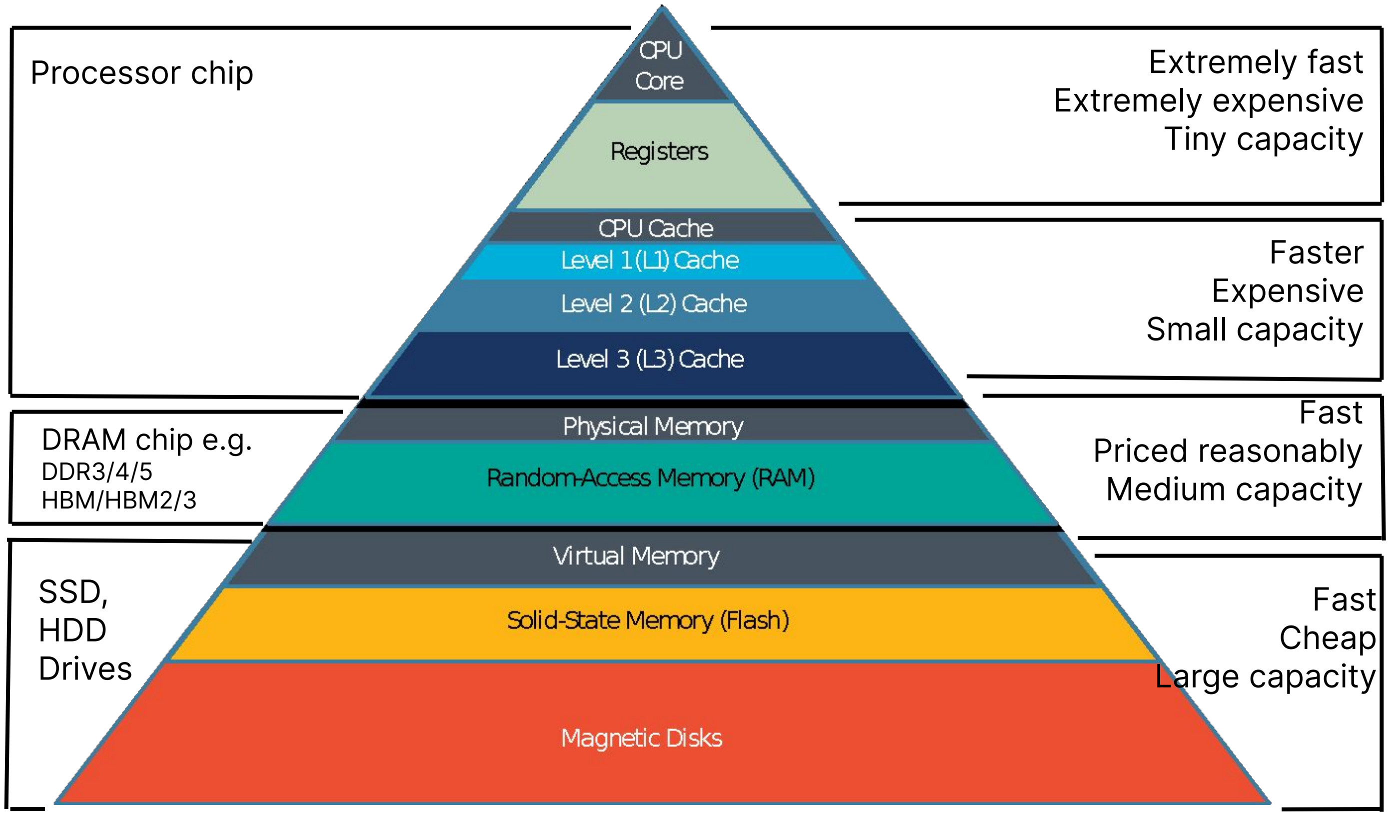


Great Idea #3: Principles of Locality / Memory Hierarchy

- The CPU (central processing unit) is the "center of everything important"
    - Everything is relative to it!
- The "farther" away data is, the slower it is to access!
    - But what does "farther" away mean?
    - In physical distance?
    - Not exactly… but sort of!
- Data "distance" defined as:
    - Physical distance, in some instances
    - Amount of time it takes for a data request to respond with data

10^9 ns	Tape / Optical Robot		2,000 years
10^6 ns	Disk		2 years
100 ns	Main Memory		1.5 hours
10 ns	On-board cache	Berkeley	15 minutes
2 ns	On-chip cache	LKS245	2 minutes
< 1 ns	Registers		< 1 minute

Great Idea #3: Principles of Locality / Memory Hierarchy



Great Idea #4: Parallelism

- What does parallelism mean?
  - Parallelism: the idea of doing more than one thing at the same time.
  - e.g. multi-tasking, breathing + eating, doing 4 loads of laundry in succession
- Where can we see this in computer architecture?
  - In execution of programs instructions (datapath)
  - In multiprocessing (executing multiple programs at once across multiple hardware components)
  - In multithreading (making you think multiple things are happening at once)
- Ultimate goal is for better performance!

Gene Amdahl

Computer Pioneer

So does this mean we can improve performance without limits?

Unfortunately, and unsurprisingly, no. Even with the appropriate hardware and software support, we're limited by **Amdahl's law**.

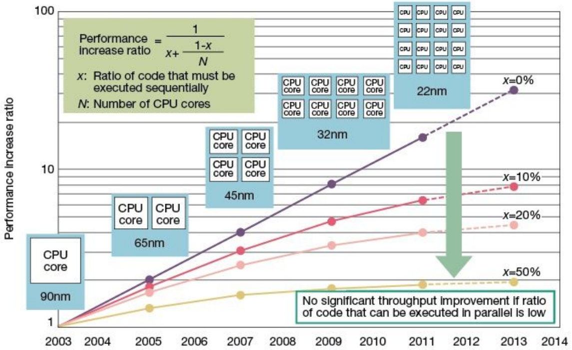


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Great Idea #5: Performance Measurement & Improvement

Matching application to underlying hardware to exploit

- Locality
- Parallelism
- Special hardware features, e.g. specialised instructions

Latency / Throughput

- How long to set the problem up *and* complete it
    - In other words, how many tasks can be completed in a given time.
- How much faster does it execute once it gets going
- Latency is all about time to finish

Great Idea #6: Dependability via Redundancy

Why do we need redundancy?

