CS 61C: Great Ideas in Computer Architecture
Lecture 2: Introduction to C, Part I

Instructors:
Vladimir Stojanovic & John Wawrzynek

http://inst.eecs.berkeley.edu/~cs61c/
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
ENIAC (U.Penn., 1946)
First Electronic General-Purpose Computer

• Blazingly fast (multiply in 2.8ms!)
  – 10 decimal digits x 10 decimal digits
• But needed 2-3 days to setup new program, as programmed with patch cords and switches
EDSAC (Cambridge, 1949)
First General Stored-Program Computer

- Programs held as numbers in memory
- 35-bit binary 2’s complement words
Components of a Computer

Processor

Control

Datapath

Registers

Arithmetic & Logic Unit (ALU)

Memory

Enable?
Read/Write

Address

Write Data

Read Data

Program

Bytes

Data

Input

Output

Processor-Memory Interface

I/O-Memory Interfaces
Great Idea: Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
<td>Machine Interpretation</td>
</tr>
<tr>
<td>Hardware Architecture Description (e.g., block diagrams)</td>
<td>Architecture Implementation</td>
</tr>
<tr>
<td>Logic Circuit Description (Circuit Schematic Diagrams)</td>
<td></td>
</tr>
</tbody>
</table>

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

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

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

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

![Logic Circuit Diagram](image-url)
Introduction to C
“The Universal Assembly Language”

• “Some” experience is required before CS61C
  C++ or Java OK

• Class pre-req included classes teaching Java

• Python used in two labs

• C used for everything else
Language Poll!

Please raise hand for *first* one of following you can say yes to

- I have programmed in C, C++, C#, or Objective-C
- I have programmed in Java
- I have programmed in FORTRAN, Cobol, Algol-68, Ada, Pascal, or Basic
- None of the above
Intro 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.

  — Kernighan and Ritchie

• Enabled first operating system not written in assembly language: UNIX - A portable OS!
Intro to C

• Why C?: we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism

• C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!
The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.

http://www.tiobe.com
TIOBE Programming Community Index
Disclaimer

• You will not learn how to fully code in C in these lectures! You’ll still need your C reference for this course
  – K&R is a must-have
    • Check online for more sources
  – “JAVA in a Nutshell,” O’Reilly
    • Chapter 2, “How Java Differs from C”
    • http://oreilly.com/catalog/javanut/excerpt/index.html
  – Brian Harvey’s helpful transition notes
    • On CS61C class website: pages 3-19
    • http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf

• Key C concepts: Pointers, Arrays, Implications for Memory management
Compilation: Overview

- **C compilers** map C programs into architecture-specific machine code (string of 1s and 0s)
  - Unlike *Java*, which converts to architecture-independent *bytecode*
  - Unlike *Python* environments, which *interpret* the code
  - These differ mainly in exactly when your program is converted to low-level machine instructions ("levels of interpretation")
  - For C, generally a two part process of compiling `.c` files to `.o` files, then linking the `.o` files into executables;
  - Assembling is also done (but is hidden, i.e., done automatically, by default); we’ll talk about that later
C Compilation Simplified Overview
(more later in course)

*foo.c*

*bar.c*

Compiler

foo.o

bar.o

Compiler/assembler combined here

Linker

lib.o

Pre-built object file libraries

Machine code executable file

C source files (text)

Machine code object files
Compilation: Advantages

• Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)

• Reasonable compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
Compilation: Disadvantages

• Compiled files, including the executable, are architecture-specific, depending on processor type (e.g., MIPS vs. RISC-V) and the operating system (e.g., Windows vs. Linux)

• 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 tool only rebuilds changed pieces, and can do compiles in parallel (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 */
- `#define M_PI (3.14159)` /* Define constant */
- `#if/#endif` /* Conditional inclusion of text */
- Use –save-temps option to gcc to see result of preprocessing
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
# C vs. Java

