Integers II

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

This is definitely unsigned overflow!

Announcements, Reminders

- HW3 due tonight, HW4 due Friday (05 Apr)
- Lab 1a due Monday (8 Apr)
  - Use `ptest` and `dlc.py` to check your solution for correctness (on the **CSE Linux environment**)
  - Submit `pointer.c` and `lab1Asynthesis.txt` to Gradescope
    - Make sure you pass the File and Compilation Check – all the correct files were found and there were no compilation or runtime errors
- Lab 1b releases tomorrow, due next Monday (15 Apr)
  - Bit manipulation on a custom encoding scheme
  - Bonus slides at the end of today’s lecture have examples for you to look at 😉

Reading Review

- Terminology:
  - `UMin`, `UMax`, `TMin`, `TMax`
  - Type casting: implicit vs. explicit
  - Integer extension: zero extension vs. sign extension
  - Modular arithmetic and arithmetic overflow
  - Bit shifting: left shift, logical right shift, arithmetic right shift

Review Questions

- What is the value and encoding of **`Tmin`** `(minimum signed value)` for a fictional 7-bit wide integer data type?

$-2^6 = \boxed{-64}$

$\underset{-2^6}{1}\ \underset{2^5}{0}\ \underset{2^4}{0}\ \underset{2^3}{0}\ \underset{2^2}{0}\ \underset{2^1}{0}\ \underset{2^0}{0}$

- For `unsigned char uc = 0xB3;`, what are the produced data for the cast **`(unsigned short)uc`**?

0x B3 → 0x 00 B3

two bytes → short!

one byte → char!

- What is the result of the following expressions?
  - **`(signed char)uc >> 2`**
  - **`(unsigned char)uc >> 3`**

sign extend

11

0b 1011 00~~11~~ → 0b 1110 1100

$\boxed{0xEC}$

000

0b 1011 0~~011~~ → 0b 0001 0110

no sign extend

$\boxed{0x16}$

Why Does Two's Complement Work?

- For all representable positive integers x, we theoretically want:

$$\begin{array}{r} \textit{bit representation of } x \\ + \textit{ bit representation of } {-x} \\ \hline 0 \end{array}$$ (**ignoring** the carry-out bit)

We want the *additive inverse!*

- What are the 8-bit negative encodings for the following?

$$\begin{array}{r} 00000001 \\ +\ ???????? \\ \hline 00000000 \end{array} \qquad \begin{array}{r} 00000010 \\ +\ ???????? \\ \hline 00000000 \end{array} \qquad \begin{array}{r} 11000011 \\ +\ ???????? \\ \hline 00000000 \end{array}$$

1   -1?   2   -2?   -61   61?

Why Does Two's Complement Work?

- For all representable positive integers x, we theoretically want:

$$\begin{array}{r} \textit{bit representation of } \; x \\ + \; \textit{bit representation of } -x \\ \hline 0 \end{array}$$

(**ignoring** the carry-out bit)

  - What are the 8-bit negative encodings for the following?

```
   00000001        00000010        11000011
+  11111111     +  11111110     +  00111101
 ----------      ----------      ----------
  100000000       100000000       100000000
```

ignore this; this overflow is consequence of making two's complement work! Plus, the limited number of bits doesn't help...

These are the bitwise complement plus 1!

`-x == ~x + 1`

Integers

- **Binary representation of integers**
  - **Unsigned and signed**
  - **Casting in C**
- Consequences of finite width representations
  - Sign extension, overflow
- Shifting and arithmetic operations

Values To Remember (Review)

- Unsigned Values
  - UMin = 0b00…0
    = $\boxed{0}$
  - UMax = 0b11…1
    = $\boxed{2^w - 1}$

- Two's Complement Values
  - TMin = 0b10…0
    = $\boxed{-2^{w-1}}$
  - TMax = 0b01…1
    = $\boxed{2^{w-1} - 1}$
  - -1 = 0b11…1

