CS 61C: Great Ideas in Computer Architecture

Introduction to C, Pointers

Instructor: Justin Hsia
Review of Last Lecture

• Six Great Ideas in Computer Architecture
• Number Representation
  – Bits can represent anything!
  – \( n \) bits can represent up to \( 2^n \) things
  – Unsigned, one’s complement, two’s complement
  – Sign extension: same number using more bits
• From Discussion: Powers of 2
Question: Take the 4-bit number \( x = 0b1010 \).

Which of the following numbers does \( x \) NOT represent in the schemes discussed last lecture?
- unsigned
- sign and magnitude
- one’s complement
- two’s complement

-4 (X)
-6
10
-2
Great Idea #1: Levels of Representation/Interpretation

- **Higher-Level Language Program (e.g. C)**
- **Assembler**
- **Machine Language Program (MIPS)**
- **Machine Interpretation**
- **Hardware Architecture Description (e.g. block diagrams)**
- **Architecture Implementation**
- **Logic Circuit Description (Circuit Schematic Diagrams)**

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

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

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```
Agenda

• Basic C Concepts
  – Compilation
  – Variable Types

• Administrivia

• C Syntax and Control Flow

• Pointers
  – Address vs. Value
Experience with C

- Official prerequisites: “Some” C experience is required before CS61C
  - C++ or JAVA okay

- Average CS61C class:
  \[ \approx \frac{9}{10} \text{ already know JAVA} \]
  \[ \approx \frac{1}{2} \text{ already know C++} \]
  \[ \approx \frac{1}{3} \text{ already know C} \]
  \[ \approx \frac{1}{10} \text{ already know C#} \]
  \[ \approx \frac{1}{20} \text{ have not taken 61B or equivalent} \]

- 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, so make use of C references!
  – K&R is a must-have
  – “JAVA in a Nutshell” by David Flanagan
  – Brian Harvey’s notes (on course website)
    • [http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf](http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf)
  – Other online sources
Introducing 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

• With C we can write programs that allow us to exploit underlying features of the architecture
# C Concepts

These concepts distinguish C from other programming languages that you may know:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Creates useable programs from C source code</td>
</tr>
<tr>
<td>Typed variables</td>
<td>Must declare the kind of data the variable will contain</td>
</tr>
<tr>
<td>Typed functions</td>
<td>Must declare the kind of data returned from the function</td>
</tr>
<tr>
<td>Header files (.h)</td>
<td>Allows you to declare functions and variables in separate files</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>
Compilation Overview

• C is a compiled language

• C compilers map C programs into architecture-specific machine code (string of 0s and 1s)
  – Unlike Java, which converts to architecture-independent bytecode (run by JVM)
  – Unlike most Scheme environments, which directly interpret the code
  – These differ mainly in exactly when your program is mapped to low-level machine instructions
Compilation Advantages

• **Excellent run-time performance:** Generally much faster than Scheme or Java for comparable code because it optimizes for the given architecture

• **Fair compilation time:** Enhancements in compilation procedure (*Makefiles*) allow us to recompile only the modified files
Compilation Disadvantages

• Compiled files, including the executable, are architecture-specific (CPU type and OS)
  — 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
Typed Variables in C

You must declare the type of data a variable will hold.
Declaration must come before or simultaneously with assignment.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed integer</td>
<td>5, -12, 0</td>
</tr>
<tr>
<td>short int</td>
<td>smaller signed integer</td>
<td></td>
</tr>
<tr>
<td>long int</td>
<td>larger signed integer</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>single text character or symbol</td>
<td>'a', 'D', '?'</td>
</tr>
<tr>
<td>float</td>
<td>floating point non-integer numbers</td>
<td>0.0, 1.618, -1.4</td>
</tr>
<tr>
<td>double</td>
<td>greater precision FP number</td>
<td></td>
</tr>
</tbody>
</table>

• Integer sizes are machine dependent!
  – Common size is 4 bytes, but can’t ever assume this

• Can add “unsigned” before int or char

6/19/2012
sizeof()

• If integer sizes are machine dependent, how do we tell?

• Use \texttt{sizeof}() function
  – Returns size in bytes of variable or data type name
    Examples: \texttt{int x; sizeof(x); sizeof(int)};

• Acts differently with arrays and structs, which we will cover later
  – Arrays: returns size of whole array
  – Structs: returns size of one instance of struct (sum of sizes of all struct variables + padding)
Characters

• Encode characters as numbers, same as everything!
• How many characters do we want to encode?
  – a-z, A-Z, 0-9, and what else?
• ASCII standard defines 128 different characters and their numeric encodings (http://www.asciitable.com)
  – char representing the character ‘a’ contains the value 97
  – char c = ‘a’; or char c = 97; are both valid
• A char takes up 1 byte of space
  – 7 bits is enough to store a char ($2^7 = 128$), but we add a bit to round up to 1 byte since computers usually deal with multiples of bytes
Typecasting in C (1/2)