- It helps make data requests dependable.
- What does that imply?
  - Mistakes and failures at any point in data storage/request will not cause persistent loss of data.

What does this apply to?

- Redundant datacenters: loss of 1 datacenter does not cause Internet downtime
- Redundant disks: loss of disk(s) does not cause loss of data
  - RAID: Redundant Arrays of Independent Disks
- Redundant memory bits: loss of n bits in communication does not result in loss of or mistakes in data
  - ECC: Error Correcting Code

Announcements

Assignments & Due Dates

- Monday, 6/26 — Lab 0: Intro, Setup
- Wednesday, 6/28 — HW1: Number Rep, C (extremely long, start early!)
    - We will cover everything you need for HW1 on Monday, but at least half of it should be doable already.

OH and discussions start today!

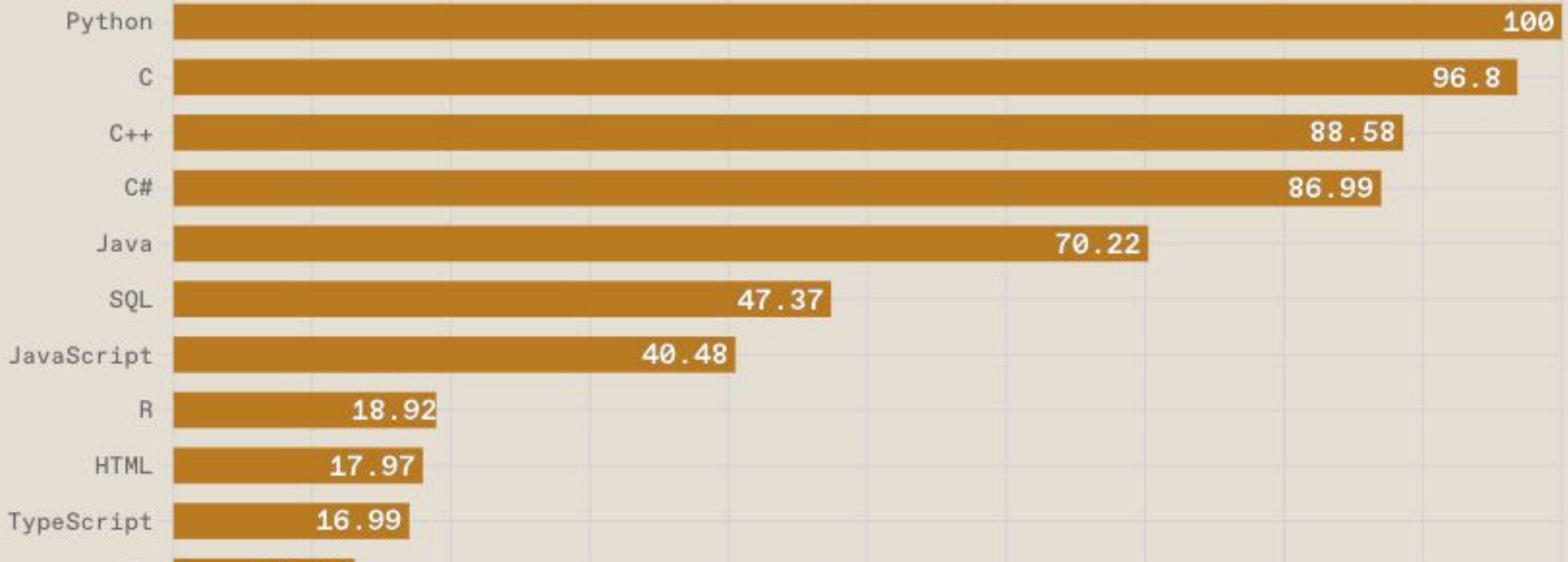
Intro to C Programming

Top Programming Languages 2022

Click a button to see a differently-weighted ranking

IEEE Spectrum
2022-08-23

Job listings are of course not the only metrics we look at in Spectrum. A complete list of our sources is here, but in a nutshell we look at nine metrics that we think are good proxies for measuring what languages people are programming in. Sources include GitHub, Google, Stack Overflow, Twitter, and IEEE Xplore. The raw data is normalized and weighted according to the different rankings offered—for example, the Spectrum default ranking is heavily weighted toward the interests of IEEE members,



Great Idea #1: Abstraction

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

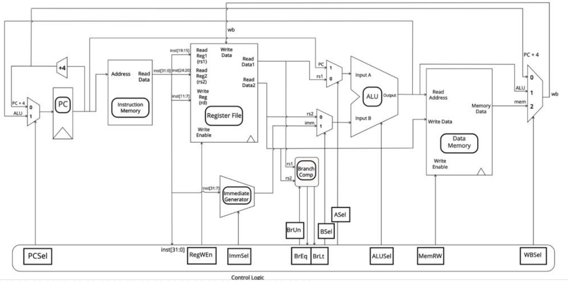
```
lw   x3, 0(x10)
lw   x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

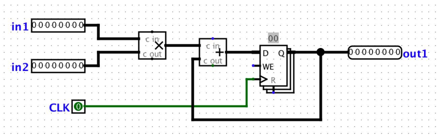
Machine Language Program (RISC-V)

```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Logic Circuit Description (Circuit Schematic Diagrams)



Introduction to C

- “C is not a “very high-level” language, nor a “big” one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.” -K&R
- Enabled the first OS not written in assembly language
- Why study C?
  - We can write programs that allow us to exploit underlying features of the computer architecture
    - Memory management!!
  - However that does also means things can more easily go wrong in C…

How experienced are you with C?

