CS 61C:
Great Ideas in Computer Architecture

*Introduction to C, Part I*

Instructor:
Randy H. Katz

http://inst.eecs.Berkeley.edu/~cs61c/fa13

---

**Agenda**

- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, ...
Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, ...

WSC Case Study
Server Provisioning

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSC Power Capacity</td>
<td>8.00 MW</td>
</tr>
<tr>
<td>Power Usage Effectiveness (PUE)</td>
<td>1.50</td>
</tr>
<tr>
<td>IT Equipment Power Share</td>
<td>0.67</td>
</tr>
<tr>
<td>Power/Cooling Infrastructure</td>
<td>0.33</td>
</tr>
<tr>
<td>IT Equipment Measured Peak (W)</td>
<td>145.00</td>
</tr>
<tr>
<td>Assume Average Pwr @ 0.8 Peak</td>
<td>116.00</td>
</tr>
<tr>
<td># of Servers</td>
<td>46207</td>
</tr>
<tr>
<td># of Servers per Rack</td>
<td>40.00</td>
</tr>
<tr>
<td># of Racks</td>
<td>1150</td>
</tr>
<tr>
<td>Top of Rack Switches</td>
<td>1150</td>
</tr>
<tr>
<td># of TOR Switch per L2 Switch</td>
<td>16.00</td>
</tr>
<tr>
<td># of L2 Switches</td>
<td>72</td>
</tr>
<tr>
<td># of L2 Switches per L3 Switch</td>
<td>24.00</td>
</tr>
<tr>
<td># of L3 Switches</td>
<td>3</td>
</tr>
</tbody>
</table>

Diagram:
- Internet
- L3 Switch
- L2 Switch
- TOR Switch
- Server
- Rack...
### WSC Case Study

#### Capital Expenditure (Capex)

- Facility cost and total IT cost look about the same

<table>
<thead>
<tr>
<th>Item</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facility Cost</td>
<td>$88,000,000</td>
</tr>
<tr>
<td>Total Server Cost</td>
<td>$66,700,000</td>
</tr>
<tr>
<td>Total Network Cost</td>
<td>$12,810,000</td>
</tr>
<tr>
<td>Total Cost</td>
<td>$167,510,000</td>
</tr>
</tbody>
</table>

- However, replace servers every 3 years, networking gear every 4 years, and facility every 10 years

### WSC Case Study

#### Operational Expense (Opex)

<table>
<thead>
<tr>
<th>Years Amortization</th>
<th>Monthly Cost</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 3</td>
<td>$2,000,000</td>
<td>55%</td>
</tr>
<tr>
<td>Network 4</td>
<td>$295,000</td>
<td>8%</td>
</tr>
<tr>
<td>Facility 4</td>
<td>$625,000</td>
<td>17%</td>
</tr>
<tr>
<td>Pwr&amp;Cooling 10</td>
<td>$140,000</td>
<td>4%</td>
</tr>
<tr>
<td>Other 10</td>
<td>$15,840,000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operational Expense</th>
<th>Cost/kWh</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power (8MW)</td>
<td>$0.07</td>
<td>13%</td>
</tr>
<tr>
<td>People (3)</td>
<td>$85,000</td>
<td>2%</td>
</tr>
<tr>
<td>Total Monthly</td>
<td>$3,620,000</td>
<td>100%</td>
</tr>
</tbody>
</table>

- $3.6M/46000 servers = ~$80 per month per server in revenue to break even
- ~$80/720 hours per month = $0.11 per hour
- So how does Amazon EC2 make money???
### August 2013 AWS Instances & Prices

