

# CS 33: Introduction to Computer Organization

## Week 0

# Disc 1C

- TA: Uen-Tao Wang
- Email: [cerberiga@g.ucla.edu](mailto:cerberiga@g.ucla.edu)
- Office Hours: Tentatively (Tuesday or Thursday after 4PM), will update later
- TA Room: BH 2432

# Today's Schedule

- Administrative information
- Course introduction
- Linux overview and accessing the SEASnet Linux servers
- C (aka unlearning C++)
- Binary representation
- Binary operators

# Course Admin: Grading

- Tentative:
  - 5 Homeworks (1% each)
  - 4 Labs (10% each)
  - 3 French Hens
  - 2 Midterms (12.5% each, open book)
  - 1 Final Exam (30%, open book)
- Note: Combined, the labs are worth 40% of your grade while the midterms are worth only 25%.
- You may want to keep this in mind when midterm season approaches

# Course Admin: Personal Website

- Any materials (including slides like these) that I think you should see will be posted here:
  - [seas.ucla.edu/~uentao](http://seas.ucla.edu/~uentao)
- This includes:
  - Discussion Slides
  - Class notes that I take during lecture
  - Any additional notes or resources that will be helpful.

# Course Admin: Piazza

- There will be a Piazza.
- Stay tuned for more info...

# Course Admin: Textbook

- The course textbook:
  - Computer Systems: A Programmer's Perspective **3<sup>rd</sup> Edition**
  - After 6+ years of using the 2<sup>nd</sup> edition, we're finally entering the modern era.
  - CAUTION: The 2<sup>nd</sup> Edition, 2<sup>nd</sup> International Edition, and the 3<sup>rd</sup> Edition all have essentially the same homework questions... except the numbers are different.

# Course Admin: Textbook

- “Do I need to buy the book?”
- If you just want to read the 2<sup>nd</sup> edition... you don't need to *buy* it.
- However, since the test is open book, having a physical copy will prove to be advantageous.
- You won't find any test answers in the book, but it contains all of the fundamentals.



# Introduction to Computer Organization

- What is it anyway?

# Introduction to Computer Organization

- The highly reliable and totally peer reviewed source (wikipedia) suggests that “computer organization” is synonymous with microarchitecture.
- Microarchitecture is one half of the subjects that comprise the more general topic of “Computer Architecture”
- Introduction to Computer Organization → Introduction to Computer Architecture

# For this class

- How are higher level programming languages compiled into executable files?
  - How are variables (integers, floats, structs) stored in memory?
  - How does memory really work?
- How are programs executed by a computer?
  - How are the higher level programming instructions understood?

# Main Goal

- To make you a better person

# More specifically

- Understand how high level compiled code is converted into lower level representations and consequently run on a computer.
- Understand the decisions regarding why these design choices were made.
- Write better code based on your understanding of what will happen to the code when it is compiled and executed.

# Getting Started

- This class is based around C, not C++.
- As a result, you are highly recommended to ditch Visual Studio and work in a Linux environment, specifically the SEASnet Linux servers.
- Your assignments will be tested on the SEASnet Linux servers
- The class lectures are likely to be Linux heavy.
- Linux is love. Linux is life.

# Getting Started: Accessing the SEASnet

- For Windows users:
  - PuTTY  
(<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>): An SSH client
  - SSH stands for Secure Shell and is a protocol that is used to initiate text based access to a remote server.
- For Mac and Linux:
  - SSH is a command that can be issued directly.  
Open a terminal (for Mac: Applications -> Utilities -> Terminal)

