We made it! 😊

C:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
    return 0;
}
```

Java:

```java
public class Car {
    private int miles;
    private int gals;

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        return (float) miles / gals;
    }
}
```

Assembly language:

```
get_mpg:
pushq     %rbp
movq      %rsp, %rbp
...
popq      %rbp
ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101100001111
```

Computer system:

OS:

X: Windows 8

Y: Mac

Z: Linux

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Today

- Imploring you to do your course evaluations, please!

- I’m Just a Program
  - End-to-end review

- Victory lap and high-level concepts (major points)
  - More useful for “5 years from now” than “next week’s final”

- Question time
Final Exam

- **Wednesday, June 8, 2:30pm-4:20pm**
  - Right here in Miller 301.

- **We’ve covered a lot this quarter!**
  - I know it’s a lot to review
  - But probably less time pressure than midterm

- **Will cover material from the entire course**
  - Focuses primarily on the material from the second half
  - But we’ve been building on the earlier stuff, so expect to still see concepts and material from the first half
  - Best way to get a feel for it is to look at past exams (that’s what I’m doing!)
Course Evaluations

- Really matters, and 90-100% response rate makes them much more useful than 60%
  - Have to guess what sampling bias is for “missing 40%”

- We really do take them seriously and use them to improve!
  - This is my first time teaching, so I especially need your feedback!
  - I’ve been sticking to mostly what has been done before, but we need you all to help us figure out how to make it better and more useful!

- Evaluations close this Sunday, June 5th at 11:59pm
  - I don’t know why it’s so early, but please please please do it!
  - I still can’t see them until after I submit grades. 😊
    - But you can’t see the final until after... ;)


I’m Just a Bill (I mean, Program)
How Code Becomes A Program.

Schoolhouse Rock!, 1976, “I'm Just a Bill”, written by Dave Frishberg
How Code Becomes A Program.

Source code in high-level language

1. **Source code** in high-level language
2. **Compiler**
3. **Assembly** (x86-64)
4. **Assembler**
5. **Binary Executable**
6. **Hardware**
Instruction Set Architecture

Source code
Different applications or algorithms

Program A
Program B
Your program

Compiler
Perform optimizations, generate instructions

GCC
Clang

Architecture
Instruction set

x86-64
ARMv8 (AArch64/A64)

Hardware
Different implementations

Intel Pentium 4
Intel Core 2
Intel Core i7
AMD Opteron
AMD Athlon
ARM Cortex-A53
Apple A7
Assembly Programmer’s View

- **Programmer-Visible State**
  - `%rip`: Instruction pointer
    - Address of next instruction
    - Also called “PC” (“program counter”)
  - Named registers
    - Heavily used program data
    - Together, called “register file”
  - Condition codes
    - Used for conditional branching

- **Memory**
  - Byte addressable array
  - $2^{64}$ virtual addresses (18 exabytes)
  - *Private, all to you yourself*...
Program’s View

CPU
- %rip
- Registers
- Condition Codes

Memory
- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

High addresses $2^{N-1}$

Low addresses 0

Local variables; procedure context

Variables allocated with new or malloc

Static variables (global variables in C)

Large constants (e.g., “example”)
Program’s View

- **Instructions**
  - Data movement
    - mov, movz, movl
    - push, pop
  - Arithmetic
    - add, sub, imul
  - Control flow
    - cmp, test
    - jmp, je, jgt, ...
    - call, ret

- **Operand types**
  - Literal: $8
  - Register: %rdi, %al
  - Memory: \( D(Rb,Ri,S) = D+Rb+Ri*S \)
    - lea: *not a memory access!*

---

**Memory**

- Stack
  - local variables; procedure context
- Dynamic Data (Heap)
  - variables allocated with *new* or *malloc*
- Static Data
  - *static* variables (global variables in C)
- Literals
  - Large constants (e.g., “example”)
- Instructions
  - Low addresses 0
  - High addresses \( 2^{N-1} \)
Program’s View

- **Procedures**
  - Essential abstraction
  - Recursion...

- **Stack discipline**
  - Stack frame per call
  - Local variables

- **Calling convention**
  - How to pass arguments
    - Diane’s Silk Dress Costs $89
  - How to return data
  - Return address
  - Caller-saved / callee-saved registers

**Diagram:**
- **Memory**
  - **Stack**
  - **Dynamic Data (Heap)**
  - **Static Data**
  - **Literals**
  - **Instructions**

- **Addressing:**
  - **High addresses**
  - **Low addresses**

**Notes:**
- Variables allocated with `new` or `malloc`
- Static variables (global variables in C)
- Large constants (e.g., “example”)
- Local variables; procedure context

**Procedures Notes:**
- Essential abstraction
- Recursion...

**Stack discipline Notes:**
- Stack frame per call
- Local variables

**Calling convention Notes:**
- How to pass arguments
  - Diane’s Silk Dress Costs $89
- How to return data
- Return address
- Caller-saved / callee-saved registers
Program’s View

- **Heap data**
  - Variable size
  - Variable lifetime

- **Allocator**
  - Balance *throughput* and *memory utilization*
  - Data structures to keep track of free blocks.

- **Garbage collection**
  - Must always free memory
  - Garbage collectors help by finding anything *reachable*
  - Failing to free results in *memory leaks.*
But remember... it’s all an illusion!

- **Context switches**
  - Don’t really have CPU to yourself

- **Virtual Memory**
  - Don’t really have $2^{64}$ bytes of memory all to yourself.
  - Allows for *indirection* (remap physical pages, sharing...)

- **Memory**
  - Local variables; procedure context
  - Variables allocated with `new` or `malloc`
  - *Static* variables (global variables in C)
  - Large constants (e.g., “example”)

- **CPU**
  - `%rip`
  - Registers
  - Condition Codes

- **Memory Components**
  - **Stack**
  - **Dynamic Data (Heap)**
  - **Static Data**
  - **Literals**
  - **Instructions**

- **Addressing**
  - High addresses
  - Low addresses

- **1302 Context Switches**
  - Don’t really have CPU to yourself
  - Virtual Memory
    - Don’t really have $2^{64}$ bytes of memory all to yourself.
    - Allows for *indirection* (remap physical pages, sharing...)

- **Memory Layout**
  - 0
  - High addresses
  - Low addresses

- **CPU Registers**
  - `%rip`
  - Registers
  - Condition Codes
But remember... it’s all an **illusion**!

- **Fork**
  - Creates copy of the process

- **Exec**
  - Replace with new program

- **Wait**
  - Wait for child to die (to *reap* it, and prevent *zombies*)
Virtual Memory

**Address Translation**

- Every memory access must first be converted from virtual to physical!!
- *Indirection*: just change the address mapping when switching processes!
- Luckily, TLB (and page size) makes it pretty fast.
But memory is also a lie!

- **Illusion of one flat array of bytes**
  - But *caches* invisibly make accesses (to *physical addresses*) faster!
  - Locality: temporal vs spatial

- **Caches**
  - Need to be fast, so *direct-mapped/indexed* (*sets*)
  - Need to be flexible, so *associative* (*ways*)
C: The Low Level-High Level Language

- Along the way, we learned about C data types...

- Primitive types: fixed sizes & alignments
  - Endianness: only applies to memory; is the first byte the least significant (little endian) or most (big)?

- Pointers: addresses with a type
  - Always point at the beginning of the

- Arrays
  - Contiguous chunks of memory
  - 2D arrays = still one continuous chunk
  - Nested arrays: array of pointers to other arrays
  - **Buffer Overflow**: No array bounds checks in C!!!
    - How do we protect against them?

- Structs
Nested Array Example