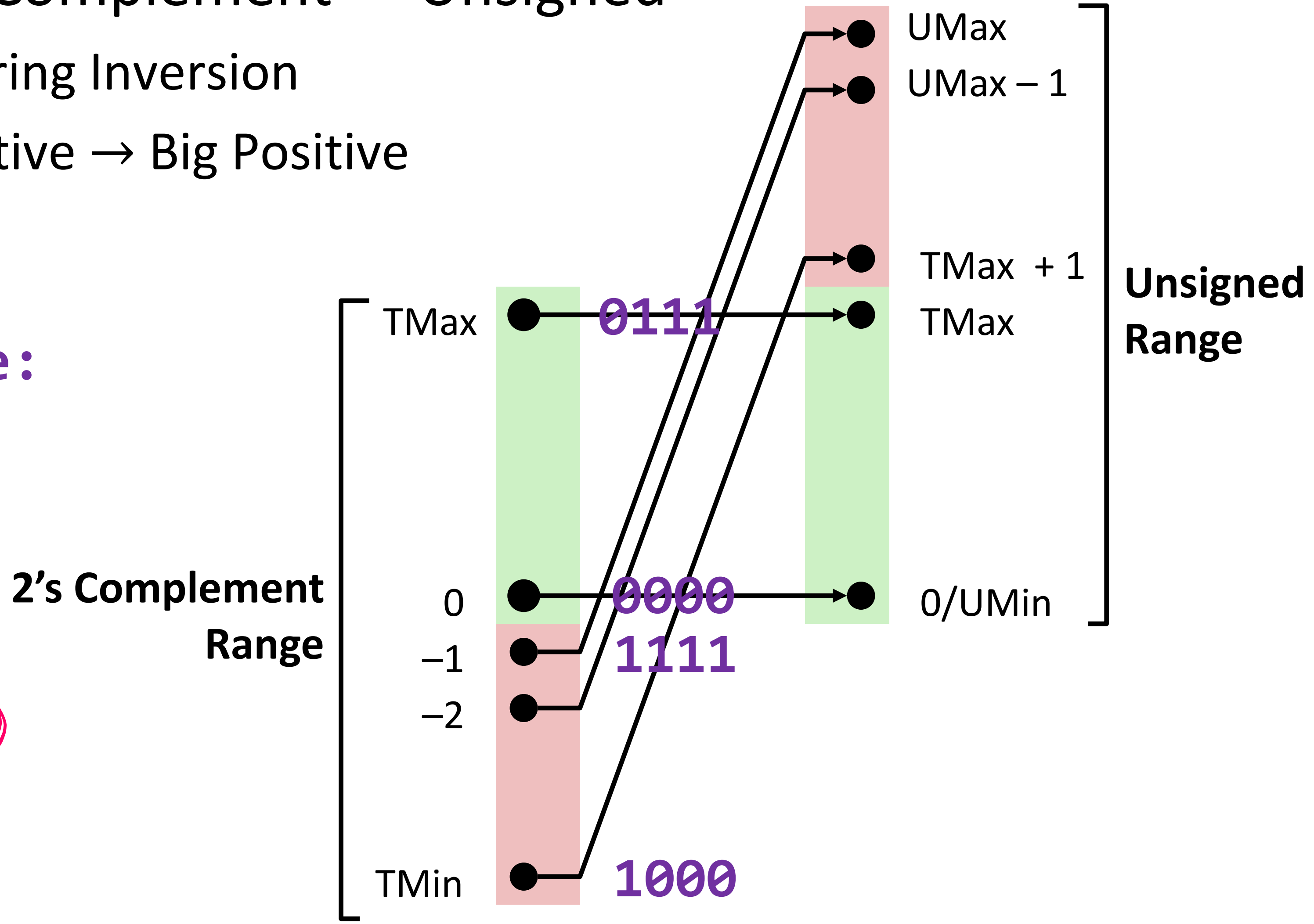
- **Example:** Values for $w = 64$

	Decimal	Hex	
UMax	18,446,744,073,709,551,615	FF FF FF FF FF FF FF FF	// All 1's!
TMax	9,223,372,036,854,775,807	7F FF FF FF FF FF FF FF	] 2's C range
TMin	-9,223,372,036,854,775,808	80 00 00 00 00 00 00 00	
-1	-1	FF FF FF FF FF FF FF FF	// All 1's!
0	0	00 00 00 00 00 00 00 00	// All 0's!

Signed/Unsigned Conversion Visualized

- Two's Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

4-bit Example:



Oooh, such fancy animations

In C: Signed vs. Unsigned (Review)

- Casting
  - Bits are unchanged, just interpreted differently! It's all about interpreting encodings!
    - **int** tx, ty; // signed by default
    - **unsigned int** ux, uy;
  - **Explicit** casting: (preferred over implicit casting...)
    - tx = (**int**) ux;
    - uy = (**unsigned int**) ty;
  - **Implicit** casting can occur during assignments or function calls:
    - tx = ux;
    - uy = ty;

Another example:

signed char sc = -1;

unsigned char uc = sc; // uc is equal to 255_{10} now! !!

