CS 61C: 
Great Ideas in Computer Architecture

Introduction to C, Part I

Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

Review

- Request-Level Parallelism
  - High request volume, each largely independent of other
  - Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - Map: Divide large data set into pieces for independent parallel processing
  - Reduce: Combine and process intermediate results to obtain final result
- WSC CapEx vs. OpEx
  - Economies of scale mean WSC can sell computing as a utility
  - Servers dominate cost
  - Spend more on power distribution and cooling infrastructure than on monthly electricity costs

Agenda

- Review
- Compile vs. Interpret
- Scheme vs. Java vs. C
- Administrivia
- Secret To Getting Good Job/Internship
- Quick Start Introduction to C
- Technology Break
- Pointers
- Arrays
- Summary

Big Idea #1: Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High-Level Language Programs (e.g., C)</th>
<th>Assembly Language Program (e.g., MIPS)</th>
<th>Machine Language Program (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Assembly</td>
<td>Machine</td>
</tr>
</tbody>
</table>

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Introduction to C

“the Universal Assembly Language”

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

Prior classes:
- 9/10 already know JAVA
- 1/2 already know C++
- 1/3 already know C
- 1/10 already know C#
- 1/20 have not taken 61B or equivalent

If you have no experience in these languages, then start early and ask a lot of questions in discussion!
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
    - Brian Harvey’s helpful transition notes
    - On CS61C course website:
      - 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
- Enable first operating system not written in assembly language: UNIX
  - Portable OS!

C vs. Java

<table>
<thead>
<tr>
<th>Language</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Language</td>
<td>Function Oriented</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>Programming Unit</td>
<td>Function</td>
<td>Class = Abstract Data Type</td>
</tr>
<tr>
<td>Compilation</td>
<td>gcc hello.c creates machine language code</td>
<td>javac Hello.java creates Java virtual machine language bytecode</td>
</tr>
<tr>
<td>Execution</td>
<td>a.out loads and executes program</td>
<td>java Hello interprets bytecodes</td>
</tr>
</tbody>
</table>

Integers: Scheme vs. Java vs. C

<table>
<thead>
<tr>
<th>Language</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
<td>sizeof(int)</td>
<td>infinite</td>
</tr>
<tr>
<td>Java</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Depends on computer; 16 or 32 or 64</td>
<td></td>
</tr>
</tbody>
</table>

- C: int should be integer type that target processor is most efficient working with
- Only guarantee: sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - All could be 64 bits

C vs. Java

<table>
<thead>
<tr>
<th>Language</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>const, #define</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 conventions</td>
<td>sum_of_squares</td>
<td>sumOfSquares</td>
</tr>
<tr>
<td>Accessing a Library</td>
<td>#include&lt;stdio.h&gt;</td>
<td>import java.io.File;</td>
</tr>
</tbody>
</table>

Basic C Concepts

- Compiler: Creates useable programs from C source
- Typed variables: Kind of data that a variable contains
- Typed functions: The kind of data returned from a function
- Header files (.h): Declare functions and variables in a separate file
- Structs: Groups of related values
- Enums: Lists of predefined values
- Pointers: Aliases to other variables

These concepts distinguish C from other languages you may know

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
- Enable first operating system not written in assembly language: UNIX
  - Portable OS!

Integers: Scheme vs. Java vs. C

<table>
<thead>
<tr>
<th>Language</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
<td>sizeof(int)</td>
<td>infinite</td>
</tr>
<tr>
<td>Java</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Depends on computer; 16 or 32 or 64</td>
<td></td>
</tr>
</tbody>
</table>

- C: int should be integer type that target processor is most efficient working with
- Only guarantee: sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - All could be 64 bits

C vs. Java

<table>
<thead>
<tr>
<th>Language</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>const, #define</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 conventions</td>
<td>sum_of_squares</td>
<td>sumOfSquares</td>
</tr>
<tr>
<td>Accessing a Library</td>
<td>#include&lt;stdio.h&gt;</td>
<td>import java.io.File;</td>
</tr>
</tbody>
</table>

Basic C Concepts

- Compiler: Creates useable programs from C source
- Typed variables: Kind of data that a variable contains
- Typed functions: The kind of data returned from a function
- Header files (.h): Declare functions and variables in a separate file
- Structs: Groups of related values
- Enums: Lists of predefined values
- Pointers: Aliases to other variables

These concepts distinguish C from other languages you may know

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
- Enable first operating system not written in assembly language: UNIX
  - Portable OS!