<table>
<thead>
<tr>
<th>Instance</th>
<th>Per Hour</th>
<th>Ratio to Small</th>
<th>Compute Units</th>
<th>Virtual Cores</th>
<th>Compute Unit/Core Memory (GB)</th>
<th>Disk (GB)</th>
<th>Address (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Small</td>
<td>$0.065</td>
<td>1.0</td>
<td>1.0</td>
<td>1.00</td>
<td>1.7</td>
<td>160</td>
<td>32 bit</td>
</tr>
<tr>
<td>Standard Large</td>
<td>$0.260</td>
<td>4.0</td>
<td>4.0</td>
<td>2.00</td>
<td>7.5</td>
<td>840</td>
<td>64 bit</td>
</tr>
<tr>
<td>Standard Extra Large</td>
<td>$0.520</td>
<td>8.0</td>
<td>8.0</td>
<td>2.00</td>
<td>15.0</td>
<td>1680</td>
<td>64 bit</td>
</tr>
<tr>
<td>High-Memory Extra Large</td>
<td>$0.460</td>
<td>7.1</td>
<td>6.5</td>
<td>3.25</td>
<td>17.1</td>
<td>420</td>
<td>64 bit</td>
</tr>
<tr>
<td>High-Memory Double Extra Large</td>
<td>$0.920</td>
<td>14.2</td>
<td>13.0</td>
<td>4.00</td>
<td>34.2</td>
<td>850</td>
<td>64 bit</td>
</tr>
<tr>
<td>High-Memory Quadruple Extra Large</td>
<td>$1.840</td>
<td>28.3</td>
<td>26.0</td>
<td>8.00</td>
<td>68.4</td>
<td>1690</td>
<td>64 bit</td>
</tr>
<tr>
<td>High-CPU Medium</td>
<td>$0.165</td>
<td>2.5</td>
<td>5.0</td>
<td>2.50</td>
<td>1.7</td>
<td>350</td>
<td>32 bit</td>
</tr>
<tr>
<td>High-CPU Extra Large</td>
<td>$0.660</td>
<td>10.2</td>
<td>20.0</td>
<td>8.00</td>
<td>7.0</td>
<td>1690</td>
<td>64 bit</td>
</tr>
<tr>
<td>XXXXXXXXXXXXXXXXXX</td>
<td>$X</td>
<td>15.3</td>
<td>33.5</td>
<td>16.00</td>
<td>23.0</td>
<td>1690</td>
<td>64 bit</td>
</tr>
</tbody>
</table>

- Closest computer in WSC example is Standard Extra Large
- @$0.11/hr, Amazon EC2 can make money!
  - even if used only 50% of time

### Which statement is TRUE about Warehouse Scale Computer economics?

- The dominant operational monthly cost is **server replacement**.
- The dominant operational monthly cost is the **electric bill**.
- The dominant operational monthly cost is facility replacement.
- The dominant operational monthly cost is **operator salaries**.
Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Python vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, ...

New-School Machine Structures (It’s a bit more complicated!)

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

- 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 @ one time

- Programming Languages
Great Idea: Levels of Representation/Interpretation

- **High Level Language Program (e.g., C)**
  - Compiler
  - Assembly Language Program (e.g., MIPS)
  - Assembler
  - Machine Language Program (MIPS)

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

- **Machine Interpretation**
  - Hardware Architecture Description (e.g., block diagrams)
  - Architecture Implementation
  - Logic Circuit Description (Circuit Schematic Diagrams)

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

- **Register File**
  - **ALU**

---

Introduction to C

“The Universal Assembly Language”

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

- Class pre-req included classes teaching Java
- Java used in two labs and one project
- C used for everything else

- **THE C PROGRAMMING LANGUAGE**
  - Brian W. Kernighan
  - Dennis M. Ritchie
Language Poll!

Please raise card 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

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
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!
- C and derivatives (C++/Obj-C/C#) still one of the most popular application programming languages after >40 years!
Basic C Concepts

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Creates useable programs from C source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typed variables</td>
<td>Kind of data that a variable contains</td>
</tr>
<tr>
<td>Typed functions</td>
<td>The kind of data returned from a function</td>
</tr>
<tr>
<td>Header files (.h)</td>
<td>Declare functions and variables in a separate file</td>
</tr>
<tr>
<td>Structs</td>
<td>Groups of related values</td>
</tr>
<tr>
<td>Enums</td>
<td>Lists of predefined values</td>
</tr>
<tr>
<td>Pointers</td>
<td>Aliases to other variables</td>
</tr>
</tbody>
</table>