Casting Surprises (Review)

- Integer literals (constants)
  - By default, integer constants are considered *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - **Examples:** 0U, 4294967259u

// for legibility 'U' preferred over 'u'
// Using suffix forces machine to interpret
// as unsigned. Though technically optional.

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, then **signed values are implicitly cast to unsigned**

i.e. unsigned has precedence!

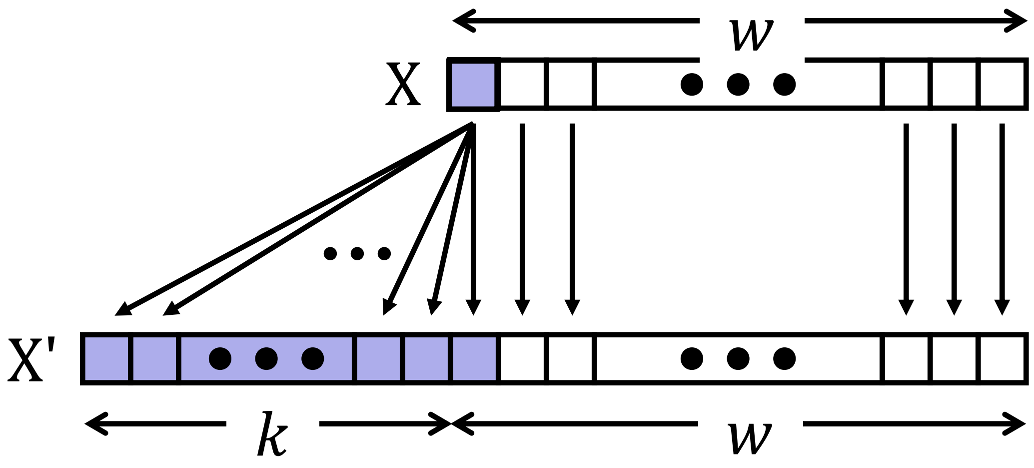
  - Including comparison operators <, >, ==, <=, >=
  - Yeah, no idea why. Thanks, C.

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- **Consequences of finite width representations**
  - **Sign extension, overflow**
- Shifting and arithmetic operations

Sign Extension (Review)

- **Task:** Given a w-bit signed integer X, convert it to w+k-bit signed integer X' *with the same value*
- **Rule: Add k copies of sign bit** (ensures sign is maintained.)
  - Let x_i be the i-th digit of X in binary
  - $\mathrm{X}' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \ldots, x_1, x_0}_{\text{original } \mathrm{X}}$



Ex:

0b1000 = -8_{10}

Make it 5 bit: 0b11000 = -8

Value does not change.

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition 😇
  - **Algorithm:** simple addition, discard the highest carry bit
    - Called modular addition: result is sum, then *modulo* by 2^w

Ex: 0b 1111
\+ 1
10000

discard via modulo !!

Arithmetic Overflow (Review)

What Homer did.

Bits	Unsigned	Signed
0000	0 Umin	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15 UMax	-1

TMax

Tmin

- What happens a calculation produces a result that can’t be represented in the current encoding scheme?
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions
- Well… C and Java ignore overflow exceptions
  - You end up with a bad value in your program and get no warning/indication… oops!

If we add 1 to this, is it overflow?
Well, yes, but it takes us to 0.
Good overflow! :)

Overflow: Unsigned

- **Addition:** drop carry bit (wrong by -2^N)