# Getting Started: Accessing the SEASnet

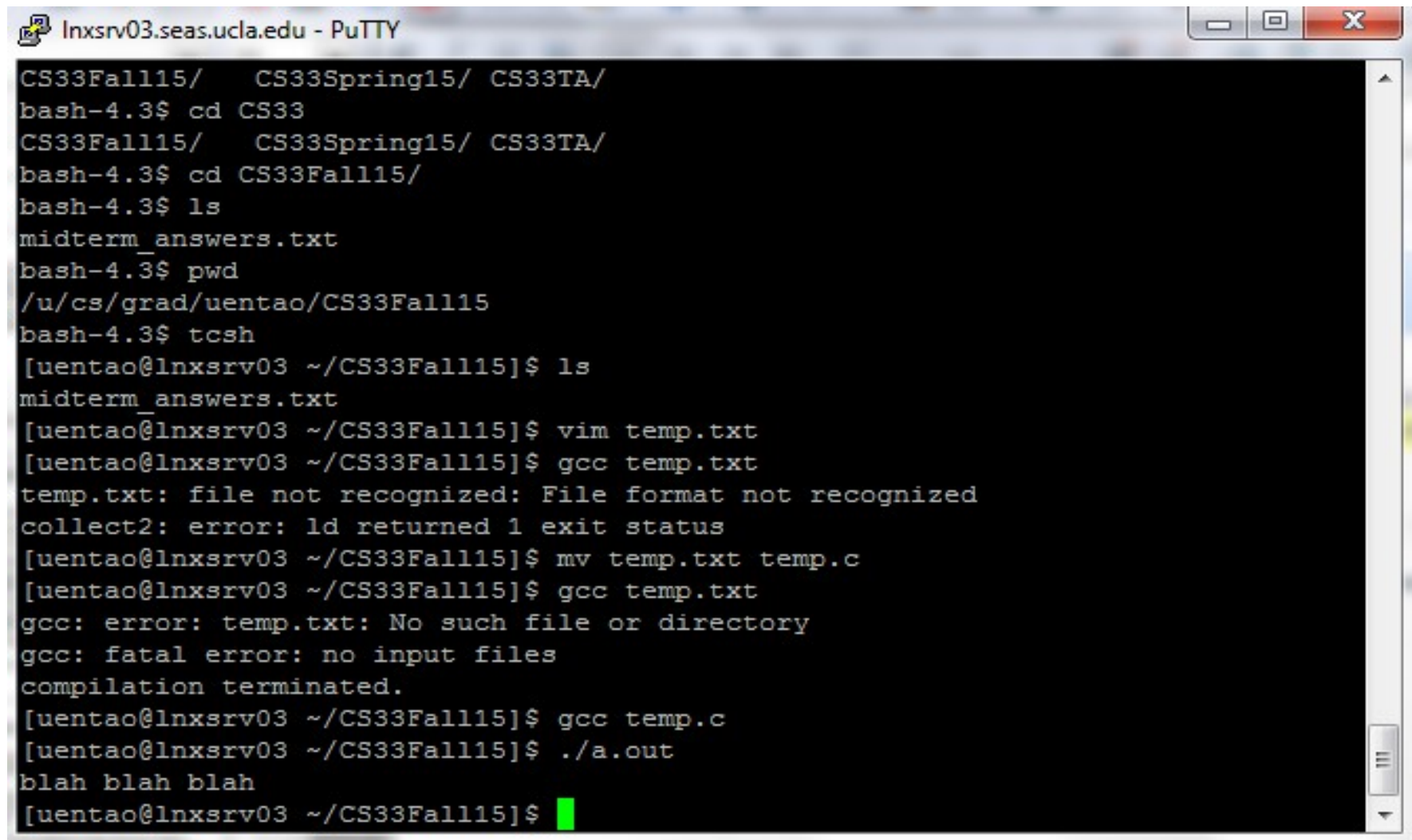
- For Windows users:
  - In “host name”, enter  
`[username]@lnxsrv.seas.ucla.edu`
- For Mac and Linux:
  - `ssh [username]@lnxsrv.seas.ucla.edu`
- If you are in the Henry Samueli School of Engineering, you should already have a SEASnet account. Otherwise go to the SEASnet office at 2684 Boelter Hall to get one.



# Getting Started: Linux Introduction

- Linux is the operating system/kernel that you will be learning about in this class as well as CS 111.
- The primary interface into Linux that you'll be using is a command line interface akin to MSDOS.
- There's a lot more to it than this, but what's important is that you'll be doing a lot of work on a screen that looks like this:

