

## Lecture 3 – Introduction to the C Programming Language (pt 1)



**2010-01-25**

**Lecturer SOE Dan Garcia**

**[www.cs.berkeley.edu/~ddgarcia](http://www.cs.berkeley.edu/~ddgarcia)**

**MIPS Supercomputer ⇒  
China's next supercomputer  
(the Dawning 6000) will be built using the  
Loongson (MIPS) processor and run Linux.  
Currently, the top 500 supercomputers are  
mostly x86 chips. You'll learn MIPS in CS61C!**



# And in review...

META: We often make design decisions to make HW simple

- We represent “things” in computers as particular bit patterns: **N bits  $\Rightarrow 2^N$  things**
- These 5 integer encodings have different benefits; 1s complement and sign/mag have most problems.
- **unsigned** (C99’s `uintN_t`) :

00000 00001 ... 01111 10000 ... 11111



- **2’s complement** (C99’s `intN_t`) universal, learn!

00000 00001 ... 01111



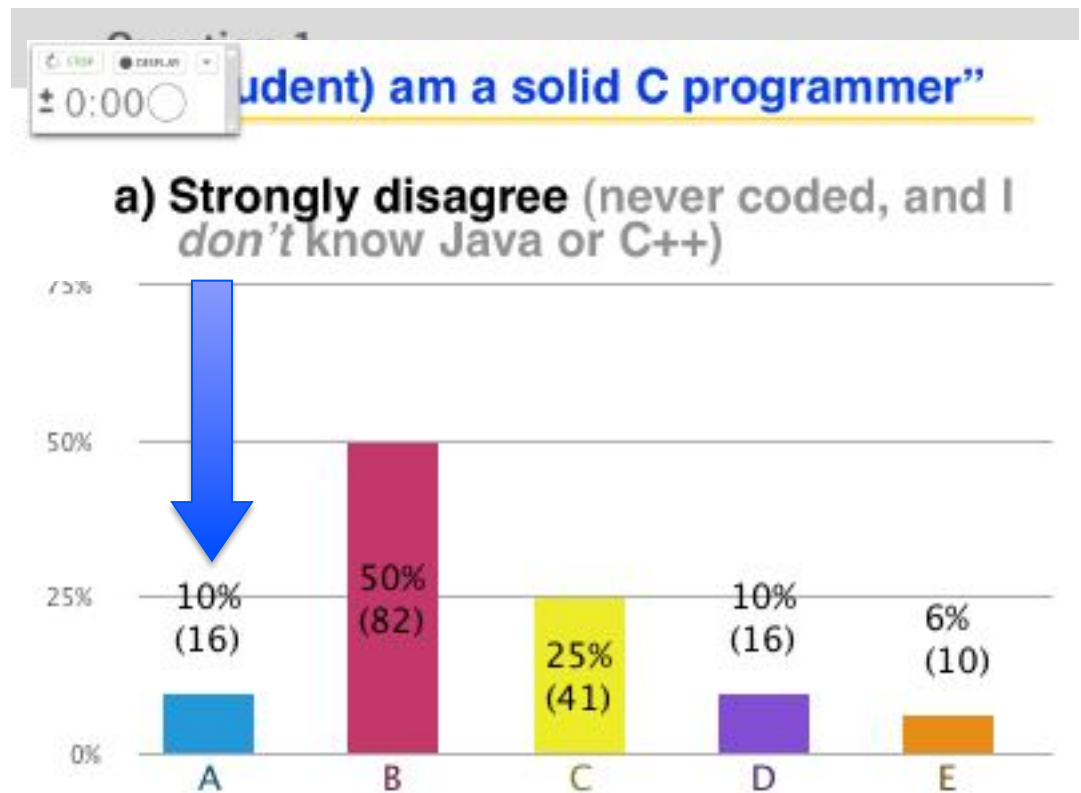
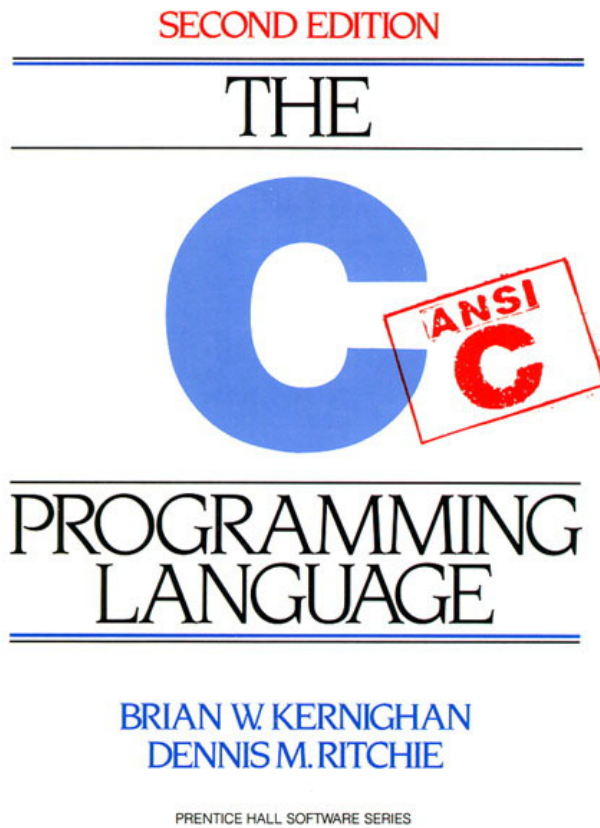
10000 ... 11110 11111