• C is a “weakly” typed language
  – You can explicitly typecast from any type to any other:
    ```c
    int i = -1;
    if(i < 0)
      printf("This will print\n");
    if((unsigned int)i < 0)
      printf("This will not print\n");
    ```

• This is possible because everything is stored as bits!
  – Can be seen as changing the “programmer’s perspective” of the variable
Typecasting in C (2/2)

• C is a “weakly” typed language
  – You can explicitly typecast from any type to any other:
    ```c
    int i = -1;
    if(i < 0)
      printf("This will print\n");
    if((unsigned int)i < 0)
      printf("This will not print\n");
    ```

• Can typecast anything, even if it doesn’t make sense:
  ```c
  struct node n; /* structs in a few slides */
  int i = (int) n;
  ```
  – More freedom, but easier to shoot yourself in the foot
/// function prototypes
int my_func(int, int);
void sayHello();

/// function definitions
int my_func(int x, int y)
{
    sayHello();
    return x*y;
}
void sayHello()
{
    printf("Hello\n");
}
Structs in C

• Way of defining compound data types
• A structured group of variables, possibly including other structs

typedef struct {
    int lengthInSeconds;
    int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds = 213;
song1.yearRecorded = 1994;

Song song2;

song2.lengthInSeconds = 248;
song2.yearRecorded = 1988;
<table>
<thead>
<tr>
<th></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>Creates machine-dependant code</td>
<td>Creates machine-independent bytecode</td>
</tr>
<tr>
<td>Execution</td>
<td><em>Loads and executes</em> program</td>
<td>JVM <em>interprets</em> bytecode</td>
</tr>
<tr>
<td><strong>Hello World</strong></td>
<td>#include&lt;stdio.h&gt; int main(void) {</td>
<td>public class HelloWorld {</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Hello\n&quot;); return 0; }</td>
<td>public static void main(String[] args) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System.out.println(&quot;Hello&quot;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Memory management</td>
<td>Manual <em>(malloc, free)</em></td>
<td>Automatic <em>(garbage collection)</em></td>
</tr>
</tbody>
</table>

Agenda

• Basic C Concepts
  – Compilation
  – Variable Types

• Administrivia

• C Syntax and Control Flow

• Pointers
  – Address vs. Value
Administrivia

• Lab 1 is today
  — Get class account and register
  — Find partner for labs
• HW0 is due tonight
  — Need lab account to submit
• Start HW1 when you feel comfortable
• Don’t forget about office hours!
  — Justin’s moved to Sat 10-12 in 200 SDH this week
Agenda

• Basic C Concepts
  – Compilation
  – Variable Types
• Administrivia
• C Syntax and Control Flow
• Pointers
  – Address vs. Value
C and Java operators nearly identical

For precedence/order of execution, see Table 2-1 on p. 53 of K&R

• arithmetic: +, −, *, /, %
• assignment: =
• augmented assignment: +=, −=, *=, /=, %=, &=, |=, ^=, <<=, >>=
• bitwise logic: ~, &, |, ^
• bitwise shifts: <<, >>
• boolean logic: !, &&, | |
• equality testing: ==, !=
• subexpression grouping: ( )
• order relations: <, <=, >, >=
• increment and decrement: ++ and --
• member selection: . , ->
• conditional evaluation: ? :
Generic C Program Layout

```c
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables and structs */

int main(int argc, char *argv[]) {
    /* the innards */
}

/* define other functions */

Programs start at main()  
main() must return int
```

Handled by Preprocessor

Dumps other files here (.h and .o)

Macro substitutions

Remember rules of scope!  
(internal vs. external)
Sample C Code

#include <stdio.h>
#define REPEAT 5

int main(int argc, char *argv[]) {
    int i;
    int n = 5;
    for (i = 0; i < REPEAT; i = i + 1) {
        printf("hello, world\n");
    }
    return 0;
}
C Syntax: `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 name counts as one, plus one for each argument).
  – `argv` is a `pointer` to an array containing the arguments as strings (more on pointers later)
main Example

% foo hello 87

• **Here argc = 3 and the array argv contains pointers to the following strings:**
  argv[0] = "foo"
  argv[1] = "hello"
  argv[2] = "87"

• **We will cover pointers and strings later**
C Syntax: Variable Declarations

• *All* variable declarations must appear before they are used (e.g. at the beginning of a block of code)
• A variable may be initialized in its declaration; if not, it holds garbage!
• Variables of the same type may be declared on the same line
• Examples of declarations:
  – Correct:  
    ```
    int x;
    int a, b=10, c;
    ```
  – Incorrect:  
    ```
    for(int i=0; i<10; i++);
    short x=1, y=1.0;
    ```
C Syntax: True or False

- No explicit Boolean type in C (as opposed to Java)
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of pointer: more on this later)
- 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!
C Syntax: Control Flow