# Getting Started: Linux Introduction



The image shows a PuTTY terminal window titled "lnxsrv03.seas.ucla.edu - PuTTY". The terminal displays a series of commands and their outputs in a Linux shell environment. The user navigates through directories, lists files, and attempts to compile a C program. The session ends with a green cursor on the prompt line.

```
lnxsrv03.seas.ucla.edu - PuTTY
CS33Fall15/  CS33Spring15/  CS33TA/
bash-4.3$ cd CS33
CS33Fall15/  CS33Spring15/  CS33TA/
bash-4.3$ cd CS33Fall15/
bash-4.3$ ls
midterm_answers.txt
bash-4.3$ pwd
/u/cs/grad/uentao/CS33Fall15
bash-4.3$ tcsh
[uentao@lnxsrv03 ~/CS33Fall15]$ ls
midterm_answers.txt
[uentao@lnxsrv03 ~/CS33Fall15]$ vim temp.txt
[uentao@lnxsrv03 ~/CS33Fall15]$ gcc temp.txt
temp.txt: file not recognized: File format not recognized
collect2: error: ld returned 1 exit status
[uentao@lnxsrv03 ~/CS33Fall15]$ mv temp.txt temp.c
[uentao@lnxsrv03 ~/CS33Fall15]$ gcc temp.txt
gcc: error: temp.txt: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[uentao@lnxsrv03 ~/CS33Fall15]$ gcc temp.c
[uentao@lnxsrv03 ~/CS33Fall15]$ ./a.out
blah blah blah
[uentao@lnxsrv03 ~/CS33Fall15]$
```

# Getting Started: Useful Linux Commands

- Linux command format:
- [command name] -X -Y -Z [argument]
- (X, Y, and Z are optional flags)
- Flags modify/specify the behavior of the command.

# Getting Started: Useful Linux Commands

- `pwd` – print working directory
- `ls` – list contents of current directory
  - `ls -l` (“-l” flag will print contents in long form)
- `cd` – change directory
  - `cd blah` (navigates to the `blah` directory which is located in the current directory)
  - `cd ..` (navigate to the parent directory)
  - `cd ../dir` (navigate to a directory called “dir” that is located within the parent)
  - `cd .` (navigate to the current directory. Great work)

# Getting Started: Useful Linux Commands

- Editing files. If you're interested familiarizing yourselves with Linux (which will have to happen eventually), it is recommended that you use “vi” or “emacs”.
  - `vi text.txt`
  - `emacs text.txt`
- If you simply want to end your suffering, it is recommended that you use “nano” or “pico”.
  - `nano text.txt`
  - `pico text.txt`

# Getting Started: Useful Linux Commands

- The standard Linux C compiler is gcc.
  - gcc main.c (compile the file main.c into an executable file with default name “a.out”)
  - gcc main.c -o main (compile the file main.c into an executable file called “main”)
  - gcc main.c -O2 (compile the file with optimizations, level 2)
- Note: For lab 1, you'll be using “make” (ie make btest) to compile the project.
- Executing executables
  - ./main (executes the executable file called “main”)

# Getting Started: Copying from local machine to SEASnet

- For Windows users:
  - WinSCP: An scp client
  - scp stands for secure copy. You use it to secure copy.
  - You can get it at the incredibly sketchy looking site here: <https://winscp.net/eng/download.php>
- For Mac and Linux:
  - scp is a command that can be issued directly. Open a terminal (for Mac: Applications -> Utilities -> Terminal)

# Getting Started: Copying from local to SEASnet

- For Windows users:
  - In “host name”, enter  
`[username]@lnxsrv.seas.ucla.edu`
  - Right click a file and select upload or download.
- For Mac and Linux:
  - Local to remote:
    - `scp [file name] [username]@lnxsrv.seas.ucla.edu:.`
  - Remote to local:
    - `scp [username]@lnxsrv.seas.ucla.edu:<path to file> .`



# C (as opposed to C++)

- In a (very simplified) nutshell, C++ is an extension to C.
- The syntax of the language is nearly identical, but you will find that C lacks certain features, namely the “Object Oriented” paradigm.
- Some features are analogous, but have different names.

# C (as opposed to C++)

- In C++:
  - `for(int i = 0; i < size; i++)`  
...
- By default, gcc uses a 1990's C standard which prohibits declarations in “for” loops. As a result, you will have to do either
  - `int i;`
  - `for(i = 0; i < size; i++)`
- Or explicitly use gcc to compile with a different C standard
  - `gcc -std=c99 temp.c`

# C (as opposed to C++)

- “Or explicitly use gcc to compile with a different C standard”
- The status of graders/readers is unknown. If we (the TAs) grade your homework or labs, we will compile with the fewest flags necessary.
- As a result, take caution if a solution for this class requires a different flag or standard; it may incur a loss of points.