(A) I don't know C, and have not worked with Java or C++

0%

(B) I don't know C, but has coded in Java

0%

(C) I've coded a little bit in C

0%

(D) I've coded a fair amount in C

0%

(E) I've coded a lot in C

0%

Disclaimer

- You will not learn how to fully code in C in these lectures! We will be mostly focusing on the structures of C programs
- Here are some helpful C references
  - K&R
  - Brian Harvey's helpful transition notes
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key concepts
  - Pointers, arrays, memory management
  - All of the above are *unsafe*. If your program contains an error in any of these areas, it may not cause the program to crash immediate, but will leave the program in an inconsistent and exploitable state
  - Use Rust (C but safe!) or Go

How does the system understand our code?
I.e. Compiled vs. Interpreted

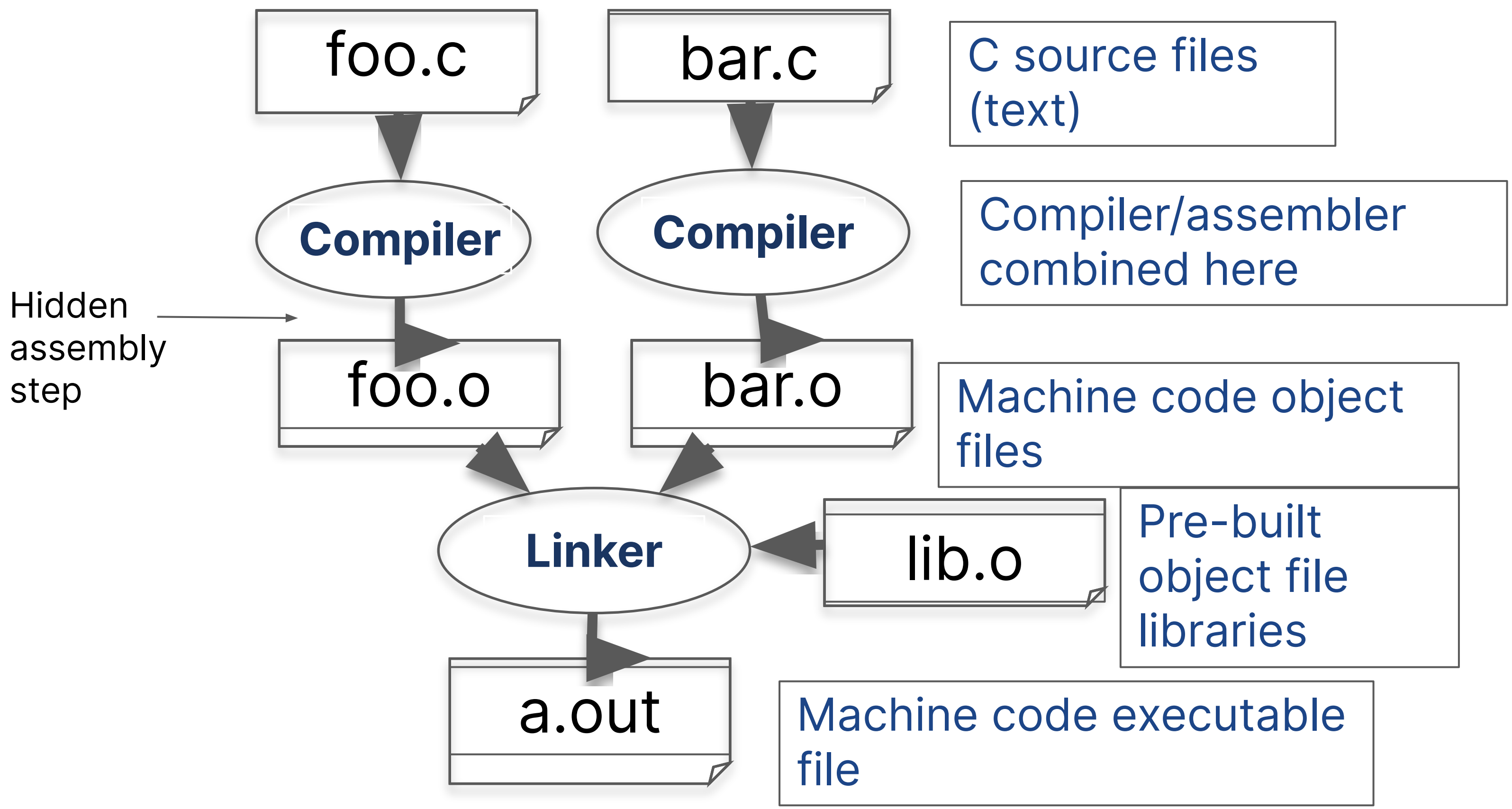
Translation

- We write code in English to some extent, but the system really only sees 0s and 1s. How do we fix that?
  - If someone doesn't understand English, but understands French, we would translate the text!
  - Similar deal in systems, but we translate to a non-human readable language
- Translation happens in two ways
  - Compilation
  - Interpretation
  - Some languages use both!