• Should be similar to what you’ve seen before
  – if-else
    • if (expression) statement
    • if (expression) statement1
      else statement2
  – while
    • while (expression)
      statement
    • do
      statement
      while (expression);
C Syntax: Control Flow

• Should be similar to what you’ve seen before
  – for
    • for (initialize; check; update) statement
  – switch
    • switch (expression){
      case const1: statements
      case const2: statements
      default: statements
    }
    • break
switch and break

• Case statement (switch) requires proper placement of break to work properly
  – “Fall through” effect: will execute all cases until a break is found

```java
switch(ch){
    case "+": ... /* does + and - */
    case "-": ... break;
    case "*": ... break;
    default: ... 
}
```

  – In certain cases, can take advantage of this!
Has there been an update to ANSI C?

• Yes! It’s called the “C99” or “C9x” std
  – Use option “gcc -std=c99” at compilation

• References
  http://home.tiscali.net/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> for explicit integer types (#38)
  – <stdbool.h> for boolean logic definitions (#35)
Get to Know Your Staff

• Category: Cal
Agenda

• Basic C Concepts
  – Compilation
  – Variable Types

• Administrivia

• C Syntax and Control Flow

• Pointers
  – Address vs. Value
Address vs. Value

• Consider memory to be a single huge array
  – Each cell/entry of the array has an address
  – Each cell also stores some value

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

• **A pointer** is a variable that contains an address
  
  – An address refers to a particular memory location, usually also associated with a variable name
  
  – Name comes from the fact that you can say that it **points** to a memory location
Pointer Syntax

• `int *x;`
  – Declare variable `x` the address of an `int`

• `x = &y;`
  – Assigns address of `y` to `x`
  – `&` called the “address operator” in this context

• `z = *x;`
  – Assigns the value at address in `x` to `z`
  – `*` called the “dereference operator” in this context
Pointer Example

```c
int *p, x, y;
x = 3; y = 4;
p = &x;
*p = 5;
y = *p;
```

- **Declare**
  - `int *p, x, y;`

- **Assign vals**
  - `x = 3; y = 4;`

- **Assign ref**
  - `p = &x;`

- **Dereference (1)**
  - `*p = 5;`

- **Dereference (2)**
  - `y = *p;`
Pointer Types (1/2)

• Pointers are used to point to one kind of data (int, char, a struct, etc.)
  — Pointers to pointers? Oh yes! (e.g. int **pp)

• Exception is the type void *, which can point to anything (generic pointer)
  — Use sparingly to help avoid program bugs, and security issues, and other bad things!
Pointer Types (2/2)

• Functions can return pointers
  char *foo(char data) {
    return &data;
  }

• Placement of * does not matter to compiler, but might to you
  – int* x is the same as int *x
  – int *x,y,z; is the same as int* x,y,z; but NOT the same as int *x,*y,*z;
Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - Procedure/function/method gets a copy of the parameter, so changing the copy does not change the original

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

**Code:**
```java
int y = 3;
addOne(y); ← y remains equal to 3
```
Pointers and Parameter Passing

• How do we get a function to change a value?
  – Pass “by reference”: function accepts a pointer and then modifies value by dereferencing it

Function: 

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

Code: 

```c
int y = 3;
addOne(&y); ← y is now equal to 4
```
Pointers in C

• Why use pointers?
  – When passing a large struct or array, it’s easier/faster to pass a pointer than a copy of the whole thing
  – In general, pointers allow cleaner, more compact code

• Careful: Pointers are likely the single largest source of bugs in C
  – Most problematic with dynamic memory management, which we will cover later
  – *Dangling references* and *memory leaks*
Pointer Bugs

- Local variables in C are not initialized, they may contain anything (a.k.a. “garbage”)
- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!

```c
void f() {
    int p, x;
    x = *p;
}
```

```c
void f2() {
    int *ptr;
    *ptr = 5;
}
```
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) }
**Answer:** How many errors (syntax and logic) in this C code?

```c
#include <stdio.h>     \(1\)
void main(); {  \(2\)
  int *p, x=5, y; // init
  y = *(p = &x) + 1;
  int z;
  flip-sign(p); \(3\)
  printf("x=%d, y=%d, p=%d
", x, y, *p);
} \(4\)
flip-sign(int *n){ \(5\)
  *n = -(*n);}
```

- 2
- 3
- 4
- 5+
Question: What is output from the corrected code below?

```c
#include <stdio.h>
int 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);}
```

Options:
- 5, 6, -5
- -5, 6, -5
- -5, 4, -5
- -5, -6, -5

Correct answer: -5, 6, -5
Summary

• C is an efficient (compiled) language, but leaves safety to the programmer
  – Weak type safety, variables not auto-initialized
  – Use pointers with care: common source of bugs!

• Pointer is a C version (abstraction) of a data address
  – Each memory location has an address and a value stored in it
  – * "follows" a pointer to its value
  – & gets the address of a value

• C functions “pass by value”