Integers: Scheme vs. Java vs. C

<table>
<thead>
<tr>
<th>Language</th>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme</td>
<td>sizeof(int)</td>
<td>infinite</td>
</tr>
<tr>
<td>Java</td>
<td>32 bits</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Depends on computer; 16 or 32 or 64</td>
<td></td>
</tr>
</tbody>
</table>

- C: int should be integer type that target processor is most efficient working with
- Only guarantee: sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - All could be 64 bits

C vs. Java

<table>
<thead>
<tr>
<th>Language</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>const, #define</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 conventions</td>
<td>sum_of_squares</td>
<td>sumOfSquares</td>
</tr>
<tr>
<td>Accessing a Library</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 most Scheme 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

- 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</td>
<td>integer numbers (no negatives)</td>
<td>0, 46, 500</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 at runtime
  - Just think of this as saying that no value will be returned
- Also necessary to define types for values passed into a function
- Variables and functions MUST be defined before they are used

Typed Functions in C

- Structs are structured groups of variables, e.g.,
  ```c
typedef struct {
    int length_in_seconds;
    int yearRecorded;
} Song;
Song song1;
song1.length_in_seconds = 213;
song1.yearRecorded = 1994;
Song song2;
song2.length_in_seconds = 248;
song2.yearRecorded = 1988;
```

Structs in C

- Variables and functions MUST be defined before they are used

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

- Compilers map C programs into architecture-specific machine code (string of 1s and 0s)
- 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

- 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

- 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

- 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</td>
<td>integer numbers (no negatives)</td>
<td>0, 46, 500</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>

- 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 at runtime
  - Just think of this as saying that no value will be returned
- Also necessary to define types for values passed into a function
- Variables and functions MUST be defined before they are used

- Structs are structured groups of variables, e.g.,
  ```c
typedef struct {
    int length_in_seconds;
    int yearRecorded;
} Song;
Song song1;
song1.length_in_seconds = 213;
song1.yearRecorded = 1994;
Song song2;
song2.length_in_seconds = 248;
song2.yearRecorded = 1988;
```
Consts and Enums in C

- Constant is assigned a value once in the declaration; value can’t change until the program is restarted
- You can have a constant version of any of the standard C variable types
- Enums: a group of related constants used to parameterize libraries

```c
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

Question: 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

Administivia

- CS61C is relentless!
  - This week: Lab #2, HW #2
  - Lab #2, Amazon EC2
  - HW #2 will soon be posted
- TA Scott Beamer guest lecture 1/31
- Due to conflicts with CS188, will start midterm at 6:40; ends at 9:40
- Wonderful to see the valuable discussion and help going on in Piazza!

Secret To Getting Good Job/Internship

- Job/Intern interviews are (now) oral exams
- Long-term memory vs. short-term/working memory
  “Long-term memory is intended for storage of information over a long time. Information from the working memory is transferred to it after a few seconds. Unlike in working memory, there is little decay.”
- Learning for recall 6-12 months later in oral exam vs. cramming to get good grades?
  - Read before lecture, think about lecture, do assignments, labs, ...
  - Cram day before exam, start project with 24 hours to go?

A First C Program: Hello World