Compilation: Overview

- C compilers map C programs directly into architecture-specific machine code (string of 1s and 0s)
  - Java converts to architecture-independent bytecode which is then compiled by a just-in-time (JIT) compiler.
  - Python environments converts to Python bytecode at runtime instead of at compile-time.
    - Runtime versus JIT compilation differ in when the program is converted to low-level assembly language that is eventually translated into machine code.
- With C, there is generally a 3-part process in handling a .c file
  - .c files are compiled into .s files ⇒ compilation by the compiler
  - .s files are assembled into .o files ⇒ assembly by the assembler (this step is generally hidden, so most of the time we directly convert .c files into .o files)
  - .o files are linked together to create an executable ⇒ linking by the linker
  - We’ll go into detail in the CALL lecture later

Simplified C Compilation Overview

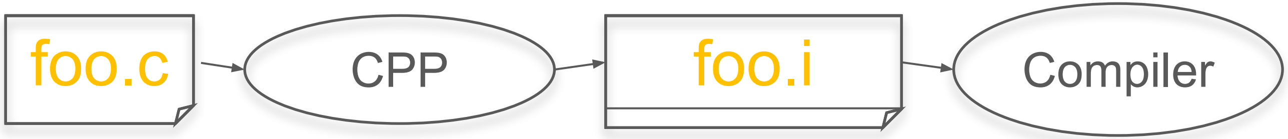


Compilation: Advantages vs. Disadvantages

- Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
  - But these days, a lot of performance is in libraries:
  - Plenty of people do scientific computation in Python!?!
    - they have good libraries for accessing GPU-specific resources
    - Also, many times python allows the ability to drive many other machines very easily …
    - Also, Python can call low-level C code to do work: Cython

- Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. x86 vs. RISC-V) and the operating system (e.g., Windows vs. Linux vs. MacOS)
- Executable must be rebuilt on each new system
  - I.e., “porting your code” to a new architecture
- “Change → Compile → Run [repeat]” iteration cycle can be slow during development
  - but make only rebuilds changed pieces, and can compile in parallel: `make -j`
  - linker is sequential though → Amdahl’s Law

C Pre-Processor (CPP)



- C source files first pass through macro processor, CPP, before compiler sees code
- CPP replaces comments with a single space
- CPP commands begin with "#"
  - **#include "file.h"** /* Inserts **file.h** into output */
  - **#include <stdio.h>** /* Looks for file in standard location, but no actual difference! */
  - **#define PI (3.14159)** /* Define constant */
  - **#if/#endif** /* Conditionally include text */
- Use **-save-temps** option to gcc to see result of preprocessing
  - Full documentation at: **http://gcc.gnu.org/onlinedocs/cpp/**

CPP Macros

- You often see C preprocessor macros defined to create small "functions"
  - But they aren't actual functions, instead it just changes the text of the program
  - In fact, all `#define` does is string replacement
  - `#define min(X,Y) ((X)<(Y)?(X):(Y))`
- This can produce, umm, interesting errors with macros, if **`foo(z)`** has a side-effect
  - `next = min(w, foo(z));`
  - `next = ((w)<(foo(z))?(w):(foo(z)));`

C vs. Java

C vs. Java (1/3)

	C	**Java**
Function	Function Oriented	Object Oriented
Programming Unit	Function	Class = Abstract Data Type
Compilation	`gcc hello.c` creates machine language code	`javac Hello.java` compiles and creates Java virtual machine language bytecode
Execution	`./a.out` loads and executes the program	`java Hello` interprets the bytecode

"Hello World!" or just… hi.	`#include <stdio.h>`
	`int main(void) {`
	`    printf("Hi\n");`
	`    return 0;`
	`}`
		`public class HelloWorld {`
		`    public static void main`
		`(String[] args) {`
		`        System.out. println("Hi");`
		`    }`
		`}`

Memory Management	No memory management, all memory handling happens via requests from user to system	new keyword allocates and initialises variables; automatically frees via garbage collection

C vs. Java (2/3)

	C	Java
Preprocessor?	Yes	No
Constants	`#define type CONST_NAME val`	`static final type constName = val;`
Variable naming conventions	structs/unions: `VariableName` functions: `function_name()` variables: `var_name`	`variableName`
Variable declaration location?	Typically at the beginning of a block; can be declared as they're used as well.	Before you use it.
Comments	block comments: `/* … */` single line comments: `// comment name` NOTE: single line comments only exist starting from the C99 standard	block comments: `/* … */` single line comments: `// comment name`
Library Management	`#include <libname.h>`	`import java.path.to.library.name;`

C vs. Java (3/3)

	C	Java
Strings	Non-native	Native
Booleans	Non-native	Native
Portability	Non-portable, system-dependent	Portable, non-system dependent
Datatypes	Specifically, granular	Larger, less "precise"

Variable Declaration	`/* Variable declaration here typically. */`
	
	`/* Rest of function. */`
		Standard to declare variables anywhere, though good practice to do at the beginning of a block.

Calling Functionality	Call by value, call by reference	Only call by value

C/Java: Operators

- arithmetic: +, -, *, /, %
- assignment: =
  - type var_name; ⇒ variable declaration
  - var_name = var_value; ⇒ initialisation
- augmented assignment: +=, -=, *=, /=, %= &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: (expr)
- order relations: <, <=, >, >=
- increment/decrement: ++, --
- member selection: ., →
  - NOTE: C's memory access policies means that these operations work slightly differently from that of Java.
  - We'll talk more about this in later lectures!
- ternary operator
  - expr ? true_ret : false_ret

Bitwise operators: AND, OR, XOR, NOT, SHIFTS

<table>
<tr><th>IN1</th><th>IN2</th><th>IN1 & IN2</th><th>IN1 | IN2</th><th>IN1 ^ IN2</th><th>!IN1</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
</table>

Left shift: 0b10**110011** << 2 -> 0b**110011**00
Right shift (logical): 0b**101100**11 >> 2 -> 0b00**101100**
Right shift (arithmetic): sign preserving

- 0b**001100**11 >> 2 -> 0b00**001100**
- 0b**101100**11 >> 2 -> 0b11**101100**

C Syntax

Main function

- To get the main function to accept arguments, use this:
  - **`int main (int argc, char *argv[])`**
- What does this mean?
  - **`argc`** will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here **`argc`** is 2:

    ```
    unix% sort myFile
    ```
  - **`argv`** is a pointer to an array containing the arguments as strings (more on pointers later).