- **Overflow: numbers  $\infty$ ; computers finite, errors!**



# Introduction to C

- Officially, “some” C experience is required before CS61C
  - 2010Sp an experiment!
- What to do if you’re in that 10%?
  - Start early, ask questions as soon as they come up!



# Has there been an update to ANSI C?

---

- **Yes! It's called the "C99" or "C9x" std**

- You need "gcc -std=c99" to compile

- **References**

<http://en.wikipedia.org/wiki/C99>

[http://home.tiscalinet.ch/t\\_wolf/tw/c/c9x\\_changes.html](http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html)

- **Highlights**

- **Declarations in for loops, like Java (#15)**
- **Java-like // comments (to end of line) (#10)**
- **Variable-length non-global arrays (#33)**
- **<inttypes.h>: explicit integer types (#38)**
- **<stdbool.h> for boolean logic def's (#35)**



# Disclaimer

---

- **Important:** 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 reference**
    - Check online for more sources
  - **“JAVA in a Nutshell,” O'Reilly.**
    - Chapter 2, “How Java Differs from C”
    - <http://oreilly.com/catalog/javanut/excerpt/>
  - **Brian Harvey's course notes**
    - On CS61C class website



# Compilation : Overview

---

**C compilers** take C and convert it into an **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 **when** your program is converted to machine instructions.
- For C, generally a 2 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)



# Compilation : Advantages

---

- **Great run-time performance:** generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- **OK compilation time:** enhancements in compilation procedure (`Makefiles`) allow only modified files to be recompiled



# Compilation : Disadvantages

---

- All compiled files (including the executable) are **architecture specific**, depending on *both* the CPU type and the operating system.
- Executable must be **rebuilt** on each new system.
  - Called “**porting your code**” to a new architecture.
- The “change→compile→run [repeat]” iteration cycle is slow





# C Syntax: main

---

- To get the main function to accept arguments, use this:

```
int main (int argc, char *argv[])
```

- What does this mean?
  - `argc` will contain the number of strings on the command line (the executable counts as one, plus one for each argument). Here `argc` is 2:  

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



# C Syntax: Variable Declarations

---

- Very similar to Java, but with a few minor but important differences
- All variable declarations must go before they are used (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++)`