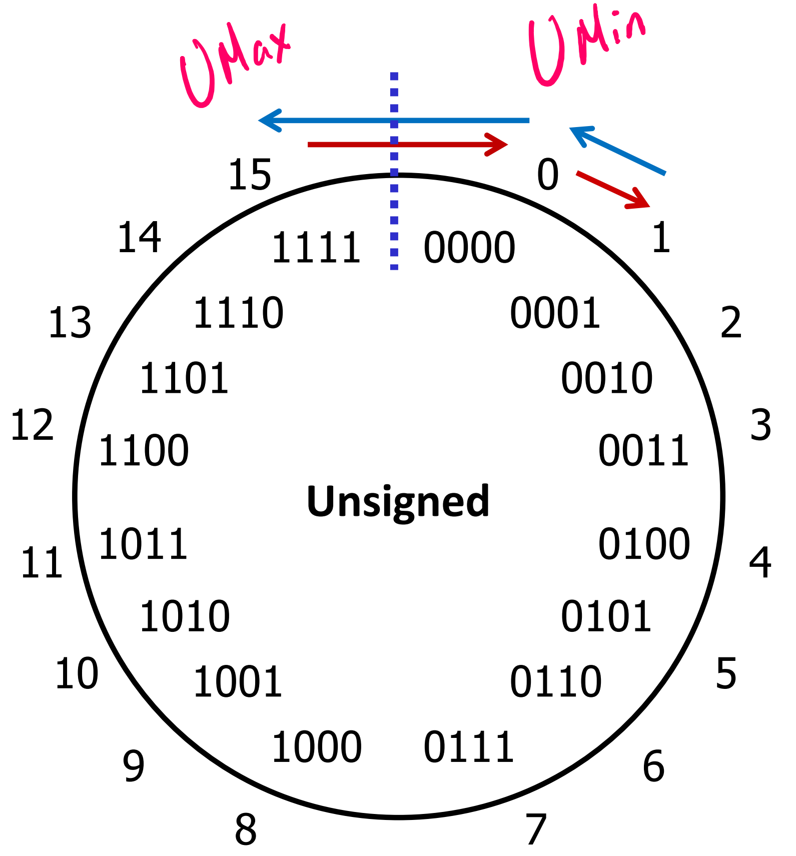
$$\begin{array}{r} 15 \\ +\ \ 2 \\ \hline \cancel{17} \end{array} \qquad \begin{array}{r} 1111 \\ +\ \ 0010 \\ \hline \cancel{1}0001 \end{array}$$

Actually: 1 !!

- **Subtraction:** borrow (wrong by $+2^N$)

$$\begin{array}{r} 1 \\ -\ \ 2 \\ \hline \cancel{-1} \end{array} \qquad \begin{array}{r} 10001 \\ -\ \ 0010 \\ \hline 1111 \end{array}$$

Actually: 15 !!



Over/Under by $\pm 2^N$ because of modular arithmetic

Here, $2^4 = 16$

Overflow: Two's Complement

- **Addition:** (+) + (+) = (−) result?

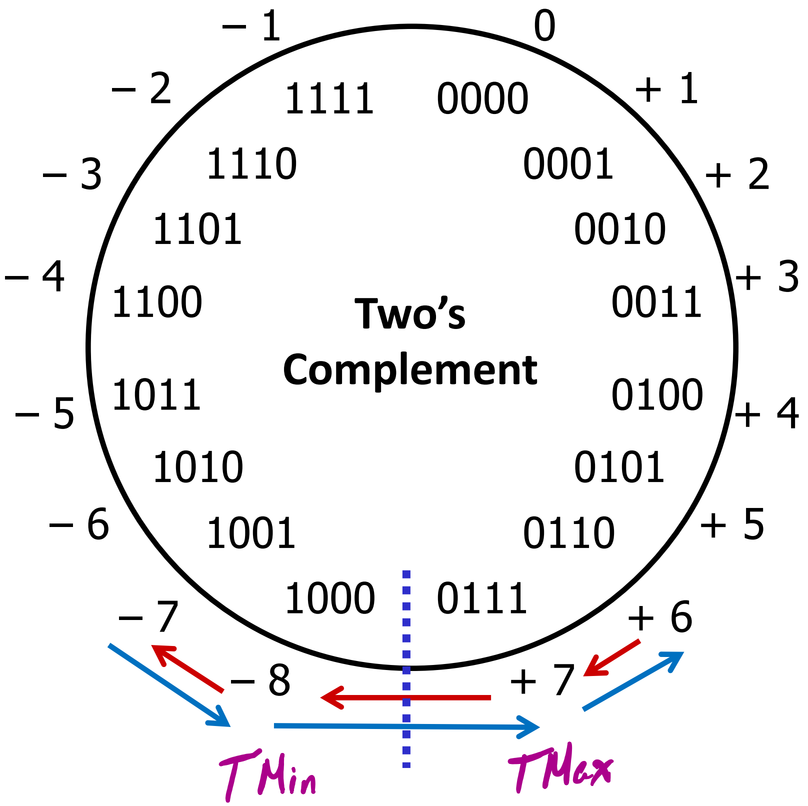
$$\begin{array}{r} 6 \\ +\ \ 3 \\ \hline \cancel{9} \end{array} \qquad \begin{array}{r} 0110 \\ +\ \ 0011 \\ \hline 1001 \end{array}$$

Actually: −7 !!

