Clarification from last lecture

• Subtraction on UNSIGNED integers can be done using traditional methods

\[
\begin{array}{c|c|c|c|c}
1 & 1 & 0 & 1 \\
\hline
0 & 1 & 0 & 1 \\
\hline
\end{array}
\]

\( (+13) - (+3) = 10 \]

\[
\begin{array}{c|c|c|c|c}
0 & 0 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 \\
\hline
\end{array}
\]

• Subtraction on SIGNED integers is done as follows:
  - \( A - B \) becomes \( A + (-B) \), which we can do

Peer Instruction

Which of the following are TRUE?

I: One of the advantages of 2’s complement is it provides symmetry in the number of positive and negative #’s

II: In 2’s compliment, each number has only one unique representation (for a given number of bits).

\[
\begin{array}{c|c|c}
| & I & II |
\hline
a) & F & F \\
b) & F & T \\
c) & T & F \\
d) & T & T \\
e) & No idea \\
\hline
\end{array}
\]

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://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”
  - 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:
  ```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). Here argc is 2:
    ```c
    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: {
      ```c
      int a = 0, b = 10;
      ```
  - incorrect: for (int i = 0; i < 10; i++)
- *C99 overcomes these limitations

Actual C Code

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  int n = 5;
  for (i = 0; i < n; i = i + 1) {
    printf("hello, world\\n");
  }
  return 0;
}
```
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.

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

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

  * “dereference operator”: get value pointed to

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

• How get a value pointed to?

  ```c
  *p = 5;
  printf("what is x? %d\n", x);
  x is 5
  ```

Common Pitfall

• In variable declarations, does * associate with the type, or the variable?

  ```c
  int *x;     // This is legal
  int* y;     // Is this legal?
  int* z, q;  // Uh oh!
  ```

  Yes, all legal code. What is the type of q? q is an int, not a pointer!
  * associates with variables in declarations, so we suggest you use the first style!

Administrivia!

• New class mailing list!
  • We will go over how to sign up in lab!

• Labs start today. Be there!
  • You should have cardkey access to the room, if not, send Noah an email!

• HW1 online now! (Dun dun dunnnnn.. =)

• C Help session!
  • Wednesday June 23rd, 7-9pm, 306 Soda

• Check back of yesterday’s handout!
  • Illustrated study guide for number rep!
More Administrivia!

• Final will be on Thursday, August 12th
  • Time TBA
• Let’s see what you guys think about midterm times
  A) I’d like the midterm to be during class time, 9:30am-12:30pm.
  B) I’d like the midterm to be during class time, 8:00am-11am.
  C) I’d like the midterm to be in the evening during a day we have lecture (M-Th)
  D) I’d like the midterm to be on Friday

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
    ```c
    void addOne (int x) {
        x = x + 1;
    }
    int y = 3;
    addOne(y);
    printf("What is y? %d\n", y);
    y is still = 3
    ```

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.).
    ```c
    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!
    ```

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:
    ```c
    /* comment */
    // comment
    System.out.print
    ```
  • You need newer C compilers to allow Java style comments, or just use C99

C
  • Low memory overhead
  • Relatively Fast
  • Arrays initialize to garbage
  • Syntax:
    ```c
    * comment */
    // comment
    printf
    ```
**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

---

**Peer Instruction Question**

```c
void main(int argc, char *argv[])
{
    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?

<table>
<thead>
<tr>
<th>#Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) 1</td>
</tr>
<tr>
<td>b) 2</td>
</tr>
<tr>
<td>c) 3</td>
</tr>
<tr>
<td>d) 4</td>
</tr>
<tr>
<td>e) 5</td>
</tr>
</tbody>
</table>

---

**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