```
#include <stdio.h>
int main(void)
{
    printf("\nHello World\n");
    return (0);
}
```
pi = 4.0*atan(1.0);

/* M_PI is defined in math.h or just use pi = M_PI, where obtain pi once for all */

/* Print a header */
printf("angle Radian | Sine
0     0.000000
10    0.173648
20    0.342020
30    0.500000
40    0.642788
50    0.766044
60    0.866025
70    0.939693
80    0.984808
90    1.000000
100   0.984808
110   0.939693
120   0.866025
130   0.766044
140   0.642788
150   0.500000
160   0.342020
170   0.173648
180   0.000000

360     -0.000000
350     -0.173648
340     -0.342020
330     -0.500000
320     -0.642788
310     -0.766044
300     -0.866025
290     -0.939693
280     -0.984808
270     -1.000000
260     -0.984808
250     -0.939693
240     -0.866025
230     -0.766044
220     -0.642788
210     -0.500000
200     -0.342020
190     -0.173648
0     0.000000

Angle Degree | Sine
0     0.000000
10    0.173648
20    0.342020
30    0.500000
40    0.642788
50    0.766044
60    0.866025
70    0.939693
80    0.984808
90    1.000000
100   0.984808
110   0.939693
120   0.866025
130   0.766044
140   0.642788
150   0.500000
160   0.342020
170   0.173648
180   0.000000

360     -0.000000
350     -0.173648
340     -0.342020
330     -0.500000
320     -0.642788
310     -0.766044
300     -0.866025
290     -0.939693
280     -0.984808
270     -1.000000
260     -0.984808
250     -0.939693
240     -0.866025
230     -0.766044
220     -0.642788
210     -0.500000
200     -0.342020
190     -0.173648
0     0.000000

Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
- if-else
  - if (expression) statement
  - else statement
- while
  - while (expression) statement
  - do
    - statement
    - while (expression);
### 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 scheme: only **#f** is false, anything else is true!

### 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

### 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
**Pointers**

- How to create a pointer:
  
  `*` operator: get address of a variable
  
  ```
  int *p, x;  
  p = &x;  
  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 to change a variable pointed to?

  ```
  *p = 5;  
  ```

  ```
  printf("p points to %d\n", *p);  
  ```

**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
  ```
  void addOne(int x) {  
    x = x + 1;  
  }  
  ```

- 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?

  ```
  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 = 4x) + 1;
  int z;
  flip-sign(p);
  printf("x=%d,y=%d,p=%d\n",x,y,p);
}
flip-sign(int *n){*n = -(*n)}
```

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)}
```

Arrays (1/5)

- Declaration:
  ```c
  int ar[2];
  ```
  declares a 2-element integer array: just a block of memory
- Initializing with values:
  ```c
  int ar[] = {795, 635};
  ```
  declares and initializes a 2-element integer array
- Accessing elements:
  ```c
  ar[num]
  ```
  returns the numth element
Arrays (2/5)

- Arrays are (almost) identical to pointers
  - char *string and char string[] are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
  - End of C string marking by 0 in last character
- Key Concept: Array variable is a “pointer” to the first \(0^{th}\) element

C Strings

- String in C is just an array of characters
  
  ```c
  char string[] = "abc";
  ```
- How do you tell how long a string is?
  
  ```c
  int strlen(char s[])
  {
    int n = 0;
    while (s[n] != 0) n++;
    return n;
  }
  ```

Arrays (3/5)

- Consequences:
  - ar is an array variable, but looks like a pointer
  - ar[0] is the same as *ar
  - We can use pointer arithmetic to conveniently access arrays
- Declared arrays are only allocated while the scope is valid
  ```c
  char *foo() {
    char string[32]; ...;
    return string;
  }
  ```
is incorrect 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
  - a[i] is treated as *(a+i)
- E.g., three equivalent ways to zero an array:
  - for (i=0; i < size; i++) a[i] = 0;
  - for (i=0; i < size; i++) *(a+i) = 0;
  - for (p=a; p < a+size; p++) *p = 0;
What is TRUE about this function?

```c
void foo(char *s, char *t)
{
    while (*s)
        s++;
    while (*s++ = *t++)
    {
        
    }
}
```

- It has syntax errors
- No syntax errors; it changes characters in string t to next character in the string s
- No syntax errors; it copies a string at address t to the string at address s
- No syntax errors; it appends the string at address t to the end of the string at address s

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