- **Subtraction:** (−) + (−) = (+)?

$$\begin{array}{r} -7 \\ -\ \ 3 \\ \hline -\cancel{10} \end{array} \qquad \begin{array}{r} 1001 \\ -\ \ 0011 \\ \hline 0110 \end{array}$$

Actually: 6 !!



For signed: overflow happened if operands have same sign and result's sign is different

Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Sign extension, overflow
- **Shifting and arithmetic operations**

Last time: Bit masks.

Now, looking back, we could have isolated the suit bits via bit shifting instead:

$\boxed{0\,|\,0\,|\,\underbrace{1\,|\,0}_{\text{suit}}\,|\,\underbrace{1\,|\,0\,|\,1\,|\,0}_{\text{value}}}$

We could have logically shifted instead:

$\boxed{0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,0\,|\,\underbrace{1\,|\,0}_{\text{suit}}}$

1 0 1 0

Bye, value bits!

They're supposed to be "falling off"...

Shift Operations (Review)

Always: Throw away (drop) extra bits that "fall off" either end

- Left shift (x<<n) bit vector x by n positions
  - Fill with 0's on right
- Right shift (x>>n) bit-vector x by n positions
  - For unsigned values: Logical shift—Fill with 0's on left
  - For signed values: Arithmetic shift—Replicate most significant bit on left. Maintains sign of x! Exactly like we did with sign extension!

8-bit Examples:

Ex: 0x22

	x	0010 0010
	x<<3	0001 0000
logical:	x>>2	0000 1000
arithmetic:	x>>2	0000 1000

left, always 0's.

Ex: 0xA2

	x	1010 0010
	x<<3	0001 0000
logical:	x>>2	0010 1000
arithmetic:	x>>2	1110 1000

Shift Operations (Review)

- Arithmetic:
  - Left shift (x<<n) is equivalent to multiply by 2^n
  - Right shift (x>>n) is equivalent to divide by 2^n
  - **Compiler Hack:** Shifting is faster than general multiply and divide operations!

- Notes:
  - Shifts by n<0 or n≥w (w is bit width of x) are *undefined*
  - **In C:** behavior of >> is determined by the compiler
    - In gcc / clang, depends on data type of x (signed/unsigned)
  - **In Java:** logical shift is >>> and arithmetic shift is >>

Yep, you can shift away all your data...

Left Shifting 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: `x*2`^{n}?

(In a perfect world)

			Signed	Unsigned	No Overflow
`x = 25;`	`00011001`	=	25	25	25
`L1=x<<2;`	`00|01100100`	=	100	100	100
`L2=x<<3;`	`000|11001000`	=	-56	200	200
`L3=x<<4;`	`0001|10010000`	=	-112	144	400

signed overflow

unsigned overflow

Lost some data!

Right Shifting 8-bit Examples

- **Reminder:** C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Logical Shift: `x/2`^{n}?

(In a perfect world)

			Unsigned	No Rounding
`xu = 240u;`	`11110000`	=	240	240
`R1u=xu>>3;`	`00011110000`	=	30	30
`R2u=xu>>5;`	`0000011110000`	=	7	7.5?

rounding (down)

`//`
`^`

Right Shifting 8-bit Examples

- **Reminder:** C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - Arithmetic Shift: `x/2`^{n}?

			Signed	No Rounding
`xs = -16;`	`11110000`	=	-16	-16
`R1s=xs>>3;`	`11111110000`	=	-2	-2
`R2s=xs>>5;`	`1111111110000`	=	-1	-0.5