# C (as opposed to C++)

- Dynamic memory allocation
- In C++:
  - `char * c_arr = new char[10];`
  - `delete c_arr;`
  - “new” allows you to specify repetitions of a specific data type.

# C (as opposed to C++)

- In C, these declarations force you to be more specific. Instead of “new”, use “malloc” and instead of “delete”, use “free”.
  - `char * c_arr = (char *) malloc(sizeof(char) * 10);`
  - `free(c_arr);`
- Note: These are analogous but not the same.
- “malloc” and other “\_alloc” variations operate on the principle that you're specifying a specific amount of memory to allocate rather than a specific data type.

# C (as opposed to C++)

- Formal listing for malloc and variants of “\_alloc”
- `void * malloc(size_t size)` : allocate some amount of memory.
  - “size” is the number of bytes to allocate
- `void * calloc(size_t num, size_t size)` : allocate some amount of memory and zero out the allocated memory
  - The syntax of calloc is more closely tied to array declarations; num is the number of elements and size is the size of each element
- `void * realloc(void * ptr, size_t size)` : takes an existing pointer and reallocates the size memory allocated by that pointer, changing it's location if necessary.
  - “size” is the new number of bytes to allocate

# C (as opposed to C++)

- Some things that may be new to you:
  - What is the “void \*” data type?
  - What does it mean to do (char \*) malloc(...) ?
  - What is size\_t?
- If they aren't, skip the next few slides.

# C (as opposed to C++)

- All pointers are addresses whose size is based on the machine running the program.
- Ex. 0x04080808 is a pointer which is an address to some location in memory.
- A void \* is a pointer whose value (which is just an address) is the same as any other pointer, but there is no indication as to what type of data is intended to be stored there.
- malloc returns a void pointer.
- However, we're not trying to allocate a “void” array.



# C (as opposed to C++)

- As a result, we have to do what is called “type casting”, which is taking a variable of a specific data type and “convert” it to a variable of another data type.
- Format: (destination\_type) some\_var
- Ex: (char \*) malloc(...)
  - Convert the void \* returned by malloc and use it as a char \*.
- I say “convert”, but it's a little different from that. But more on that later...

# C AND C++

- Type casting works for other data types as well.
  - ex. `int i = (int) c;`
  - We'll cover what happens in these cases in detail later.
- Type casting actually happens implicitly in some case.
  - `int i = 10;`
  - `char c = 11;`
  - `if(i < c)`
  - Generally, what happens in this case is that the variables are cast to the same default type... but more on that later.

# C (as opposed to C++)

- Instead of:
  - `int x = 10;`
  - `cout << x;`
- You'll use “printf”
  - `printf(“blah”);`
  - `printf(“%d”, x);`
- `printf` takes in as the first parameter a string to print out that is populated with format codes that correspond to the remaining arguments.

# C (as opposed to C++)

- For example, if we want to print an integer x, we need to do the following:
  - `int x = 10;`
  - `printf("integer x: %d", x);`
  - This will print out "integer x: 10"
  - `%d` is the format code for integers
- Go here: for a full and confusing list of codes:
  - <http://www.cplusplus.com/reference/cstdio/printf/>

# C AND C++

- Finally, this notation is never explicitly taught but ALWAYS appears on these CS33 tests for some reason damn reason (it is uncanny):
- `int a = ____;`
- `int b = ____;`
- `int c = a < b ? a : b;`
- The `(x ? y : z)` notation is shorthand for:  
“if(x) then y else z”
- Thus, if a is less than b, then `c = a`. Otherwise, `c = b`.

# Binary Number Representation

- 0110 0101
- Base 2 number representation.
  - Each digit can only be one of two options, 0 or 1, hence, “bi”-nary.

# Unsigned Binary Representation:

- The base 2 method of representing numbers (compare against decimal representation, which is base 10)
- Consider the decimal number 2340:
  - $2340 = 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 0 \cdot 10^0 = 2340$
- Consider the binary number 1010:
  - $1010 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 10$
- An N bit binary number has  $2^N$  values or a range of  $[0, 2^N - 1]$

# Decimal to Binary (informal)

- Let  $d$  be a decimal value.
- We build the binary number from the least significant bit to the most significant bit. Start from the least significant bit.
- To get the current bit of the number, take the modulo of  $d$  and 2 (i.e.  $d \% 2$ ).
- Then, integer divide  $d$  by 2, that is to say if  $d$  were 5, 5 integer divided by 2 would be 2. Now we are dealing with the next bit
- Repeat the process until  $d$  is 0.