Booleans

- What evaluates to FALSE in C?
  - 0
  - NULL (pointer, we will talk about it later)
  - False statements (e.g. 1 == 2)
- What evaluates to TRUE in C?
  - …everything else
  - Non-zero number
  - Pointer that’s not NULL
  - True statements (e.g. 1 == 1)
- `True` and `False` can only be used if when you include the stdbool.h (standard boolean) header
  - `#include <stdbool.h>`

Typed Variables in C

- The type of a variable must be declared
  - Like in Java, types can't change. E.g. `int var = 2;`

Type	Description	Example
`int`	integer values (positive, negative, 0); usually be >= 16 bits	`0, 78, -217, 0x2E`
`unsigned int`	integers > 0 (lecture 3)	`0, 6, 35102`
`float`	floating point decimal representation	`0.0, 3.14, 6.02e3`
`double`	equal or higher precision floating point	^
`char`	single character (size is the basis of a byte)	`'a', 'D', '\n'`
`long`	longer integer, >= 32 bits	`0, 78, -217, 301713123194`
`long long`	very long integer, >= 64 bits	`3170519272109251`

Consts and Enums in C

- Constant is assigned a typed value once in the declaration; value can't change during entire execution of program

```
const float  golden_ratio = 1.618;
const int    days_in_week = 7;
const double the_law      = 2.99792458e8;
```

  - You can have a constant version of any of the standard C variable types
- Enums: a group of related integer constants. E.g.,

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

Typed functions in C

- You have to declare the type of data you plan to return from a function
- Return type can be any C variable type, and is placed to the left of the function name
- You can also specify the return type as `void`
  - Just think of this as saying that no value will be returned
- Also need to declare types for values passed into a function
- Variables and functions MUST be declared before they are used

```
int number_of_people () { return 3; }
float dollars_and_cents () { return 10.33; }
```

Control Flow in C

- Very similar to Java
- Within a function, remarkably close to Java constructs (shows Java's legacy) for control flow
  - A statement can be a {} of code or just a standalone statement
- if-else
  - `if (expression)` statement
    ```
    if (x == 0) y++;
    if (x == 0) {y++;}
    if (x == 0) {y++; j = j + y;}
    if (expression) statement1 else statement2
    ```
- while
  - `while (expression)`
- for
  - `for (initialize; check; update)` statement

Memory Model Review

Before we talk about memory... arrays!

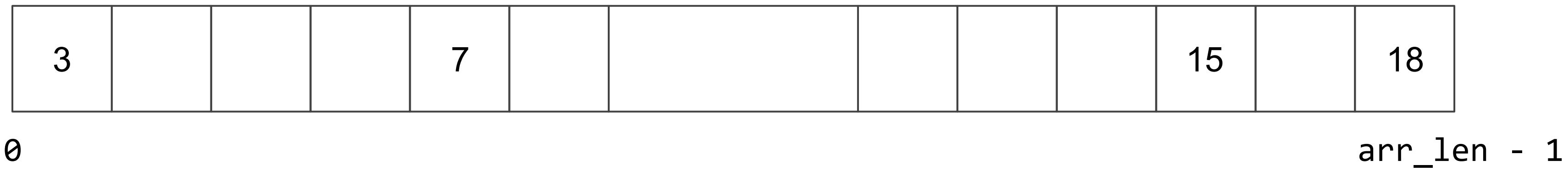
We have an array `arr`, how do we store data?

In Java, Python, etc... we *index* into them with *indices*.

How many indices do we need with an array of `arr_len`?

`arr_len - 1` since we 0-index!

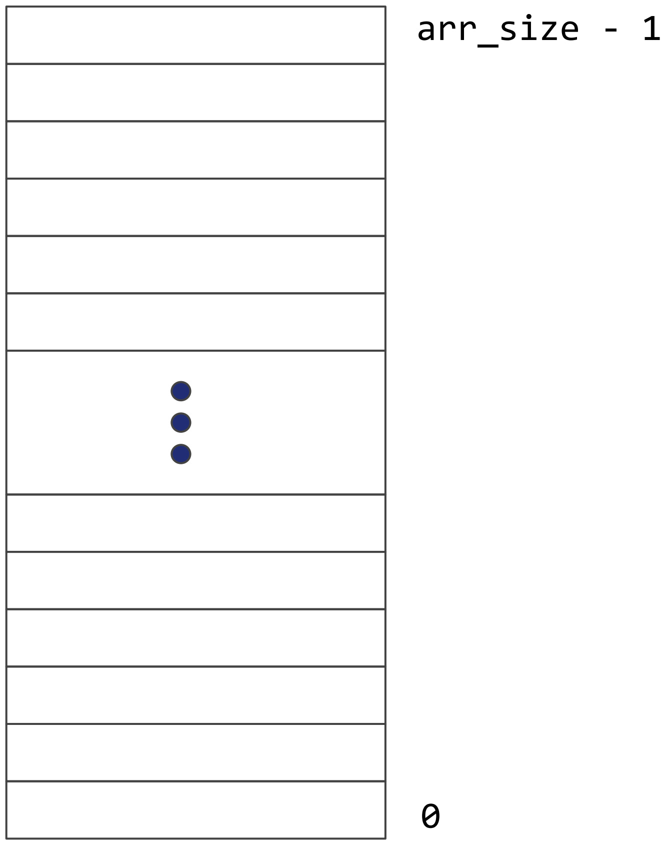
```
arr[0] = 3

arr[arr_len - 1] = 18

arr[arr_len - 3] = 15

arr[4] = 7
```



Takeaway: memory works very similarly! You can think of most versions of memory you work with conceptually to be one very long array.