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of Language</strong></td>
<td>Function Oriented</td>
<td>Object Oriented</td>
</tr>
<tr>
<td><strong>Programming Unit</strong></td>
<td>Function</td>
<td>Class = Abstract Data Type</td>
</tr>
<tr>
<td><strong>Compilation</strong></td>
<td>gcc hello.c creates machine language code</td>
<td>javac Hello.java creates Java virtual machine language bytecode</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>a.out loads and executes program</td>
<td>java Hello interprets bytecodes</td>
</tr>
<tr>
<td><strong>Hello, world</strong></td>
<td>#include&lt;stdio.h&gt; int main(void) {</td>
<td>public class HelloWorld {</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Hello\n&quot;); return 0; }</td>
<td>public static void main(String[] args) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.out.println(&quot;Hello&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>Manual <em>(malloc, free)</em></td>
<td>Automatic (garbage collection)</td>
</tr>
</tbody>
</table>

## C vs. Java

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>/* … */</td>
<td>/* … */ or // … end of line</td>
</tr>
<tr>
<td>Constants</td>
<td>#define, const</td>
<td>final</td>
</tr>
<tr>
<td>Preprocessor</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Variable declaration</td>
<td>At beginning of a block</td>
<td>Before you use it</td>
</tr>
<tr>
<td>Variable naming</td>
<td><code>sum_of_squares</code></td>
<td><code>sumOfSquares</code></td>
</tr>
<tr>
<td>conventions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accessing a library</td>
<td>#include &lt;stdio.h&gt;</td>
<td>import java.io.File;</td>
</tr>
</tbody>
</table>

Typed Variables in C

- Must declare the type of data a variable will hold
  - Types can't change

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer numbers, including negatives</td>
<td>0, 78, -1400</td>
</tr>
<tr>
<td>unsigned int</td>
<td>integer numbers (no negatives)</td>
<td>0, 46, 900</td>
</tr>
<tr>
<td>float</td>
<td>floating point decimal numbers</td>
<td>0.0, 1.618, -1.4</td>
</tr>
<tr>
<td>char</td>
<td>single text character or symbol</td>
<td>'a', 'D', '?'</td>
</tr>
<tr>
<td>double</td>
<td>greater precision/big FP number</td>
<td>10E100</td>
</tr>
<tr>
<td>long</td>
<td>larger signed integer</td>
<td>6,000,000,000</td>
</tr>
</tbody>
</table>
Integers: Python vs. Java vs. C

<table>
<thead>
<tr>
<th>Language</th>
<th>sizeof(int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>&gt;=32 bits (plain ints), infinite (long ints)</td>
</tr>
<tr>
<td>Java</td>
<td>32 bits</td>
</tr>
<tr>
<td>C</td>
<td>Depends on computer; 16 or 32 or 64</td>
</tr>
</tbody>
</table>

• **C**: `int` should be integer type that target processor works with most efficiently

• **Only guarantee**: `sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)`
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits
Consts and Enums in C

• Constant is assigned a typed value once in the declaration; value can't change during entire execution of program
  
  ```c
  const float golden_ratio = 1.618;
  const int days_in_week = 7;
  ```

• You can have a constant version of any of the standard C variable types

• Enums: a group of related integer constants. Ex:
  
  ```c
  enum cardsuit {CLUBS, DIAMONDS, HEARTS, SPADES};
  enum color {RED, GREEN, BLUE};
  ```
Clicker Test

• Clicker participation starting on Thursday
• No web-based clickers or phone apps
• Participation only is recorded, not correctness of answers
• Register on bCourses
Clicker Test

A: I have an iClicker

B: I don’t have an iClicker

C: I don’t have an iClicker

D: I don’t have an iClicker

E: I don’t have an iClicker
Compare “#define PI 3.14” and “const float pi=3.14” – which is true?

A: Constants “PI” and “pi” have same type

B: Can assign to “PI” but not “pi”

C: Code runs at same speed using “PI” or “pi”

D: “pi” takes more memory space than “PI”

E: Both behave the same in all situations
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
Administrivia

• Enrollment: We plan to admit from the waitlist and concurrent enrollment. Details TBD.

• Class website is up:
  http://www-inst.eecs.berkeley.edu/~cs61c/fa15/

• Labs start this week
  – Meet people in your lab, think about proj1 teams

• HW0 out, Please sign up for EDX
  – Due: Sunday 9/06 @ 11:59:59pm

• HW0-mini-bio posted on course website
  – Give paper copy to your TA in lab section this week
  – If you have lab on Monday, give it to your TA in section next week

• Monday is Labor Day – if you have a Monday lab, attend any lab this week (Thur, Fri)

• Get iClickers and register on bCourses! Participation points start Thursday.