# Decimal to Binary (slightly more formal)

d is the decimal value

b is the binary value ( $b_i b_{i-1} \dots b_1 b_0$ )

$i = 0$

while  $d \neq 0$

$b_i = d \bmod 2$

$d = d / 2$  (integer)

$i += 1;$

# Decimal to Binary Example

1.  $D = 23$

$$b_0 = 23 \bmod 2 = 1$$

2.  $D = 23 / 2 = 11$

$$b_1 = 11 \bmod 2 = 1$$

3.  $D = 11 / 2 = 5$

$$b_2 = 5 \bmod 2 = 1$$

4.  $D = 5 / 2 = 2$

$$b_3 = 2 \bmod 2 = 0$$

5.  $D = 2 / 2 = 1$

$$b_4 = 1 \bmod 2 = 1$$

6.  $D = 1 / 2 = 0$

$$b = 10111$$

# Number representations

- One binary digit is a bit. Can be one of 2 values.
- $x$  bits has a range of  $2^x$  values.
- 8 bits = byte. Has a range of 256 values.
- To have a practical range of memory, we need a lot of address space, which means a lot of bits.
- In modern computers, that's likely going to be 32 or 64 bits.
- Scientifically speaking, this is known as “a lot”. This is, in part, why we have:

# Hexadecimal Representation

- The base 16 method of representing numbers.
- Hexadecimal digits range from 0-9 and A-F where A-F correspond to values 10-15
- The prefix “0x” is used to denote numbers as written in hexadecimal.
- In hexadecimal:
  - $0x234C = 2 \cdot 16^3 + 3 \cdot 16^2 + 4 \cdot 16^1 + 12 \cdot 16^0 = 9036$

# Binary <=> Hexadecimal

- Hexadecimal has the useful property where a single digit has a range that is a power of two.
- Four bits in binary have the equivalent range of one digit in hexadecimal.
- To convert from binary to hexadecimal, group the binary in sets of four and convert each set individually.

1010 1001 0110 0010 = 43362

V	V	V	V
0xA	9	6	2

# Signed Binary: Two's Complement

- How do we represent negative numbers?
- The two's complement of a number is technically its value subtracted from  $2^N$ .
- In two's complement, most bits have the same contribution as in unsigned. The value of the  $i$ -th bit is  $2^i$  (assuming  $i$  starts from 0).
- However, the most significant bit of an  $N$  bit number has a value of  $-2^{N-1}$  instead of  $2^{N-1}$ .

# Signed Binary: Example

- Assume we're dealing with four bit numbers.
- Consider the unsigned binary number 1010:
  - $1010 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10$
- Now consider the signed binary number 1010:
  - $1010 = 1*(-(2^3)) + 0*2^2 + 1*2^1 + 0*2^0 = -6$

# How to convert between negative and positive

- The method: take the bitwise inverse and add one. Consider 0101 (5).
- 1. 0101
- 2. Bitwise inverse of 0101 = 1010.
- 3.  $1010 + 0001 = 1011$ .
- 4. Confirm:  $1011 = -(2^3) + 2^1 + 2^0 = -5$



# Signed Binary: Notes

- The value of a signed binary number depends on the number of bits there are.
  - Four bit signed:  $1111 = -1$
  - Five bit signed:  $01111 = 15$
- An N-bit signed binary number has  $2^N$  possible values with a range of  $[-2^{N-1}, 2^{N-1}-1]$ .
- REMEMBER THIS: The range of a two's complement signed binary number is not symmetrical around 0.
- Henceforth all signed binary is two's complement unless otherwise specified.

# Binary arithmetic

- What does it mean to add bits?
- The idea is the same as in decimal. Let's try with unsigned.

0001	0001	0001	0011
+ 0010	+ 0001	+ 0111	+ 0111
-----	-----	-----	-----
0011	0010	1000	1010

Note that adding two or three 1 bits will produce a carry bit that must be added to the next bit over.