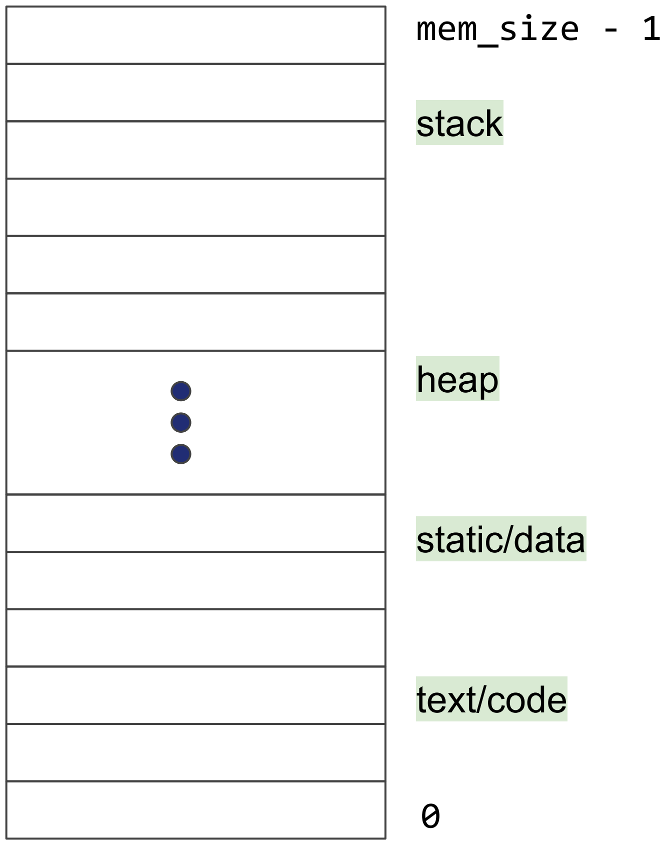
What is memory?

- Also consists of a bunch of bits!
- "Memory" usually refers to to main memory
- Can be made of ROM or RAM
  - ROM: **R**ead-**O**nly **M**emory; cannot be written to/updated by user, immutable
  - RAM: **R**andom **A**ccess **M**emory; data can be loaded/accessed in non-sequential order (access pattern can be random)
- Is usually byte-addressed
  - Implies that we need indices for every 8 bits
  - In other words, every consecutive byte will have a different address
- Addresses, as array indices, are always ≥ 0



Main Memory Structure

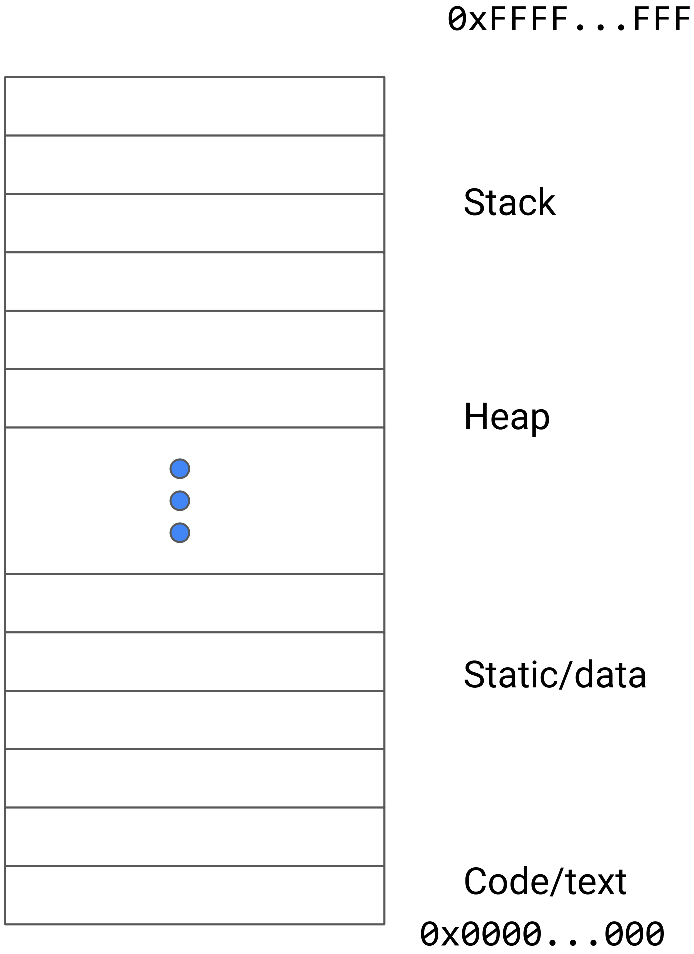
- Looks like an array, just flipped on one side!
  - Lower end of array (left side) on the bottom, higher end of array (right side) on the top
  - **Bottom-up memory model**: what 61C (and most other courses) will be using
- Memory is separated into 4 major chunks
  - Stack
  - Heap
  - Static/data
  - Text/code
  - Slide 40: Components of Main Memory