```
typedef int zip_dig[5];

zip_dig sea[4] =
    {{ 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

- "Row-major" ordering of all elements
- Elements in the same row are contiguous
- Guaranteed (in C)
Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`

```java
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = {uw, cmu, ucb};
```

Note: this is how Java represents multi-dimensional arrays.
Array Element Accesses

Nested array

```c
int get_sea_digit (int index, int digit) {
    return sea[index][digit];
}
```

Multi-level array

```c
int get_univ_digit (int index, int digit) {
    return univ[index][digit];
}
```

Access *looks* the same, but it isn’t:

- `Mem[sea+20*index+4*digit]`
- `Mem[Mem[univ+8*index]+4*digit]`
C: The Low Level-High Level Language

- **Structs**
  - Each *primitive element* must be aligned
  - Overall struct must be aligned to alignment of largest primitive member, size must be multiple of that as well.
  - **Fragmentation**
    - Internal fragmentation: space between members
    - External fragmentation: space after last member, *inside the struct*

```c
struct Foo {
    int a;
    double b;
    char c;
};
```

```
int a
```

```
double b
```

```
c
```

```
sizeof(Foo) ==
```
Java: A High Level Language

- Java Virtual Machine is an interpreter
  - Just need to port the JVM to your machine, then it can run your program
  - It has its own “Assembly Program’s View”

```
variable table
  0 1 2 3 4  n

operand stack

constant pool

Holds pointer ‘this’

Other arguments to method

Other local variables
```

Memory
- Call stack
- Heap
Victory Lap

A victory lap is an extra trip around the track
- By the exhausted victors (that's us) 😊

Review course goals
- Slides from Lecture 1
- What makes CSE351 special
Next 7 slides copied without change from Lecture 1

They should make much more sense now!
Welcome!