These concepts distinguish C from other languages you may know

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) \(\geq\) sizeof(long) \(\geq\) sizeof(int) \(\geq\) sizeof(short)
  - All could be 64 bits
<table>
<thead>
<tr>
<th></th>
<th><strong>C</strong></th>
<th><strong>Java</strong></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</td>
<td>javac Hello.java creates Java virtual machine language bytecode</td>
</tr>
<tr>
<td></td>
<td>language code</td>
<td></td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>a.out loads and executes</td>
<td>java Hello interprets bytecode</td>
</tr>
<tr>
<td></td>
<td>program</td>
<td></td>
</tr>
<tr>
<td><strong>hello, world</strong></td>
<td>#include&lt;stdio.h&gt; int main(void) { printf(&quot;Hello\n&quot;); return 0; }</td>
<td>public class HelloWorld { public static void main(String[] args) { System.out.println(&quot;Hello&quot;); } }</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>Manual (<strong>malloc, free</strong>)</td>
<td>Automatic (garbage collection)</td>
</tr>
</tbody>
</table>


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

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

Compilation: Advantages

• Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
• Fair compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
• Why C?: we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism
Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on processor type and the operating system
- 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 the development cycle

Typed Variables in C

```c
int variable1 = 2;
float variable2 = 1.618;
char variable3 = 'A';
```

- 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>
Typed Functions in C

```c
int number_of_people ()
{
    return 3;
}

float dollars_and_cents ()
{
    return 10.33;
}

char first_letter ()
{
    return 'A';
}
```

- 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 yearRecorded;
} Song;
```

```c
Song song1;
Song1.length_in_seconds = 213;
song1.yearRecorded = 1994;

Song song2;
Song2.length_in_seconds = 248;
song2.yearRecorded = 1988;
```

- Dot notation: `x.y = value`
Constats and Enums in C

- Constant is assigned a 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 used to parameterize libraries:
  ```c
  enum cardsuit {CLUBS, DIAMONDS, HEARTS, SPADES};
  ```

Which statement is TRUE regarding C and Java?

- Short, int, and long are in both languages and they have the same meaning
- As Java was derived from C, it has the same names of data types
- C programs use compilers to produce executable code but Java does not
- C has a preprocessor that allows conditional compilation, but Java does not
Agenda

• WSC Economics (Highlights)
• Compile vs. Interpret
• Scheme vs. Java vs. C
• Administrivia
• Quick Start Introduction to C
• Technology Break
• Pointers
• Arrays
• And in Conclusion, ...

Administrivia

• CS61c is relentless!
  – Next week: Lab #2, HW #2
  – Lab #2, Amazon EC2
  – HW #2 will soon be posted
• Your professor requests:
  – Need to leave early? Sit near aisles please ...
  – Want to use computer, cell phone? Sit near back of auditorium ...
Agenda

- WSC Economics (Highlights)
- Compile vs. Interpret
- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- And in Conclusion, ...

Relative Smartphone Marketshare
A First C Program: Hello World

Original C:

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

ANSI Standard C:

```c
#include <stdio.h>

int main(void)
{
    printf("
Hello World\n");
    return (0);
}
```

C Syntax: `main`

- When C program starts,
  - 1st runs job to set up computer
  - Then calls your procedure names `main()`
- To get arguments to the main function, use:
  - `int main (int argc, char *argv[])`
- What does this mean?
  - `argc` contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:
    ```c
    unix% sort myFile
    ```
  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later)
Example

- `foo hello 87`
- `argc = 3 /* number arguments */`
  - Array of pointers to strings (cover 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("\nCompute 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 */
    }
    printf("\n");
}
```
### Second C Program Sample Output

