CS 61c: Great Ideas in Computer Architecture

Introduction to C, Pointers

Instructor: Alan Christopher

June 24, 2014

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, biased, 1's complement, 2's complement
  - Overflow
  - Sign extension: same number using more bits

Question: Consider the 4-bit numeral $x = 0b1010$
Which of the following numbers does x not represent, using *any* of the schemes discussed yesterday in lecture (unsigned, sign and magnitude, 1's complement, bias, 2's complement)?

(blue) -4

(green) -6

(purple) 10

(yellow) -2

Question: Consider the 4-bit numeral $x = 0b1010$
Which of the following numbers does x not represent, using *any* of the schemes discussed yesterday in lecture (unsigned, sign and magnitude, 1's complement, bias, 2's complement)?

(blue) -4
(green) -6
(purple) 10
(yellow) -2

Great Idea #1: Levels of Representation/Interpretation

High Level Language Program (e.g.C)

↓ *Compiler*

Assembly Language Program (e.g. MIPS)

↓ *Assembler*

Machine Language Program

↓ *Machine Interpretation*

Hardware Architecture Description (e.g. block diagrams)

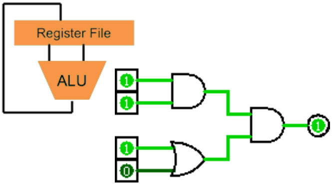
↓ *Architecture Implementation*

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)
```

```
1000 1100 0100 1000 0000 0000 0000 0000
1000 1100 0100 1001 0000 0000 0000 0100
1010 1100 0100 1001 0000 0000 0000 0000
1010 1100 0100 1000 0000 0000 0000 0100
```



Outline

Basic C

General Introduction
Compilation
Types

Administrivia

C Syntax and Control Flow

Syntax
Control

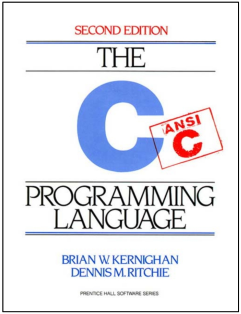
Pointers

Address vs. Value
Pointer Syntax
Pointer Applications

Summary

Experience with C

- Offical prerequisites: "Some" C experience is required before cs61c
  - C++ or Java is fine



- Average cs61c class:
  - ≈ 9/10 already know Java
  - ≈ 1/2 already know C++
  - ≈ 1/3 already know C
  - ≈ 1/10 already know C#
  - ≈ 1/20 have not take 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 the full body of C in these lectures, so make use of C references!
  - K&R is **THE** resource
  - Brian Harvey's notes (on course website)
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
  - Other online resources
    - http://www.stackoverflow.com/
    - http://www.google.com/ (Not a joke, you'd be amazed how effective searching on error messages can be.)

Intoducing 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 langauges* — 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:	
Compiler	Creates useable programs from C source code
Typed variables	Must declare the kind of data the variable will contain
Typed functions	Must declare the kind of data returned from the function
Header files (`.h`)	Allows you to declare functions in files separate from their definitions
Structs	Groups of related values
Enums	Lists of predefined values
Pointers	Aliases to other variables

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 direclty interpret the code
  - These differ mainly in exactly when your program is mapped to low-level machine instructions
- Note, we're discussing compiled-ness as though that were a language feature.
  - This is technically a decision of how to implement a language
  - But most implementors of languages follow the same choice

Compilation Advantages

- **Excellent run-time performance:** Generally much faster than interpretted languages like Scheme or Java, because code can be optimized for a given architecture and avoids costly interpretation at runtime.
- **Fair compilation time:** Modern compilation technologies (e.g. `gmake`) usually only have to recompile modified files.
  - Negligible for small projects
  - Indispensible for large projects

Compilation Disadvantages

- Compiled files, including the executable, are architecture-specific (CPU type and OS)
  - Executable must be rebuilt on each new system
  - Known as "porting your code" to a new architecture
- "Modify Code $\rightarrow$ Compile $\rightarrow$ Run [repeat]" iteration cycle can be relatively slow.

Typed Variables in C