10 weeks to see the key abstractions “under the hood” to describe “what really happens” when a program runs

- How is it that “everything is 1s and 0s”?
- Where does all the data get stored and how do you find it?
- How can more than one program run at once?
- What happens to a Java or C program before the hardware can execute it?
- What is The Stack and The Heap?
- And much, much, much more...

An introduction that will:

- Profoundly change/augment your view of computers and programs
- Connect your source code down to the hardware
- Leave you impressed that computers ever work.
C/Java, assembly, and machine code

The three program fragments are equivalent

You'd rather write C! (more human-friendly)

Hardware likes bit strings!

- Everything is voltages
- The machine instructions are actually much shorter than the number of bits we would need to represent the characters in the assembly language

if (x != 0) y = (y+z)/x;

```c
if (x != 0) y = (y+z)/x;
```
The Big Theme: Abstractions and Interfaces

- Computing is about abstractions
  - (but we can’t forget reality)

- What are the abstractions that we use?

- What do you need to know about them?
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?

- How does the hardware (0s and 1s, processor executing instructions) relate to the software (C/Java programs)?
  - Become a better programmer and begin to understand the important concepts that have evolved in building ever more complex computer systems
Little Theme 1: Representation

- All digital systems represent everything as 0s and 1s
  - The 0 and 1 are really two different voltage ranges in the wires
  - Or magnetic positions on a disc, or hole depths on a DVD, or even DNA...

- “Everything” includes:
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects stored away in memory

- These encodings are stored throughout a computer system
  - In registers, caches, memories, disks, etc.

- They all need addresses
  - A way to find them
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed
Little Theme 2: Translation

- There is a **big gap** between how we think about programs and data and the 0s and 1s of computers
- Need **languages** to describe what we mean
- These languages need to be **translated** one level at a time
- We know **Java** as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We’ll encounter Java byte-codes, C language, assembly language, and machine code (for the X86 family of CPU architectures)
    - Not in that order, but will all connect by the last lecture!!!
Little Theme 3: Control Flow

- How do computers orchestrate everything they are doing?

- Within one program:
  - How do we implement if/else, loops, switches?
  - What do we have to keep track of when we call a procedure, and then another, and then another, and so on?
  - How do we know what to do upon “return”?

- Across programs and operating systems:
  - Multiple user programs
  - Operating system has to orchestrate them all
    - Each gets a share of computing cycles
    - They may need to share system resources (memory, I/O, disks)
  - Yielding and taking control of the processor
    - Voluntary or “by force”? 
Course Perspective

- **CSE351 will make you a better programmer**
  - Purpose is to show how software really works
  - Understanding the underlying system makes you more effective
    - Better debugging
    - Better basis for evaluating performance
    - How multiple activities work in concert (e.g., OS and user programs)
  - Not just a course for hardware enthusiasts!
    - What *every* CSE major needs to know (plus many more details)
    - See many *patterns* that come up over and over in computing (like caching)
  - Like other 300-level courses,
    “stuff everybody learns and uses and forgets not knowing”

- **CSE351 presents a world-view that will empower you**
  - The intellectual tools and software tools to understand the trillions+ of 1s and 0s that are “flying around” when your program runs
AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICHRenderer DOZENS OF VIDEO FRAMES EVERY SECOND BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.

I AM A GOD.
And of course don’t forget...
Memory Hierarchy

- **registers**: <1 ns
- **on-chip L1 cache (SRAM)**: 1 ns
- **off-chip L2 cache (SRAM)**: 5-10 ns
- **main memory (DRAM)**: 100 ns
- **SSD**: 150,000 ns
- **Disk**: 10,000,000 ns (10 ms)
- **remote secondary storage (distributed file systems, web servers)**: 1-150 ms
- **local secondary storage (local disks)**: 150 ns
- **remote secondary storage (distributed file systems, web servers)**: 1-2 min
- **main memory (DRAM)**: 15-30 min
- **SSD**: 31 days
- **Disk**: 66 months = 1.3 years
- **remote secondary storage (distributed file systems, web servers)**: 1 - 15 years

Larger, slower, cheaper per byte:
- Smaller, faster, costlier per byte:
Thanks for a great quarter!

- Thanks to your awesome TAs!
  - Everything that went smoothly was probably because of them!
  - Anything that didn’t was because I didn’t ask them how to do it. ;)

- Thanks for laughing occasionally at stupid jokes!

- Don’t be a stranger!
  - *(although fingers crossed, I'll graduate one of these days and you'll have to find me somewhere else)*