• Let us know about exam conflicts by the end of this week
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
int number_of_people ()
{
    return 3;
}

float dollars_and_cents ()
{
    return 10.33;
}

int sum ( int x, int y)
{
    return x + y;
}

• 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 necessary to declare types for values passed into a function
• Variables and functions MUST be declared before they are used
Structs in C

- Structs are structured groups of variables, e.g.,

```c
typedef struct {
    int length_in_seconds;
    int year_recorded;
} Song;

Song song1;

song1.length_in_seconds = 213;
song1.year_recorded = 1994;

Song song2;

song2.length_in_seconds = 248;
song2.year_recorded = 1988;
```

Dot notation: `x.y = value`
A First C Program: Hello World

Original C:

```c
main()
{
    printf("\nHello World\n");
}
```

ANSI Standard C:

```c
#include <stdio.h>

int main(void)
{
    printf("\nHello World\n");
    return 0;
}
```
C Syntax: main

• When C program starts
  – C executable a.out is loaded into memory by operating system (OS)
  – OS sets up stack, then calls into C runtime library,
  – Runtime 1st initializes memory and other libraries,
  – then calls your procedure named main ()

• We’ll see how to retrieve command-line arguments in main() later...
A Second C Program: Compute Table of Sines

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

int main(void)
{
    int angle_degree;
    double angle_radian, pi, value;
    /* Print a header */
    printf("Compute a table of the sine function\n\n");

    /* obtain pi once for all */
    /* or just use pi = M_PI, where */
    /* M_PI is defined in math.h */
    pi = 4.0*atan(1.0);
    printf("Value of PI = %f \n\n", pi);

