Princeton cracks down! ⇒
Previously, nearly half the grades given out were {A-,A,A+}...not unusual; other Ivys 44-55%. New cap is 35%. EECS policy is 17% (Lower div) and 23% (upper), though not strict.
Review (1): Overview

- We represent “things” in computers as particular bit patterns: \( N \text{ bits} \Rightarrow 2^N \)

- Decimal for human calculations, binary for computers, hex to write binary more easily

- 1’s complement - mostly abandoned

- 2’s complement universal in computing: cannot avoid, so learn

- Overflow: numbers \( \infty \); computers finite, errors!
Review(2): The way to remember #s

• What is $2^{27}$? How many bits addresses (i.e., what’s $\text{ceil } \log_2 19 = \lg$ of) 19 PiB?

• Answer! $2^{XY}$ means…

<table>
<thead>
<tr>
<th>$X=0$</th>
<th>$Y=0$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X=1$</td>
<td>$Y=1$</td>
<td>2</td>
</tr>
<tr>
<td>$X=2$</td>
<td>$Y=2$</td>
<td>4</td>
</tr>
<tr>
<td>$X=3$</td>
<td>$Y=3$</td>
<td>8</td>
</tr>
<tr>
<td>$X=4$</td>
<td>$Y=4$</td>
<td>16</td>
</tr>
<tr>
<td>$X=5$</td>
<td>$Y=5$</td>
<td>32</td>
</tr>
<tr>
<td>$X=6$</td>
<td>$Y=6$</td>
<td>64</td>
</tr>
<tr>
<td>$X=7$</td>
<td>$Y=7$</td>
<td>128</td>
</tr>
<tr>
<td>$X=8$</td>
<td>$Y=8$</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>$Y=9$</td>
<td>512</td>
</tr>
</tbody>
</table>
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”.


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.
- Generally a 2 part process of **compiling** .c files to .o files, then **linking** the .o files into executables.
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 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)

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>High</strong> memory overhead from class libraries</td>
<td>• <strong>Low</strong> memory overhead</td>
</tr>
<tr>
<td>• Relatively Slow</td>
<td>• Relatively Fast</td>
</tr>
<tr>
<td>• Arrays initialize to zero</td>
<td>• Arrays initialize to garbage</td>
</tr>
<tr>
<td>• Syntax:</td>
<td>• Syntax:</td>
</tr>
</tbody>
</table>

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

```c
/* comment */
printf
```
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.

• Examples of declarations:
  • correct: 
    
    ```c
    int a = 0, b = 10;
    ...
    ```
  • incorrect: for (int i = 0; i < 10; i++)
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!)
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
C Syntax: `main`

- To get the main function to accept arguments, use this:
  ```c
  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).
    - Example: `unix% sort myFile`
  - `argv` is a pointer to an array containing the arguments as strings (more on pointers later).
Administrivia: You have a question?

• Do **not** email Dan (& expect response)
  • Hundreds of emails in inbox
  • Email doesn’t scale to classes with 200+ students!

• Tips on getting an answer to your question:
  • Ask a classmate
  • Ask Dan after or before lecture
  • The newsgroup, ucb.class.cs61c
    - Read it: Has your Q been answered already?
    - If not, ask it and check back
  • Ask TA in section, lab or OH
  • Ask Dan in OH
  • Ask Dan in lecture (if relevant to lecture)
  • Send your TA email
  • Send one of the two Head TAs email
  • Send Dan email
Administrivia : Near term

- Upcoming lectures
  - C pointers and arrays in detail

- HW
  - HW0 due in discussion tomorrow
  - HW1 due this Wed @ 23:59 PST
  - HW2 due next Wed @ 23:59 PST

- Reading
  - K&R Chapters 1-5 (lots, get started now!)
  - First quiz due Friday

- Get cardkeys from CS main office Soda Hall 3rd floor if you need/want them
  - Soda locks doors @ 6:30pm & on weekends
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.

• Don’t confuse the address referring to a memory location with the value stored in that location.
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

  \textbf{int} \ *p, x; \quad p \quad ^{?} \quad x \quad ^{?} \\
  x = 3; \quad p \quad ^{?} \quad x \quad 3 \\
  p = \&x; \quad p \quad x \quad 3

• How get a value pointed to?
  * "dereference operator": get value pointed to

  \texttt{printf(\"p points to %d\n\",*p);}
Pointers

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

```
p x 3
```

```
*p = 5;
```

```
p x 5
```
Pointers and Parameter Passing

• Java and C pass a parameter “by value”

  • procedure/function gets a copy of the parameter, so changing the copy cannot change the original

    ```c
    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?

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

int y = 3;

addOne (&y);

• y is now = 4
```
Pointers

• 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!
Peer Instruction Question

```c
void main(); {
    int *p, x=5, y; // init
    y = *(p = &x) + 10;
    int z;
    flip-sign(p);
    printf("x=%d,y=%d,p=%d\n",x,y,p);
}

flip-sign(int *n){*n = -(*n)}
```

How many errors?
void main() {  
  int *p, x=5, y; // init  
  y = *(p = &x) + 10;  
  int z;  
  flip-sign(p);  
  printf("x=%d,y=%d,p=%d\n",x,y,*p);  
}  
flip-sign(int *n){*n = -(*n);}

And in conclusion...

• All declarations go at the beginning of each function.

• 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