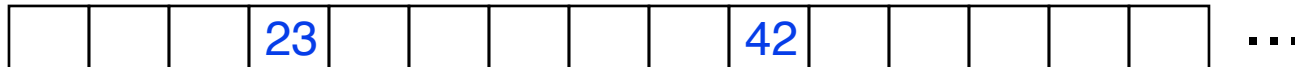
\*C99 overcomes these limitations

# 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 in that location.

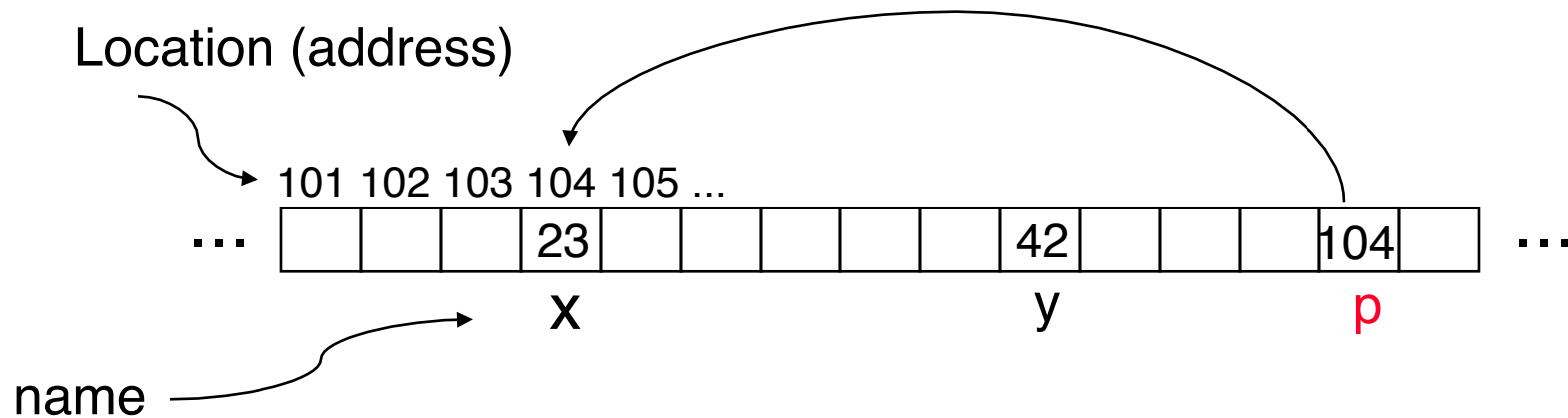
101 102 103 104 105 ...



# Pointers

---

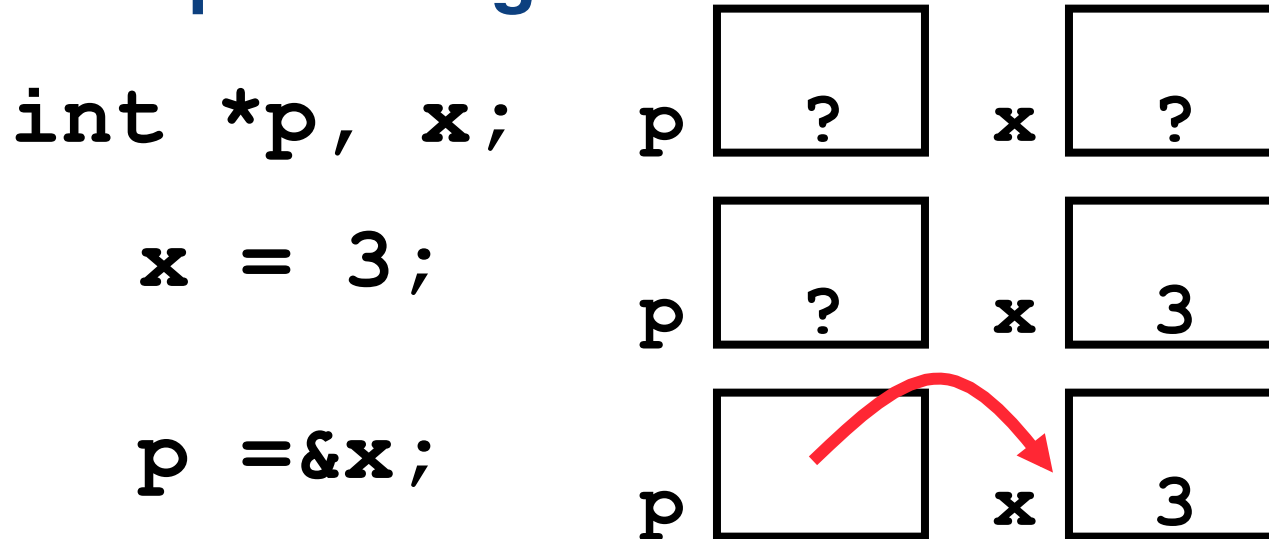
- An address refers to a particular memory location. In other words, it points to a memory location.
- **Pointer**: A variable that contains the address of a variable.