Components of Main Memory

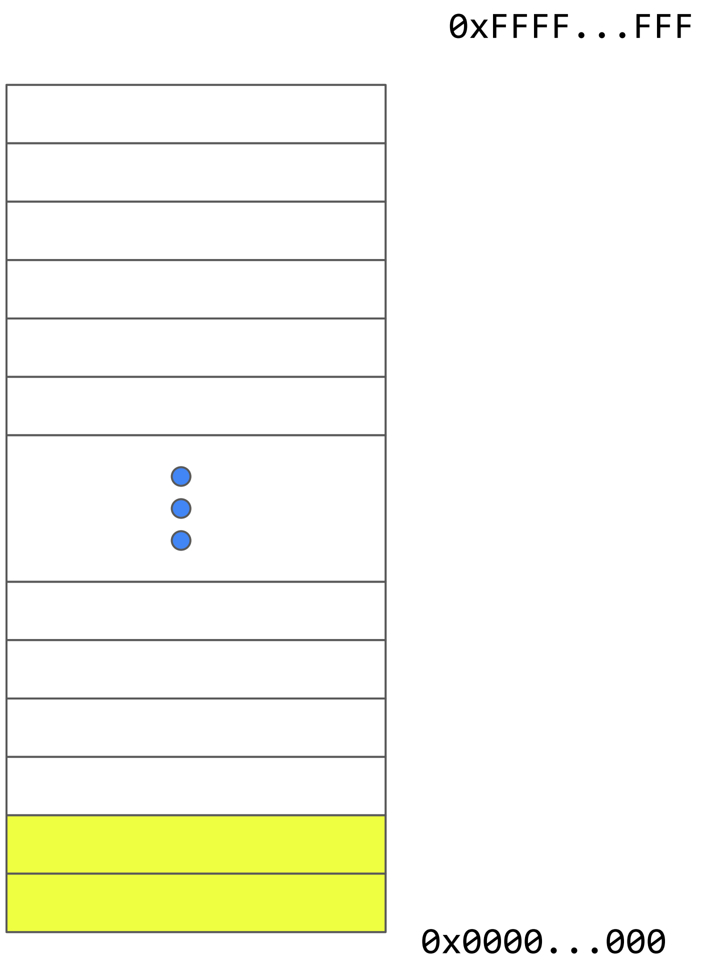
Memory Structure

- Memory is contiguous!
- Separated into 4 "chunks", top-down
  - Stack
  - Heap
  - Static/data
  - Text/Code
- Permissions
  - Read-Only (RO) ⇒ ROM
  - Read-Write Memory
  - RWX Memory
    - Read-write-execute (in practice, we don't usually want this, or write-execute)



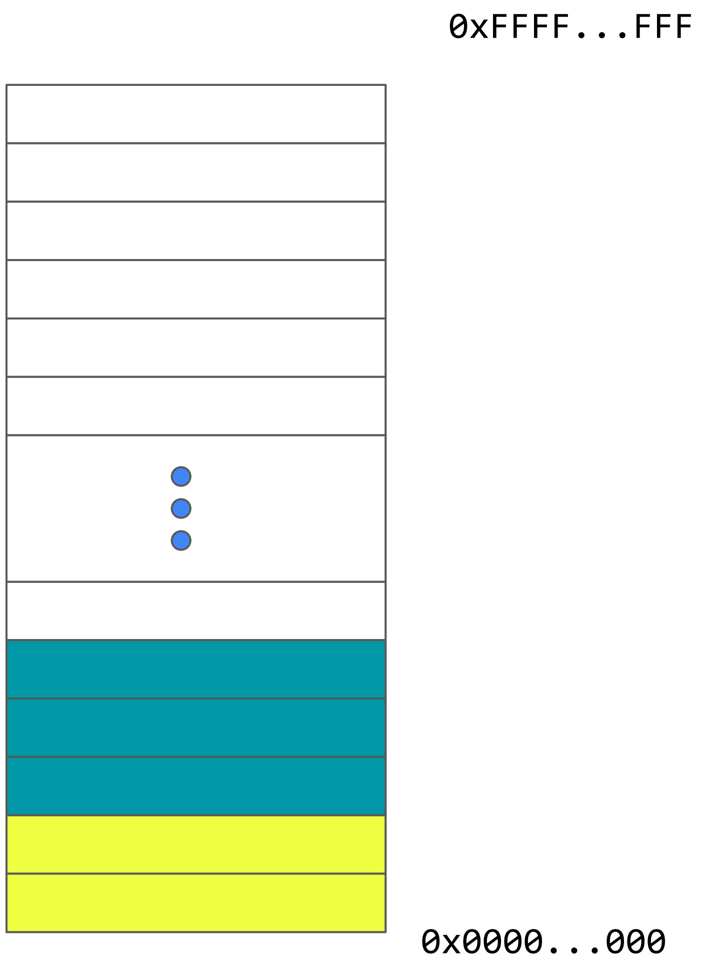
Memory Structure

- Code/text
  - The... code that you intend to execute!
  - In read-execute memory
  - Includes some constants!
    - Constants that are considered "built-in" to the code
    - x = y + 1, where does the 1 go?



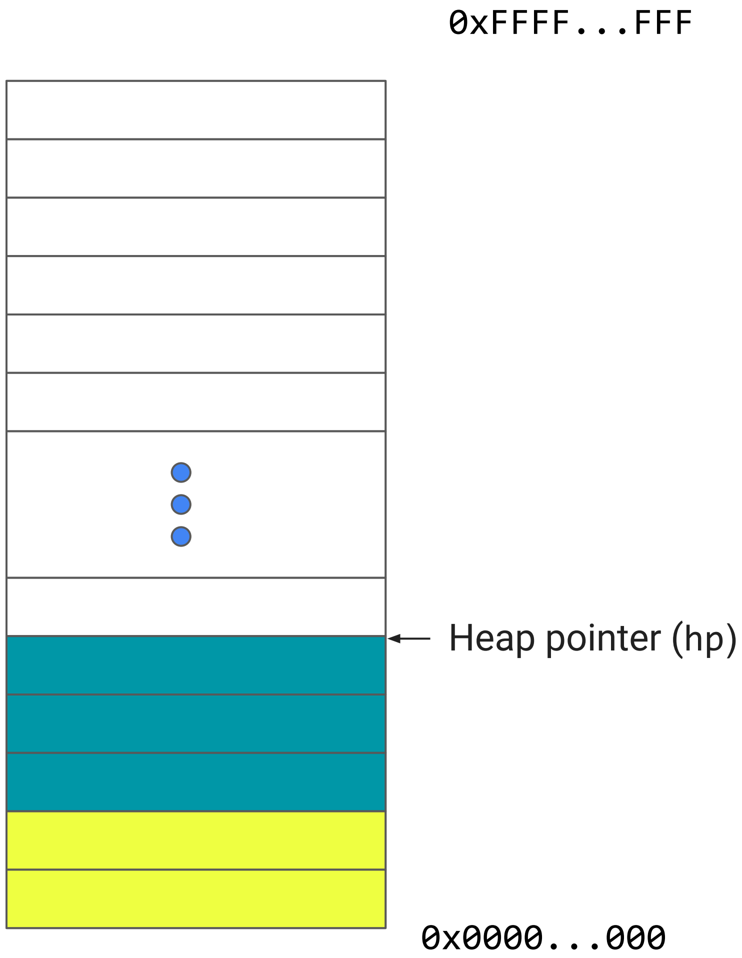
Memory Structure

- Code/text
  - The... code that you intend to execute!
  - In read-execute memory
  - Includes some constants!
    - Constants that are considered "built-in" to the code
    - x = y + 1, where does the 1 go?
- Static/data
  - Primarily constants that don't need to be changed
  - RAM
  - CAN be changed!