```c
int x;
float y = 3.14159;
char z = 'A';
```

- The type of a variable must be declared before the variable is used
- Can combine declaration and initial assignment for brevity

Type	Description	Examples
`int`	signed integer	5, -12, 0
`short int` (`short`)	smaller signed integer	
`long int` (`long`)	larger signed integer	5L, -12L, 0L
`char`	single text character or symbol	'a', 'D', '?'
`float`	floating point rational numbers	0.0f, 3.14159f
`double`	greater precision FP number	0.0, 3.14159

- Integer sizes are implementation dependant!
  - 4 or 8 bytes are common choices, but cannot be assumed
- Can specify "`unsigned`" before `int`s or `char`s

sizeof()

- Need a way of peeking at the size of an integer on a given machine to guarantee portability
- Use `sizeof()`
  - Returns the size in bytes of a variable or datatype.
    E.g.: `int x; sizeof(x); sizeof(int);`
- Some small subtleties with arrays and structs
  - Arrays: returns the size of the size of the whole array
  - Structs: returns the size of a single struct (sum of sizes of all struct variables PLUS padding)

Characters

- Encode characters as numbers, just like everything else
- ASCII standard defines 128 different characacters 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
  - Unusual, most C types have implementation specific sizes
  - 7 bits is enough to represent all the characters we need, but we add a bit, since modern computers are almost always byte addressed

Typecasting I

- C is a weakly typed language
  - You can explicity typecast from any type to any other:

```
int i = -1;
if (i < 0)
    printf("This will print\n");
if ((unsigned) i < 0)
    printf("This will not print\n");
```

- Remember, everything is just a bitstring
  - All we're doing is changing the way those bits are interpretted

Typecasting II

- C is a weakly typed language
  - You can explicity typecast from any type to any other:

```
int i = -1;
if (i < 0)
    printf("This will print\n");
if ((unsigned) i < 0)
    printf("This will not print\n");
```

- C will let you make typecasts, even when it doesn't make sense:

```
/* structs in a few slides,
 * basically stripped-down Objects. */
struct node n;
int i = (int) n;
```

  - Occasionally useful, but an easy source of errors for new C programmers
  - Usually it's best to avoid casting at all, if possible

Functions in C

```
/* function prototypes */
int my_func(int, int);
void say_hello();
```

```
/* function definitions */
int my_func(int x, int y) {
    say_hello();
    return x * y;
}
void say_hello() {
    printf("Hello\n");
}
```

- Must declare the data type returned by a function
- Can return any C type or `void` if no return value is generated
  - Placed to the left of the function name
- Function arguments must have their types defined as well
- Functions must be declared before they are referenced. Commonly declared in a "prototype" at the top of a file, or in a header (`.h`) file.

Structs

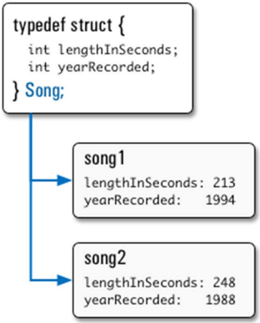
- Way of defining compound data types
- A structured group of variables, possibly including other structs

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

Song song1;

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

Song song2 = {248, 1988};
```



C vs. Java

<table>
<tr><th></th><th>C</th><th>Java</th></tr>
<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 dependent code</td><td>Creates machine-independent bytecode</td></tr>
<tr><td>Execution</td><td>Loads and executes program</td><td>JVM interprets bytecode</td></tr>
<tr><td>Hello World</td><td><pre>#include <stdio.h>
int main(void) {
    printf("Hello\n");
    return 0;
}</pre></td><td><pre>public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello");
  }
}</pre></td></tr>
<tr><td>Memory management</td><td>Manual — <code>malloc()</code>, <code>free()</code></td><td>Automatic (garbage collection)</td></tr>
</table>

From http://www.cs.princeton.edu/introcs/faq/c2java.html

Outline

- Basic C
  - General Introduction
  - Compilation
  - Types
- Administrivia
- C Syntax and Control Flow
  - Syntax
  - Control
- Pointers
  - Address vs. Value
  - Pointer Syntax
  - Pointer Applications
- Summary

Administrivia

- Lab 1 is today
  - Get class account and GitHub repository set up
  - Find a partner for labs
- Don't forget about office hours

Outline

- Basic C
  - General Introduction
  - Compilation
  - Types
- Administrivia
- C Syntax and Control Flow
  - Syntax
  - Control
- Pointers
  - Address vs. Value
  - Pointer Syntax
  - Pointer Applications
- Summary

C Operators

C operators and Java operators are nearly identical. For precedence/order of execution, see Table 2-1 on p. 53 of K&R. When in doubt, use parentheses!

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

Generic C Program Layout

```
/* Import declarations from other files. */
#include <system_files>
#include "local files"

/* Replaces macro_name with macro_expr
 * everywhere else in the program. */
#define macro_name macro_expr

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

/* Programs start at main(), which must return an int. */
int main(int argc, char *argv[]) {
    /* Do stuff. */
}

/* define other functions */
...
```

Sample Code

```
#include <stdio.h>
/* Magic numbers are bad. This is better. */
#define REPEAT 5

int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < REPEAT; i += 1) {
        printf("Hello world!\n");
    }
    return 0;
}
```

The syntax of `main()`

- To get arguments to `main()`, 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 an array containing *pointers* to the arguments as strings (more on pointers later)

main() Example

```
$ foo hello 87
```

- Here argc is 3, and the array argv contains the pointers to the following strings

```
argv[0]: "foo"
argv[1]: "hello"
argv[2]: "87"
```

- Pointers and strings will be covered later

Variable Declarations

- Variables must always be declared before they are used
- A variable may be initialized in its declaration
  - Unitialized variables hold garbage
- Variables of the same type may be declared on the same line
- Examples:

Correct:
```
int x;
int a, b = 10, c;
```
Incorrect:
```
int x = y = z;
```

Booleans

- No explicit boolean type (unlike Java)
- What evaluates to false in C?
  - 0 (integer value)
  - `NULL` (a special pointer)
- What evaluates to true in C?
  - Everything that isn't false!
  - Similar idea to scheme with #f and python with `None` (although the list of false values in python is a bit longer than that).

Control Flow I

- Should be similar to what you've seen before
  - if-else
    - `if (predicate) statement`
    - `if (predicate) statement1`
      `else statement2`
  - while
    - `while(predicate) statement`
    - ```
      do
          statement
      while (predicate);
      ```
      ```

Control Flow II

- Should be similar to what you've seen before
  - for
    - ```
      for (initialize; check; update)
          statement
      ```
  - switch
    - ```
      switch (expression) {
      case const1:   statements1
      case const2:   statements2
      ...
      case constn:   statementsn
      default:   statements_default
      }
      ```
  - break

switch and break

- Switch statements require proper use of break to work properly
  - "Fall through" effect: will execute all cases until a break is found

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

  - In some cases this is convenient, but it's a common source of bugs, so be careful

C99

- The K&R describes the ANSI C standard. C99 adds some new, convenient features to the language.
  - To compile with C99 use the "-std=c99" or "-std=gnu99" flag with gcc
- References
  - http://en.wikipedia.org/wiki/C99
  - http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html
- Highlights
  - Declarations in for loops, like Java
  - Java-like //-style comments
  - Variable length non-global arrays
  - <inttypes.h> for explicit integer types
  - <stdbool.h> for boolean logic definitions

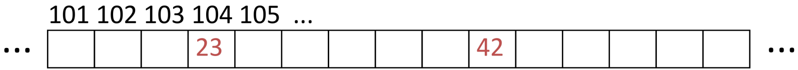
Technology Break

Outline

- Basic C
  - General Introduction
  - Compilation
  - Types
- Administrivia
- C Syntax and Control Flow
  - Syntax
  - Control
- Pointers
  - Address vs. Value
  - Pointer Syntax
  - Pointer Applications
- Summary

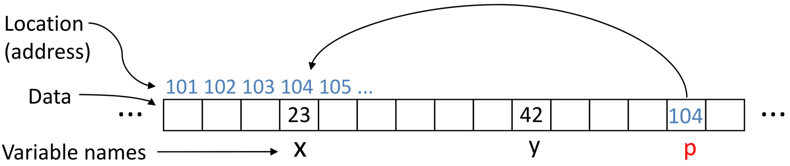
Address vs. Value

- 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 a pointer *points* to a value in memory



Pointer Syntax

- `int *x;`
  - Declare variable `x` to be the address of an `int`
- `x = &y;`
  - Assigns the *address of* `y` to `x`
  - & called the "address operator" in this context
- `z = *x;`
  - Assigns the *value at* `x` to `z`
  - * called the "dereference operator" in this context

An Example

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

p: ? x: ? y: ? Declare

An Example

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

p: ? x: ? y: ? Declare

```
x=3; y=4;
```

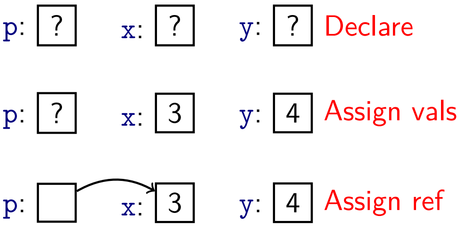
p: ? x: 3 y: 4 Assign vals

An Example

```c
int *p, x, y;

x=3; y=4;

p=&x;
```



An Example

Code	p	x	y	Step
`int *p, x, y;`	?	?	?	Declare
`x=3; y=4;`	?	3	4	Assign vals
`p=&x;`		3	4	Assign ref
`*p = 5;`		5	4	Dereference (1)

An Example

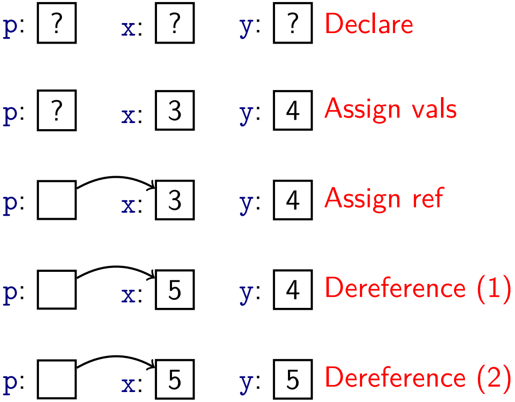
```c
int *p, x, y;

x=3; y=4;

p=&x;

*p = 5;

y = *p;
```



Pointer Types I

- Pointers are used to point to one kind of data (`int`, `char`, a `struct`, etc.)
  - Pointers to pointers? Why not! (e.g. `int **h`)
- Exception is the type `void *`, which can point to anything
  - Use sparingly to avoid bugs and unreadable code.

Pointer Types II

- Functions can return pointers

```
char *foo(char data) {
    return &data;
}
```

- Placement of * does not matter to the compiler, but it might matter to you
  - `int* x;` is equivalent to `int *x;`
  - `int* x,y,z;` is 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:

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

Code:

```
int y = 3;
addOne(y); /* Does nothing. */
```

Pointers and Parameter Passing

- how do we get a function to change a value?
  - Pass "by reference": Instead of passing in the value, pass in a *pointer* to the value. The function can then modify the value by dereferencing the pointer it was given.

Function:

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

Code:

```
int y = 3;
addOne(&y); /* y == 4 */
```

Pointers in C

- Why use pointers?
  - When passing a large struct or array, it's much faster to pass a pointer than to copy the whole thing
  - Pointers allow for cleaner, more compact code
- Pointers are likely the single largest source of bugs in C
  - Most problematic with dynamic memory management, which we'll cover later
  - *Dangling references* and *memory leaks*

Pointer Bugs

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

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

    /* BAAAD! */
    x = *p;
}
```

```c
void f2() {
    int *p;

    /* BAAAD! */
    *p = 5;
}
```

Question: How many errors (syntactical and logical) exist in this C99 code?

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

(blue) 2
(green) 3
(purple) 4
(yellow) 5+

Question: How many errors (syntactical and logical) exit in this C99 code?

```
#include <stdio.h>
void flip-sign(int *n) { *n = -(*n); }
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);
}
```

(blue) 2
(green) 3
(purple) 4
(yellow) 5+

Question: What is the output from the corrected code below?

```
#include <stdio.h>
void flip_sign(int *n) { *n = -(*n); }
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);
}
```

(blue) 5,6,-5
(green) -5,6,-5
(purp) -5,4,-5
(yell) -5,-6,-5

Question: What is the output from the corrected code below?

```
#include <stdio.h>
void flip_sign(int *n) { *n = -(*n); }
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);
}
```

(blue) 5,6,-5
(green) -5,6,-5
(purp) -5,4,-5
(yell) -5,-6,-5

Outline

- Basic C
  - General Introduction
  - Compilation
  - Types
- Administrivia
- C Syntax and Control Flow
  - Syntax
  - Control
- Pointers
  - Address vs. Value
  - Pointer Syntax
  - Pointer Applications
- Summary

Summary

- C is an efficient (compiled) language, but leaves safety to the programmer
  - Weak type safety, variables not auto-initialized
  - Pointers are awesome, but dangerous; be careful
- Pointers in C are really addresses
  - Each memory location has an address and has a value stored in it
  - * "follows" a pointer to its value
  - & gets the address of a value
- C functions are "pass by value"