# Pointers

- How to create a pointer:

& operator: get address of a variable



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 value pointed to

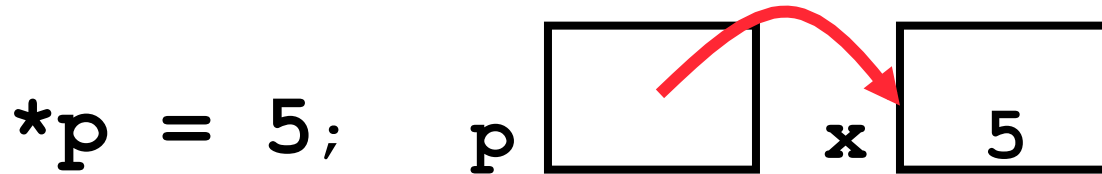
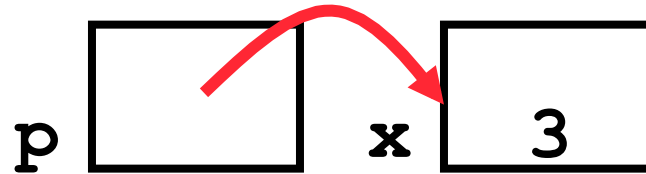
```
printf("p points to %d\n", *p);
```



# Pointers

---

- How to change a variable pointed to?
  - Use dereference `*` operator on left of `=`



# 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;  
}
```

```
int y = 3;  
addOne (y) ;
```

**y is still = 3**



# Pointers and Parameter Passing

---

- How to get a function to change a value?

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

```
int y = 3;
```

```
addOne (&y) ;
```

**y is now = 4**





# Pointers

---

- Pointers are used to point to **any** data type (`int`, `char`, a `struct`, etc.).
- Normally a pointer can only point 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 a lot of other bad things!



# Peer Instruction Question



```
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) }
```

**How many syntax+logic errors in this C99 code?**

- | <u>#Errors</u> |
|----------------|
| a) 1           |
| b) 2           |
| c) 3           |
| d) 4           |
| e) 5           |



# Peer Instruction Answer



```
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); }
```

How many syntax+logic errors in this C99 code?

I get **5**...

(signed ptr print is logical err)

#Errors

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5**



## And in conclusion...

---

- All declarations go at the beginning of each function except if you use C99.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A **pointer** is a C version of the address.
  - \* “follows” a pointer to its value
  - & gets the address of a value



# C vs. Java™ Overview (1/2)

---

## Java

- Object-oriented (OOP)
- “Methods”
- Class libraries of data structures
- Automatic memory management

## C

- No built-in object abstraction. Data separate from methods.
- “Functions”
- C libraries are lower-level
- **Manual** memory management
- **Pointers**



# C vs. Java™ Overview (2/2)

---

## Java

- **High** memory overhead from class libraries
- **Relatively Slow**
- Arrays initialize to **zero**
- **Syntax:**

```
/* comment */  
// comment  
System.out.print
```

## C

- **Low** memory overhead
- **Relatively Fast**
- Arrays initialize to **garbage**
- **Syntax: \***

```
/* comment */  
// comment  
printf
```

\* You need newer C compilers to allow Java style comments, or just use C99



# C Syntax: True or False?

---

- **What evaluates to FALSE in C?**
  - 0 (integer)
  - NULL (pointer: more on this later)
  - no such thing as a Boolean\*
- **What evaluates to TRUE in C?**
  - **everything else...**
  - (same idea as in scheme: only #f is false, everything else is true!)



\*Boolean types provided by C99's `stdbool.h`

# C syntax : flow control

---

- Within a function, remarkably **close to Java** constructs in methods (shows its legacy) in terms of flow control
  - `if-else`
  - `switch`
  - `while` and `for`
  - `do-while`