    angle_degree = 0;
    /* initial angle value */
    /* scan over angle */
    while (angle_degree <= 360)
    /* loop until angle_degree > 360 */
    {
        angle_radian = pi*angle_degree/180.0;
        value = sin(angle_radian);
        printf(" %3d      %f \n", angle_degree, value);
        angle_degree = angle_degree + 10;
        /* increment the loop index */
    }
    return 0;
}
```

Compute a table of the sine function

Value of PI = 3.141593

<table>
<thead>
<tr>
<th>angle</th>
<th>Sine</th>
<th>angle</th>
<th>Sine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000</td>
<td>190</td>
<td>-0.173648</td>
</tr>
<tr>
<td>10</td>
<td>0.173648</td>
<td>200</td>
<td>-0.342020</td>
</tr>
<tr>
<td>20</td>
<td>0.342020</td>
<td>210</td>
<td>-0.500000</td>
</tr>
<tr>
<td>30</td>
<td>0.500000</td>
<td>220</td>
<td>-0.642788</td>
</tr>
<tr>
<td>40</td>
<td>0.642788</td>
<td>230</td>
<td>-0.766044</td>
</tr>
<tr>
<td>50</td>
<td>0.766044</td>
<td>240</td>
<td>-0.866025</td>
</tr>
<tr>
<td>60</td>
<td>0.866025</td>
<td>250</td>
<td>-0.939693</td>
</tr>
<tr>
<td>70</td>
<td>0.939693</td>
<td>260</td>
<td>-0.984808</td>
</tr>
<tr>
<td>80</td>
<td>0.984808</td>
<td>270</td>
<td>-1.000000</td>
</tr>
<tr>
<td>90</td>
<td>1.000000</td>
<td>280</td>
<td>-0.984808</td>
</tr>
<tr>
<td>100</td>
<td>0.984808</td>
<td>290</td>
<td>-0.939693</td>
</tr>
<tr>
<td>110</td>
<td>0.939693</td>
<td>300</td>
<td>-0.866025</td>
</tr>
<tr>
<td>120</td>
<td>0.866025</td>
<td>310</td>
<td>-0.766044</td>
</tr>
<tr>
<td>130</td>
<td>0.766044</td>
<td>320</td>
<td>-0.642788</td>
</tr>
<tr>
<td>140</td>
<td>0.642788</td>
<td>330</td>
<td>-0.500000</td>
</tr>
<tr>
<td>150</td>
<td>0.500000</td>
<td>340</td>
<td>-0.342020</td>
</tr>
<tr>
<td>160</td>
<td>0.342020</td>
<td>350</td>
<td>-0.173648</td>
</tr>
<tr>
<td>170</td>
<td>0.173648</td>
<td>360</td>
<td>-0.000000</td>
</tr>
<tr>
<td>180</td>
<td>0.000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C Syntax: Variable Declarations

• Similar to Java, but with a few minor but important differences
• All variable declarations must appear before they are used (e.g., at the beginning of the block)
• A variable may be initialized in its declaration; if not, it holds garbage!
• Examples of declarations:
  – Correct: 
    
    ```c
    int a = 0, b = 10;
    ...
    ```
  
  – Incorrect: 
    ```c
    for (int i = 0; i < 10; i++)
    }
    ```


Newer C standards are more flexible about this, more later
C Syntax : Control Flow (1/2)

• Within a function, remarkably close to Java constructs (shows Java’s legacy) in terms of control flow
  – if-else
    • if (expression) statement
    • if (expression) statement1
      else statement2
  – while
    • while (expression)
      statement
    • do
      statement
      while (expression);
- for
  - for (initialize; check; update)
    statement

- switch
  - switch (expression) {
    case const1: statements
    case const2: statements
    default: statements
  }
  - break
C Syntax: True or False

• What evaluates to FALSE in C?
  – 0 (integer)
  – NULL (a special kind of pointer: more on this later)
  – No explicit Boolean type

• What evaluates to TRUE in C?
  – Anything that isn’t false is true
  – Same idea as in Python: only 0s or empty sequences are false, anything else is true!
C and Java operators nearly identical

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>==
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- equality testing: ==, !=
- subexpression grouping: ( )
- order relations: <, <=, >, >=
- increment and decrement: ++ and --
- member selection: ., ->
- conditional evaluation: ? :
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
The A8 is manufactured on a 20 nm process by TSMC. It contains 2 billion transistors. Its physical size is 89 mm^2. It has 1 GB of LPDDR3 RAM included in the package. It is dual core, and has a frequency of 1.38 GHz.

- Apple A8 APL1011 SoC + SK Hynix RAM as denoted by the markings H9CKNNN8KTMWR-NTH (we presume it is 1 GB LPDDR3 RAM, the same as in the iPhone 6 Plus)
- Qualcomm MDM9625M LTE Modem
- Skyworks 77802-23 Low Band LTE PAD
- Avago A8020 High Band PAD
- Avago A8010 Ultra High Band PA + FBARs
- SkyWorks 77803-20 Mid Band LTE PAD
- InvenSense MP67B 6-axis Gyroscope and Accelerometer Combo
More ICs on the front side of the logic board:

- Qualcomm **QFE1000** Envelope Tracking IC
- RF Micro Devices RF5159 Antenna Switch Module
- SkyWorks 77356-8 Mid Band PAD
- Bosch Sensortec BMA280
Back side of the logic board:

- SanDisk SDMFLBCB2 128 Gb (16 GB) NAND Flash
- Murata 339S0228 Wi-Fi Module
- Apple/Dialog 338S1251-AZ Power Management IC
- Broadcom BCM5976 Touchscreen Controller
- NXP LPC18B1UK ARM Cortex-M3 Microcontroller (also known as the M8 motion coprocessor)
- NXP 65V10 NFC module + Secure Element (likely contains an NXP PN544 NFC controller inside)
- Qualcomm WTR1625L RF Transceiver
More ICs await us on the back of the logic board:

- Qualcomm WFR1620 receive-only companion chip. Qualcomm states that the WFR1620 is "required for implementation of carrier aggregation with WTR1625L."

- Qualcomm PM8019 Power Management IC

- Texas Instruments 343S0694 Touch Transmitter

- AMS AS3923 NFC Booster IC designed to "improve performance of existing NFC controllers for challenging environments such as mobile and wearables" and also delivers "less restrictive antenna design requirements".

- We believe this chip is an NFC frontend to improve performance in noisy environments using 'active peer to peer bit rates [up to] 212kb/s.'

- Cirrus Logic 338S1201 Audio Codec
Break
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers
• And in Conclusion, ...
Address vs. Value

• Consider memory to be a single huge array
  – Each cell of the array has an address associated with it
  – Each cell also stores some value
  – For addresses do we use signed or unsigned numbers? Negative address?!

• Don’t confuse the address referring to a memory location with the value stored there
Pointers

• An *address* refers to a particular memory location; e.g., it points to a memory location

• *Pointer*: A variable that contains the address of a variable


**Pointer Syntax**

- `int *x;`
  - Tells compiler that variable `x` is address of an `int`
- `x = &y;`
  - Tells compiler to assign address of `y` to `x`
  - `&` called the “address operator” in this context
- `z = *x;`
  - Tells compiler to assign value at address in `x` to `z`
  - `*` called the “dereference operator” in this context
Creating and Using Pointers

• How to create a pointer:
  
  & operator: get address of a variable