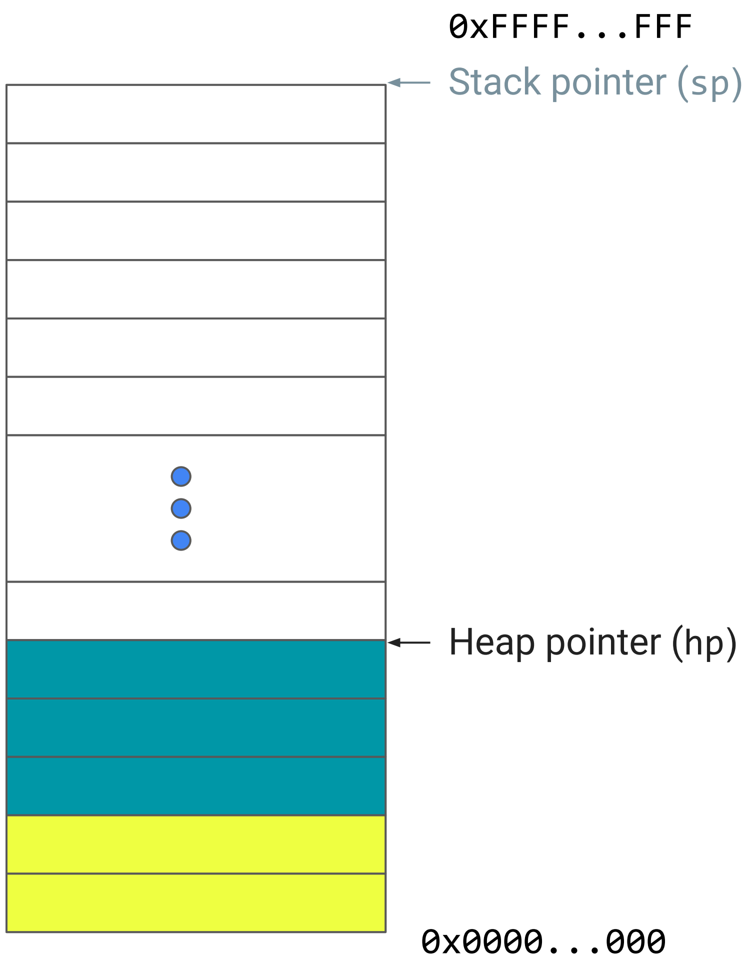
Memory Structure

- Heap
  - Memory that is dynamically-allocated
  - Read-write
  - MUST be freed!
  - Grows bottom-up in diagrams



Memory Structure

- Heap
    - Memory that is dynamically-allocated
    - Read-write
    - MUST be freed!
    - Grows bottom-up
- Stack
    - Memory that is "automatically" allocated by the system
    - Read-write
    - Will be freed by the system
    - Grows top-down



Stack Management

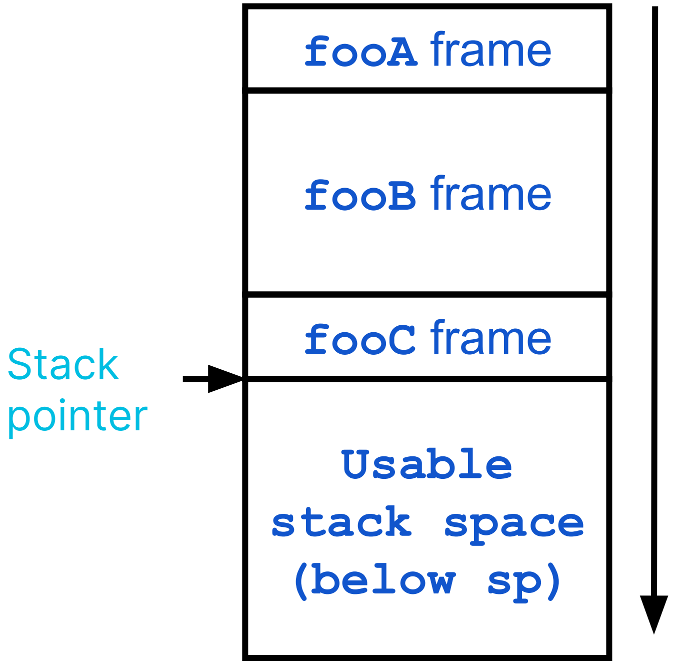
What is the stack used for?

- Anything considered "temporary"!
- This includes
  - Local (within a function) variables
  - Local constants
  - Arguments to functions
  - Local information about a function call
- All of these comprise of a "stack frame" or a "function frame"
  - Recall from 61A when we used environment diagrams
- Because of this behaviour, everything on the stack is considered to be temporary. This means that
  - anything below the stack pointer, a value we use to track the lowest valid/active address in the stack at any given moment, is considered inaccessible
  - within a function, we should only access values in our current stack frame (but we're able to access stack addresses higher than the top of our stack frame)

Stack frames, function calls

- Every time a function is called, a new “stack frame” is allocated on the stack.
- Stack frame includes:
  - Return “instruction” address (who called me?)
  - Arguments
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame.
- When function ends, stack frame is tossed off the stack; frees memory for future stack frames.
- Later, we’ll cover details for RISC-V
- **Stack grows down!!!**

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { … }
```



Summary

- 6 Great Ideas of Computer Architecture!
- Compiled and interpreted code (both translations)
  - Pros (speed) and Cons (slow edit-compile cycle)
- Compared C and Java
- Reviewed some basic C syntax (there will be more!)
- Broke down how memory is structured and used