<table>
<thead>
<tr>
<th>Angle</th>
<th>Sine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>10</td>
<td>0.173648</td>
</tr>
<tr>
<td>20</td>
<td>0.342020</td>
</tr>
<tr>
<td>30</td>
<td>0.500000</td>
</tr>
<tr>
<td>40</td>
<td>0.642788</td>
</tr>
<tr>
<td>50</td>
<td>0.766044</td>
</tr>
<tr>
<td>60</td>
<td>0.866025</td>
</tr>
<tr>
<td>70</td>
<td>0.939693</td>
</tr>
<tr>
<td>80</td>
<td>0.984808</td>
</tr>
<tr>
<td>90</td>
<td>1.000000</td>
</tr>
<tr>
<td>100</td>
<td>0.984808</td>
</tr>
<tr>
<td>110</td>
<td>0.939693</td>
</tr>
<tr>
<td>120</td>
<td>0.866025</td>
</tr>
<tr>
<td>130</td>
<td>0.766044</td>
</tr>
<tr>
<td>140</td>
<td>0.642788</td>
</tr>
<tr>
<td>150</td>
<td>0.500000</td>
</tr>
<tr>
<td>160</td>
<td>0.342020</td>
</tr>
<tr>
<td>170</td>
<td>0.173648</td>
</tr>
<tr>
<td>180</td>
<td>0.000000</td>
</tr>
<tr>
<td>190</td>
<td>-0.173648</td>
</tr>
<tr>
<td>200</td>
<td>-0.342020</td>
</tr>
<tr>
<td>210</td>
<td>-0.500000</td>
</tr>
<tr>
<td>220</td>
<td>-0.642788</td>
</tr>
<tr>
<td>230</td>
<td>-0.766044</td>
</tr>
<tr>
<td>240</td>
<td>-0.866025</td>
</tr>
<tr>
<td>250</td>
<td>-0.939693</td>
</tr>
<tr>
<td>260</td>
<td>-0.984808</td>
</tr>
<tr>
<td>270</td>
<td>-1.000000</td>
</tr>
<tr>
<td>280</td>
<td>-0.984808</td>
</tr>
<tr>
<td>290</td>
<td>-0.939693</td>
</tr>
<tr>
<td>300</td>
<td>-0.866025</td>
</tr>
<tr>
<td>310</td>
<td>-0.766044</td>
</tr>
<tr>
<td>320</td>
<td>-0.642788</td>
</tr>
<tr>
<td>330</td>
<td>-0.500000</td>
</tr>
<tr>
<td>340</td>
<td>-0.342020</td>
</tr>
<tr>
<td>350</td>
<td>-0.173648</td>
</tr>
<tr>
<td>360</td>
<td>0.000000</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: 
    ```
    int a = 0, b = 10;
    ...  
    ```
  - Incorrect: 
    ```
    for (int i = 0; i < 10; i++)
    ```
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);

C Syntax: Control Flow (2/2)

- 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

• WSC Economics (Highlights)
• Compile vs. Interpret
• Python vs. Java vs. C
• Administrivia
• Quick Start Introduction to C
• Technology Break
• Pointers
• Arrays
• 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
  - Do you think they use signed or unsigned numbers? Negative address?!

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

![Diagram of memory addresses and values](image)

**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

![Diagram of pointers and memory locations](image)
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

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

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

  ```
  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 =

\[
\begin{array}{c}
* \text{p} = 5; \\
\hline
\text{p} & \text{x} \\
\hline
\end{array}
\]

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 addOne (int x) { 
    x = x + 1;
 }
int y = 3;
addOne(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 addOne (int *p) {
    *p = *p + 1;
}
int y = 3;

addOne(&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 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

```c
struct Point {
    int x;
    int y;
};

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— which you will to know by the end of the semester, but not for the projects (there will be a lab later in the semester)
    • 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
  – Compilers even ignore hints since they do it better!
How many logic and syntax errors?

```c
void main(); {
    int *p, x=5, y; // init
    y = *(p = &x) + 1;
    int z;
    flip-sign(p);
    printf("x=%d, y=%d, p=%d\n", x, y, p);
}
flip-sign(int *n){*n = -(*n)}
```

☐ 1
☐ 2
☐ 3
☐ 24

What is output after correct errors?

```c
void main(); {
    int *p, x=5, y; // init
    int z;
    y = *(p = &x) + 1;
    flip-sign(p);
    printf("x=%d, y=%d, p=%d\n", x, y, p);
}
flip-sign(int *n){*n = -(*n)}
```

☐ x=5, y=6, p=-5
☐ x=-5, y=6, p=-5
☐ x=-5, y=4, p=-5
☐ x=-5, y=-6, p=-5
Arrays (1/5)

• Declaration:
  