  ```c
  int *p, x;
  p ??
  x ??
  x = 3;
  p ??
  x 3
  p = &x;
  p ??
  x 3
  ```

  Note the “*” gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

• How get a value pointed to?

  “*” (dereference operator): get the value that the pointer points to

  ```c
  printf("p points to %d\n", *p);
  ```
Using Pointer for Writes

• How to change a variable pointed to?
  – Use the dereference operator \( * \) on left of assignment operator =

\[
\text{p} \rightarrow \text{x} \rightarrow 3
\]

\[\text{*p} = 5;\]

\[
\text{p} \rightarrow \text{x} \rightarrow 5
\]
Pointers and Parameter Passing

• Java and C pass parameters “by value”
  – Procedure/function/method gets a copy of the parameter, so changing the copy cannot change the original

```java
void add_one (int x) {
    x = x + 1;
}
int y = 3;
add_one(y);

y remains equal to 3
```
Pointers and Parameter Passing

• How can we get a function to change the value held in a variable?

```c
void add_one (int *p) {
    *p = *p + 1;
}
int y = 3;
add_one(&y);

y is now equal to 4
```
Types of Pointers

• Pointers are used to point to any kind of data (int, char, a struct, etc.)

• Normally a pointer only points to one type (int, char, a struct, etc.).
  – void * is a type that can point to anything (generic pointer)
  – Use void * sparingly to help avoid program bugs, and security issues, and other bad things!
More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka “garbage”)
- What does the following code do?

```c
void f()
{
    int *ptr;
    *ptr = 5;
}
```
Pointers and Structures

typedef struct {
    int x;
    int y;
} Point;

Point p1;
Point p2;
Point *paddr;

/* dot notation */
int h = p1.x;
p2.y = p1.y;

/* arrow notation */
int h = paddr->x;
int h = (*paddr).x;

/* This works too */
p1 = p2;
Pointers in C

• Why use pointers?
  – If we want to pass a large struct or array, it’s easier / faster / etc. to pass a pointer than the whole thing
  – In general, pointers allow cleaner, more compact code

• So what are the drawbacks?
  – Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    • Most problematic with dynamic memory management—coming up next week
    • *Dangling references* and *memory leaks*
Why Pointers in C?

• At time C was invented (early 1970s), compilers often didn’t produce efficient code
  – Computers 25,000 times faster today, compilers better
• C designed to let programmer say what they want code to do without compiler getting in way
  – Even give compilers hints which registers to use!
• Today’s compilers produce much better code, so may not need to use pointers in application code
• Low-level system code still needs low-level access via pointers
Video: Fun with Pointers

• [https://www.youtube.com/watch?v=6pmWojisM_E](https://www.youtube.com/watch?v=6pmWojisM_E)
Agenda

• Compile vs. Interpret
• C vs. Java vs. Python
• Administrivia
• Quick Start Introduction to C
• News/Technology Break
• Pointers

• And in Conclusion, ...
And In Conclusion, ...

• All data is in memory
  – Each memory location has an address to use to refer to it and a value stored in it

• Pointer is a C version (abstraction) of a data address
  – * “follows” a pointer to its value
  – & gets the address of a value
  – Arrays and strings are implemented as variations on pointers

• C is an efficient language, but leaves safety to the programmer
  – Variables not automatically initialized
  – Use pointers with care: they are a common source of bugs in programs