# Binary arithmetic

- How about subtraction? You can generalize the decimal method for binary, but...
- The simplest way to do  $X - Y$  is to do  $X + (-Y)$ .
- Take  $0110$  (6) –  $0010$  (2)
- This becomes  $0110$  (6) +  $1110$  (-2)

```
  0110
+ 1110
-----
  0100
```

<---- What happened to the last  
carry bit? Does it matter?

# Unsigned overflow in C

- With signed arithmetic, we saw that a carry out bit was completely valid, but what about unsigned?
- Say we have 4-bit numbers and we add  $6 + 12$ .
- $6 = 0110$ ,  $12 = 1100$

0110

+1100

-----

10010 = 18

- ...but this requires 5 bits to represent. We only have 4.
- How does the wise and venerable C respond?

# Unsigned overflow in C

- Just drop bits.
- If an unsigned operation of an n-bit number requires more than n bits, the resulting number will consist only of the n least significant bits.
  - Ex. In the previous example:
  - $0110 + 1100 = (1) 0010$ , the leading one is dropped and instead of the right answer of 18, you get the incredibly wrong answer of 2
- More formally, if you have n-bits, the computation of  $x + y$  is  $(x + y) \% 2^n$ .
  - Ex.  $(6 + 12) \% 2^4 = 18 \% 16 = 2$

# Unsigned overflow in C

- This is also true of unsigned multiplication overflow:
- If we have 4 bits,  $6 * 12 = 72$
- In binary, 72 is 1001000.
- Truncate bits beyond 4 and the result is 1000 = 8.
- $72 \% 2^4 = 8$
- Keep in mind, this is for unsigned numbers only.
- Let's not think about signed numbers for now.

# Datatypes in C

- Each native datatype in C is expressed by a sequence of bits.
- Simple data types such as ints and shorts come in unsigned and signed variants where signed is the default (ie. int is actually a signed int)
- However, the number of bits used to express these numbers differs depending on whether the processor is 32 or 64-bit.
- ...but more on the processor definitions later.

# Datatypes in C

- char/unsigned char : 8-bits
- short/unsigned short : 16-bits
- int/unsigned (int) : 32-bits
- Here's where it gets weird.
- In 32-bit machines:
  - long/unsigned long : 32-bits
  - long long/unsigned long long (further proof that a five year old named these) : 64-bits
- In 64-bit machines:
  - long/unsigned long : 64-bits
  - long long/unsigned long long (ugh) : 64-bits



# For your homework...

- You are tasked with creating a variant of `_alloc` called:
  - `void *arealloc (void *ptr, size_t nmemb, size_t size);`
  - ...where it just does `realloc (ptr, nmemb * size)`
- You are to compile your code with the following:
  - `gcc -O2 -Wall -Wextra -c arealloc.c`
- If you run this on the SEASnet server (which is a 64-bit machine), this will compile in 64-bit mode.
- In a 64-bit machine, a `size_t` is an unsigned integer of 64 bits (or an unsigned long/unsigned long long).

# Boolean Operators

- Boolean operators operate on a single bit.
- AND :  $\&$ 
  - Result is 1 if both inputs are 1.
- OR :  $|$ 
  - Result is 1 if either of the inputs are 1.
- XOR :  $\wedge$ 
  - Result is 1 if one input is 1 and the other is 0
- NOT :  $\sim$ 
  - Result is 1 if the input is 0.

# Bitwise Operators

- Bitwise operators perform repeated boolean operations on each bit of a number or pair of numbers.
- Bitwise invert (not the same as logical invert or '!')
  - $\sim(1011) = 0100$
- Bitwise AND/OR (not the same as logical AND/OR or  $\&\&/||$ )
  - $1010 \& 1100 = 1000$
  - $1010 | 1100 = 1110$
- Bitwise XOR
  - $1010 \oplus 1100 = 0110$

# Bitwise Operators

- Left shift/right shift (arithmetic vs logical)
- Left shift
  - $0111 \ll 1 = 1110$
- Right shift
  - $1011 \gg 1 = 0101$  (logical)
  - $1011 \gg 1 = 1101$  (arithmetic)
- Why two different right shifts?

# Logical Operators

- Whereas bitwise operators operate on each individual bit of a number, logical operators operate on the number as a whole
  - `||`, `&&`, `!`
- To invert a bit sequence  $x$ , you would use  $\sim x$ . What happens if you use the logical invert `!`?
- $!(1010) = 0$
- $!(0111) = 0$
- $!(0) = 1$

# Logical Operators

- What happens when you use logical operators on numbers?
- $1011 \ \&\& \ 1100$ ?
- Non-zero numbers are interpreted as 1 and 0 is interpreted as... 0.
- $1011 \ \&\& \ 1100 \ ==> \ 1$
- $1011 \ \&\& \ 0 \ ==> \ 0$
- $1011 \ || \ 0 \ ==> \ 1$

# Useful Tricks: Determining if number is zero

- x is a bit vector
- ```
if(x == 0)
    return 0;
else
    return 1;
```

# Useful Tricks: Determining if number is zero

- `!X` is 0 if `X` is non-zero and 1 if `X` is zero.
- `!!X` is 0 if `X` is zero and 1 if `X` is non-zero.

```
if(x == 0)
```

```
    return 0;
```

```
else                <==>    return !!x;
```

```
    return 1;
```



# Useful Tricks: Conditioning without “if”

- a, b, and c are bits
- if(a)  
    return b;
- else  
    return c;

# Useful Tricks: Conditioning without “if”

- We know that we want to return `b` only if `a == 1`.
  - This can be represented as `a & b`.
- We want to return `c` only if `a == 0`.
  - This can be represented as `~a & c`
- `return (a & b) | (~a & c)`

# Useful Tricks: Representing AND expressions with OR and vice versa

- How can we represent  $c = a \& b$  without using AND?
- DeMorgan's Law:
  - $a \& b = \sim(\sim a \mid \sim b)$
  - $a \mid b = \sim(\sim a \& \sim b)$

# Useful Tricks: Multiplication via shifting

- Consider the 4-bit unsigned number 0110.
  - $0110 = 2^3 * 0 + 2^2 * 1 + 2^1 * 1 + 2^0 * 0 = 2^2 + 2^1 = 6$
- $0110 \ll 1 = 1100$ 
  - $1100 = 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0 = 2^3 + 2^2$
  - $2^3 + 2^2 = 2 * (2^2 + 2^1) = 12$
- $x \ll n = x * 2^n$

# Useful Tricks: Multiplication via shifting

- How can we think of multiplying two arbitrary (ie non-powers of two) numbers in binary?
- $0110 * 1011 (= 6 * 11 = 66)$   
 $= 0110 * (1000 + 0010 + 0001)$   
 $= 0110 * 1000 + 0110 * 0010 + 0110 * 0001$   
 $= 0110 \ll 3 + 0110 \ll 1 + 0110$   
 $= 0110000 + 01100 + 0110 = 1000010 \text{ (correct!)}$

# Useful Tricks: Division via shifting

- By the same logic, this ought to work for division right?
- Consider 4-bit unsigned:
  - $1100 = 12$
  - $1100 \gg 1 = 0110 = 6$  (looks good...)
- Consider 4-bit signed:
  - $1100 = -4$
  - $1100 \gg 1 = 0110 = 6$  (hrm?)

# Useful Tricks: Division via shifting

- Previously, we tried logical right shifting (shift in zeros, but that didn't seem to pan out). This is where arithmetic right shifting steps in.
- Consider 4-bit signed:
  - $1100 = -4$
  - $1100 \gg 1 = 1110 = -2$
- Logical shifting maintains correct values for unsigned operations while arithmetic shifting maintains correct values for signed operations.

# Useful Tricks: Division via shifting

- Consider the 4-bit signed number 1101.
  - $1101 = -2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = -2^3 + 2^2 + 2^0 = -3$
- $1101 \gg 1 = 1110$ 
  - $1110 = -2^3 * 1 + 2^2 * 1 + 2^1 * 1 + 2^0 * 0 = -2$
- $-3 / (\text{integer}) 2 = -1$ , not  $-2$
- How do you resolve this?



# Useful Tricks: Extracting specific bits

- Say you have the binary value 1010 and you only want to consider bits 1 and 2, that is, you want to transform 1010 into 0010.

# Useful Tricks: Extracting specific bits

- 1010
- & 0110
- -----
- 0010
- Can be extended to hexadecimal. Recall that 0xF is 1111.  
If we have 0x33221100 and we want to extract bits 8-15:
- 0x33221100
- & 0x0000FF00
- -----
- 0x00001100

**End of**  
**The First Week**  
**-Nine Weeks Remain-**