  \texttt{int ar[2];}
  
  declares a 2-element integer array: just a block of memory

  \texttt{int ar[] = \{795, 635\};}
  
  declares and initializes a 2-element integer array

• Accessing elements:
  
  \texttt{ar[num]}
  
  returns the num\textsuperscript{th} element

Arrays (2/5)

• C arrays are (almost) identical to pointers
  
  – \texttt{char *string} and \texttt{char string[]} are nearly identical declarations
  
  – Differ in subtle ways: incrementing, declaration of filled arrays
  
  – End of C string marked by 0 in last character

• \textit{Key Concept}: Array variable is a “pointer” to the first (0\textsuperscript{th}) element
C Strings

• String in C is just an array of characters
  \[
  \text{char string[]} = \text{"abc"};
  \]

• How do you tell how long a string is?
  – Last character is followed by a 0 byte (aka “null terminator”)
    \[
    \text{int strlen(char s[])}
    \{\text{int n = 0;}
    \text{while (s[n] != 0) n++;}
    \text{return n;}
    \}
    \]

Arrays (3/5)

• Consequences:
  – \text{ar} is an array variable, but looks like a pointer
  – \text{ar[0]} is the same as *\text{ar}
  – \text{ar[2]} is the same as *(\text{ar}+2)
  – We can use pointer arithmetic to conveniently access arrays

• Declared arrays are only allocated while the scope is valid
  \[
  \text{char *foo()} \{\text{char string[32]; ...;}
  \text{return string;}
  \}
  \]
  is incorrect \text{and very very bad}
Arrays (4/5)

- Array size $n$; want to access from 0 to $n-1$, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```c
    int i, ar[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  - Better pattern
    ```c
    int ARRAY_SIZE = 10
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++){ ... }
    ```
- SINGLE SOURCE OF TRUTH
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”

Arrays (5/5)

- Pitfall: An array in C does not know its own length, and its bounds are not checked!
  - Consequence: We can accidentally access off the end of an array
  - Consequence: We must pass the array and its size to any procedure that is going to manipulate it
- Segmentation faults and bus errors:
  - These are VERY difficult to find; be careful! (You’ll learn how to debug these in lab)
Array Summary

• Array indexing is syntactic sugar for pointers
• \texttt{a[i]} is treated as \(*\texttt{(a+i)}\)
• E.g., three equivalent ways to zero an array:
  \begin{itemize}
  \item for \((i=0; i < \text{size}; i++)\) \texttt{a[i]} = 0;
  \item for \((i=0; i < \text{size}; i++)\) \(*\texttt{(a+i)} = 0;\)
  \item for \((p = a; p < a+\text{size}; p++)\) \(*p = 0;\)
\end{itemize}

\textit{Incrementing a pointer makes it point to the next variable in memory (type of pointer says how big each variable is)}

What is TRUE about this function?

\begin{verbatim}
void foo(char *s, char *t)
{
    while (*s++)
        s++;
    while (*s++ = *t++)
        ;
}
\end{verbatim}

\begin{itemize}
\item \textbf{☐} It has syntax errors
\item \textbf{☐} No syntax errors; it changes characters in string \texttt{t} to next character in the string \texttt{s}
\item \textbf{☐} No syntax errors; it copies a string at address \texttt{t} to the string at address \texttt{s}
\item \textbf{☐} \textbf{No syntax errors; it appends the string at address \texttt{t} to the end of the string at address \texttt{s}}
\end{itemize}
Question: Which statement is FALSE regarding C and Java?

- Arrays in C are just pointers to the 0-th element
- As Java was derived from C, it has the same control flow constructs
- Like Java, in C you can check the length of an array (a.length gives no. elements in a)
- C has pointers but Java does not allow you to manipulate pointers or memory addresses of any kind

FYI—Update to ANSI C

- “C99” or “C9X” standard
  - gcc -std=c99 to compile
- References
  - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html
- Highlights
  - Declarations in for loops, like Java
  - Java-like // comments (to end of line)
  - Variable-length non-global arrays
  - <inttypes.h>: explicit integer types
  - <stdbool.h>: for boolean logic types and definitions
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
  - Array bounds not checked
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs