CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Introduction to C (Part I)

Instructor:
Michael Greenbaum

http://inst.eecs.Berkeley.edu/~cs61c/su11

Two's Complement Review

- Suppose we had 5 bits. What integers can be represented in two's complement?
  - a) -32 to +31
  - b) -31 to +32
  - c) 0 to +31
  - d) -16 to +15
  - e) -15 to +15
  - f) -15 to +16

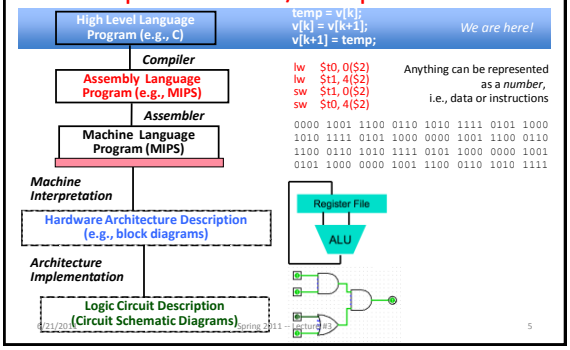
Two's Complement Review

- Suppose we had 5 bits. What integers can be represented in two's complement?
  - a) -32 to +31
  - b) -31 to +32
  - c) 0 to +31
  - **d) -16 to +15**
  - e) -15 to +15
  - f) -15 to +16

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Break
- Pointers
- Summary

Levels of Representation/Interpretation



Agenda

- C Language Overview
- Administrivia
- C Syntax and Control Flow
- Break
- Pointers
- Summary

Introduction to C

- Official prerequisites: "Some" C experience is required before CS61C
  C++ or Java OK

- Spring 2011 statistics
  - 83% already know JAVA
  - 54% already know C++
  - 34% already know C
  - 7% already know C#
  - About 10% 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! You'll still need your C reference for this course
  - K&R is a must-have
    - Check online for more sources
  - "Brian Harvey's helpful transition notes
    - On CS61C class website
    - http://inst.eecs.berkeley.edu/~cs61c/resources/HarveyNotesC1-3.pdf
- Key C concepts: Pointers, Arrays, Implications for Memory management

C vs. Java

	C	Java
Type of Language	Function Oriented	Object Oriented
Program-ming Unit	Function	Class / Abstract Data Types
Compilation	Machine dependant assembly	Machine independent bytecode
Execution	Loads and executes program	JVM interprets bytecode

hello, world	#include<stdio.h>
	int main(void) {
	printf("Hello\n");
	return 0;
	}
		public class HelloWorld {
		public static void main(String[] args) {
		System.out.printl("Hello");
		}
		}

Memory management	Manual (**`malloc`**, **`free`**)	Automatic (garbage collection)

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

Compilation: Overview

- C *compilers* map C programs into 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 exactly when your program is mapped to low-level machine instructions
  - For C, generally a two 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

- Excellent run-time performance: generally much faster than Scheme or Java for comparable code (because it optimizes for a given architecture)
- Fair compilation time: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
- Why C?: *we can write programs that allow us to exploit underlying features of the architecture – memory management, special instructions, parallelism*

Compilation: Disadvantages

- Compiled files, including the executable, are architecture-specific, depending on CPU type and the operating system
- 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, during the development cycle

Typed Variables in C

```
int   x = 2;
float y = 1.618;
char  z = 'A';
```

- You have to declare the type of data a variable will hold

Type	Description	Examples
int	signed integer	5,-12,0
short int (short)	smaller signed integer	
long int (long)	larger signed interger	
char	single text character or symbol	'a', 'D', '?'
float	floating point non-integer numbers	0.0, 1.618, -1.4
double	greater precision FP number	

- Integer sizes are machine dependant!
  - Common size is 4 bytes, but can't ever assume this.
- Can add "unsigned" before int or char type for unsigned
  - eg unsigned int

Characters

- How do we encode characters?
- Relevant question: How many different characters do we want to encode?
  - 'a'-'z', 'A-Z', '0-9'.
  - What else?

Characters

- ASCII standard – 128 different characters and their numeric encodings.
  - A char representing the character 'a' contains 97.
  - char c = 'a'; or char c = 97 are both valid
- 7 bits is enough to store a char (2^7 = 128), round up to 1 byte since computers usually only deal with multiples of bytes.

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Big Idea – Bits can represent anything!

- Say we have 0b 0110 0011
  - Interpret it as a char – 'c'
  - Interpret it as a (signed or unsigned) integer: 99
- 0b 1111 1111
  - Interpret it as a char: (not a valid char)
  - Interpret it as a signed int: -1
  - Interpret it as an unsigned int: 255
- It all depends on how the programmer looks at them.
- Just wait until Lecture 8, MIPS Instruction Formats, for an incredible example of this.

Types in C

- C is a "weakly" typed language. You can explicitly typecast from any type to any other.
  - int i = -1;
    if(i < 0) printf("This will print\n");
    if((unsigned int) i < 0) printf("This will not print\n");
- Can be seen as changing the "programmer's perspective" of the variable.
- Can typecast anything, even if it doesn't really make sense:
  - struct node n; //We will see structs in a few slides
    int i = (int) n;
  - More freedom, but easier to shoot yourself in the foot.

Typed Functions in C

```
//Function prototypes
int multiply(int,int);
void sayHello();

int multiply(int x,int y)
{
  return x*y;
}

void sayHello ()
{
  printf("Hello\n");
}
```

- You have to declare the type of data you plan to return from a function
- Return type can be any C variable type, and is placed to the left of the function name
- You can also specify the return type as `void`
  - Just think of this as saying that no value will be returned
- Also necessary to define types for values passed into a function
- Declaring the "prototype" of a function allows you to use it before the function's definition.

C Syntax: True or False

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

Structs in C

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

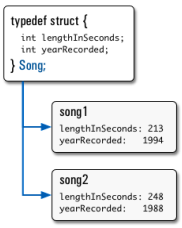
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

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

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



Consts and Enums in C

- Constant is assigned a value once in the declaration; value can't change until the program is restarted

```
const float goldenRatio = 1.618;
const int daysInWeek = 7;
```

- You can have a constant version of any of the standard C variable types
- Enums: a group of related constants used to parameterize libraries

Agenda

- C Language Overview
- Administrivia
- C Syntax and Control Flow
- Break
- Pointers
- Summary

Administrivia

- TA office hours have been decided. For this week
  - Justin – Th 1-2, F 11-12. Alcoves
  - Sean – W 3-4, Th 11-12. Moore room – 2nd floor cory courtyard.
  - Alvin – MW 12-1. Alcoves
- HW1 will be posted today.

Agenda

- C Language Overview
- Administrivia
- C Syntax and Control Flow
- Break
- Pointers
- Summary

Actual 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 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

- Similar to Java, but with a few minor but important differences
- *All* variable declarations must appear before they are used (e.g., 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++)

C Syntax : Flow Control (1/2)

- Within a function, remarkably close to Java constructs in methods (shows its legacy) in terms of flow control
  - `if-else`
    - `if (expression) statement`
    - `if (expression) statement1 else statement2`
  - `while`
    - `while (expression) statement`
    - `do statement while (expression);`

C Syntax : Flow Control (2/2)

- `for`
  - `for (initialize; check; update) statement`
- `switch`
  - `switch (expression){`
    `case const1: statements`
    `case const2: statements`
    `default: statements`
    `}`
  - `break`

Agenda

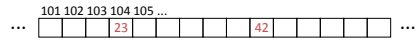
- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers

Agenda

- Scheme vs. Java vs. C
- Administrivia
- Quick Start Introduction to C
- Technology Break
- Pointers

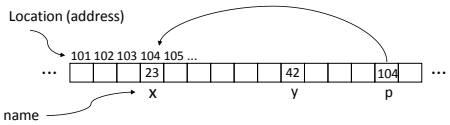
Address vs. Value

- 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 there



Pointers

- An *address* refers to a particular memory location; e.g., it points to a memory location
- *Pointer*: A variable that contains the address of a variable

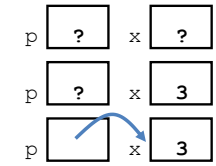


Pointers

- How to create a pointer:

  (& operator: get address of a variable)

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




```
x = 3;
```



```
p = &x;
```


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**.

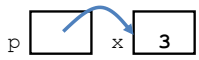
- How get a value pointed to?

  * "dereference operator": get the value that the pointer points to

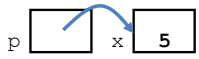
  printf("p points to %d\n",*p);

Pointers

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




```
*p = 5;
```



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 remains equal to 3

Pointers and Parameter Passing

- How can we get a function to change a value?

```
void addOne (int *p) {
   *p = *p + 1;
  }
int y = 3;

addOne(&y);
```

y is now equal to 4

Pointers

- Pointers are used to point to any kind of data (`int`, `char`, a `struct`, etc.)
- Normally a pointer only points 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 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 C code?

#Errors
a) 1
b) 2
c) 3
d) 4
e) >4

Peer Instruction Answer

```
#include <stdio.h>
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 C code?

SIX

#Errors
a) 1
b) 2
c) 3
d) 4
e) >4

More C Pointer Dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- Local variables in C are not initialized, they may contain anything (aka "garbage")
- What does the following code do?

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```


Pointers in C

- Why use pointers?
  - If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
  - In general, pointers allow cleaner, more compact code
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
    - Most problematic with dynamic memory management—which you will to know by the end of the semester, but not for the projects (there will be a lab later in the semester)
    - *Dangling references* and *memory leaks*

Why Pointers in C?

- At time C was invented (early 1970s), compilers often didn't produce efficient code
  - Computers 25,000 times faster today, compilers better
- C designed to let programmer say what they want code to do without compiler getting in way
  - Even give compilers hints which registers to use!
- Today's compilers produce much better code, so may not need to use pointers
  - Compilers even ignore hints since they do it better!

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
- 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)

And In Conclusion, …

- Bits can represent anything!
  - chars, integers, floating point, …
- C is an efficient language, but leaves safety to the programmer
  - Weak type safety
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs
- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers