15-213 Recitation: Data Lab

____TA____
23 Jan 2017

Agenda

- Introduction
- Course Details
- Data Lab
  - Getting started
  - Running your code
  - ANSI C
- Bits & Bytes
- Integers
- Puzzles

Introduction

- Welcome to 15-213/18-213/15-513!
- Recitations are for…
  - Reviewing lectures
  - Discussing homework problems
  - Interactively exploring concepts
  - Previewing future lecture material

- Please, **please** ask questions!

Course Details

- How do I get help?
  - Course website: http://cs.cmu.edu/~213
  - Office hours: **5-9PM** from Sun-Thu in Wean 5207
  - *Definitely* consult the course textbook
  - Piazza
  - **Carefully read the assignment writeups!**
- All labs are submitted on Autolab.
- All labs should be worked on using the **shark machines.**

Data Lab: Getting Started

- Download lab file (`datalab-handout.tar`)
  - Upload tar file to **shark** machine
  - `cd <my course directory>`
  - `tar xpvf datalab-handout.tar`
- Upload `bits.c` file to Autolab for submission

Data Lab: Running your code

- `dlc`: a modified C compiler that interprets *ANSI C* **only**
- `btest`: runs your solutions on random values
- `bddcheck`: exhaustively tests your solutions
  - Checks all values, formally verifying the solution
- `driver.pl`: Runs both dlc and bddcheck
  - Exactly matches Autolab's grading script
  - You will likely only need to submit once
- For more information, **read the writeup**
  - Available on theproject.zone
  - **Read it. Read the writeup… please.**

Data Lab: What is ANSI C?

This is *not* ANSI C.

Within two braces, all *declarations* must go before any *expressions*.

```
unsigned int foo(unsigned int x)
{
        x = x * 2;
        int y = 5;

        if (x > 5) {
            x = x * 3;
            int z = 4;
            x = x * z;
        }

        return x * y;
}
```

Data Lab: What is ANSI C?

This is ANSI C.

```
unsigned int foo(unsigned int x)
{
        int y = 5;
        x = x * 2;

        if (x > 5) {
            int z = 4;
            x = x * 3;
            x = x * z;
        }

        return x * y;
}
```

This is *not* ANSI C.

```
unsigned int foo(unsigned int x)
{
        x = x * 2;
        int y = 5;

        if (x > 5) {
            x = x * 3;
            int z = 4;
            x = x * z;
        }

        return x * y;
}
```

Bits & Bytes: Unsigned integers

- An unsigned number represents positive numbers between 0 and 2^k-1, where *k* is the numbers of bits used.
- Subtracting 1 from 0 will *underflow* to the highest value.
- Adding 1 to the highest value will *overflow* to 0

An 8-bit unsigned integer:

1		1		0		1		0		1		0		1	
2^7	+	2^6		+		2^4		+		2^2		+		2^0	= 213
128		64				16				4				1	

Bits & Bytes: Two's Complement

- A signed number represents positive numbers between -2^{k-1} and $2^{k-1}-1$, where *k* is the numbers of bits used.
- Subtracting 1 from the smallest value will *underflow* to the highest value
- Adding 1 to the highest value will *overflow* to the smallest value
- An 8-bit signed integer:

1	1	0	1	0	1	1	0

$$-2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -42$$

$$-128 \quad 64 \quad 16 \quad 4 \quad 2$$

Bits & Bytes: Two's Complement

To get the negative value of a positive number *x*, invert the bits of *x* and add 1.

From positive to negative:

21	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**

Bits & Bytes: Two's Complement

To get the negative value of a positive number *x*, invert the bits of *x* and add 1.

From positive to negative:

21	0	0	0	1	0	1	0	1	
-22	1	1	1	0	1	0	1	0	**Bits negated**

Bits & Bytes: Two's Complement

To get the negative value of a positive number *x*, invert the bits of *x* and add 1.

From positive to negative:

21	0	0	0	1	0	1	0	1	
-22	1	1	1	0	1	0	1	0	**Bits negated**
-21	1	1	1	0	1	0	1	1	**Add one**

C standard

- The C standard does not state that two's complement is used to represent signed numbers.
  - Partly because of this, signed overflow and underflow are listed in the standard as examples of *undefined behavior*.
  - For this lab, you can assume that two's complement is used.

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 =	513 || 0 =	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 =	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 = 1	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&

15 && 18 = 1

OR: ||

513 || 0 = 1

EQ: ==

15 == 18 = 0

NOT: !

!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 = 1	15 == 18 = 0	!15213 = 0

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101			

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101	11101101		

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101	11101101	10001000	

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101	11101101	10001000	00010010

Bits & Bytes: Shifting

Shifting modifies the positions of bits in a number:

				1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	**= x**
~~1~~	~~0~~	~~1~~	~~1~~	0	0	0	1	0	1	0	1	1	0	0	1	1	0	0	0	0	**= x << 4**
																					= x * 2^4

Shifting right on **a signed number** will *extend the sign:*

x =	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1				
x >> 4 = **x / 2^4 =**	1	1	1	1	1	0	1	1	0	0	0	1	0	1	0	1	1	~~0~~	~~0~~	~~1~~	~~1~~

(If the sign bit is zero, it will fill in with zeroes instead.)

This is known as "arithmetic" shifting.

Bits & Bytes: Shifting

Shifting right on **an *unsigned* number** will fill in with 0.

x =	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	
x >> 4 = **x / 2^4 =**	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	1	1	~~0 0 1 1~~

This is known as “logical” shifting.

Arithmetic shifting is useful for preserving the sign when dividing by a power of 2.

We get around this when we don’t need it by using *bitmasks.*

In other languages, such as Java, it is possible to choose shifting operators, regardless of the type of integer. In C, however, it depends on the signedness.

Form Groups of 3 - 4

- Series of exercises
    - Operators
    - Puzzles

Open-ended questions

- How many bits are there in an `int`? Why do you think this size is used?
- Which `int` values would you consider as edge cases in a program? Which ones are most useful for bitwise operations? For boolean operations?
- On a bitwise level, what similarities are there between signed and unsigned arithmetic?

Questions?

- Remember, data lab is due next Thursday! (Feb 2nd)
  - You really should have started already!
- Read the lab writeup.
  - **Read the lab writeup.**
    - ***Read the lab writeup.***
      - ***<u>Read the lab writeup.</u>***
        - **<u>Please. :)</u>**