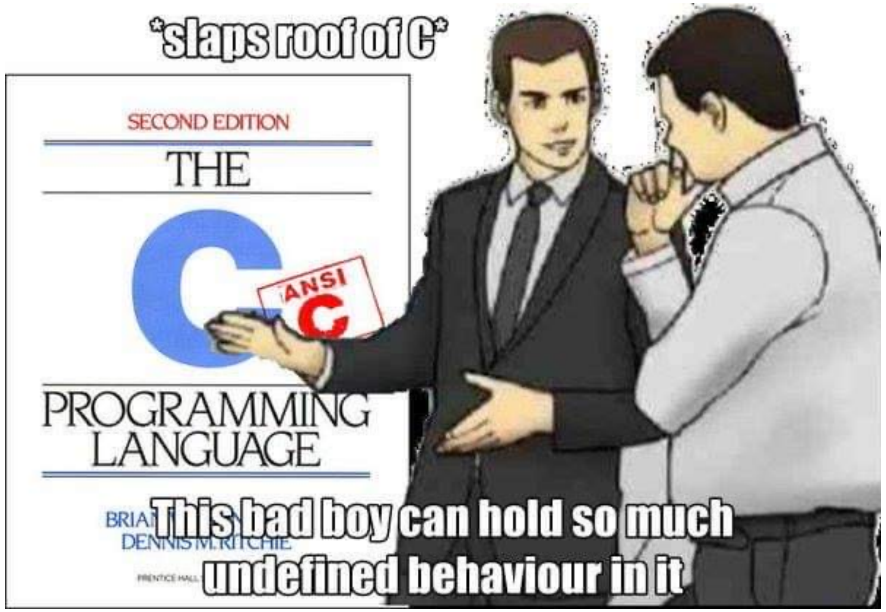
rounding (down)

Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpreted* differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
    - Type of variables affects behavior of operators (shifting, comparison)
- We can only represent so many numbers in w bits
  - When we exceed the limits, *arithmetic overflow* occurs
  - *Sign extension* tries to preserve value when expanding
- Shifting is a useful bitwise operator
  - Right shifting can be arithmetic (sign) or logical (0)
  - Can be used in multiplication with constant or bit masking

Undefined Behavior in C

- How much **undefined behavior** have we talked about in just the past few lectures?
  - Shifting by more than size of type
  - No bounds checking in arrays
  - Pointer nonsense
  - Mystery data in unassigned variables
  - ...and there will be more! 🥴



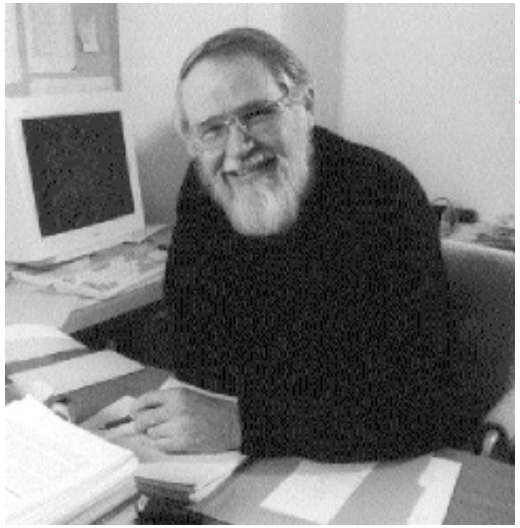
What does this tell us about the values that were embedded in C?

C language (1978)

- Developed beginning in 1971, “standardized” in 1978
  - Goal of writing Unix (precursor to Linux, macOS and others)
  - Different time— **faced with significant performance and resource limits**
- Explicit Goals:
  - Portability, performance (better than B, it’s C!)

Ken Thompson

Dennis Ritchie



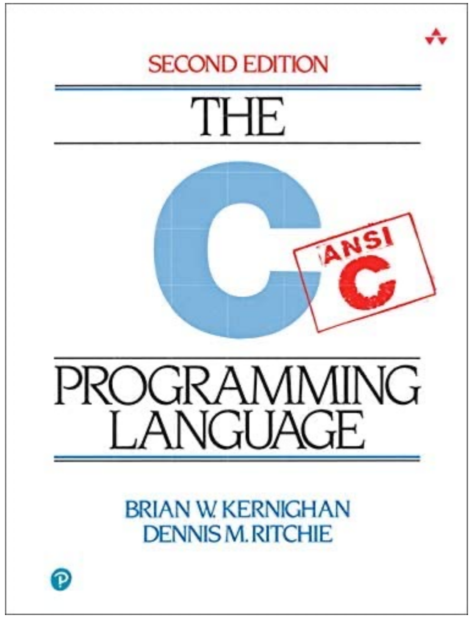
Brian Kernighan

Your Perspectives on C

- What have you noticed about the way that C works?
  - What does it make **easy?**
    - Very discrete memory manipulation
    - Total control of memory space
  - What does it make **difficult?**
    - Writing safe code...
    - Strings
    - Pointer sorcery

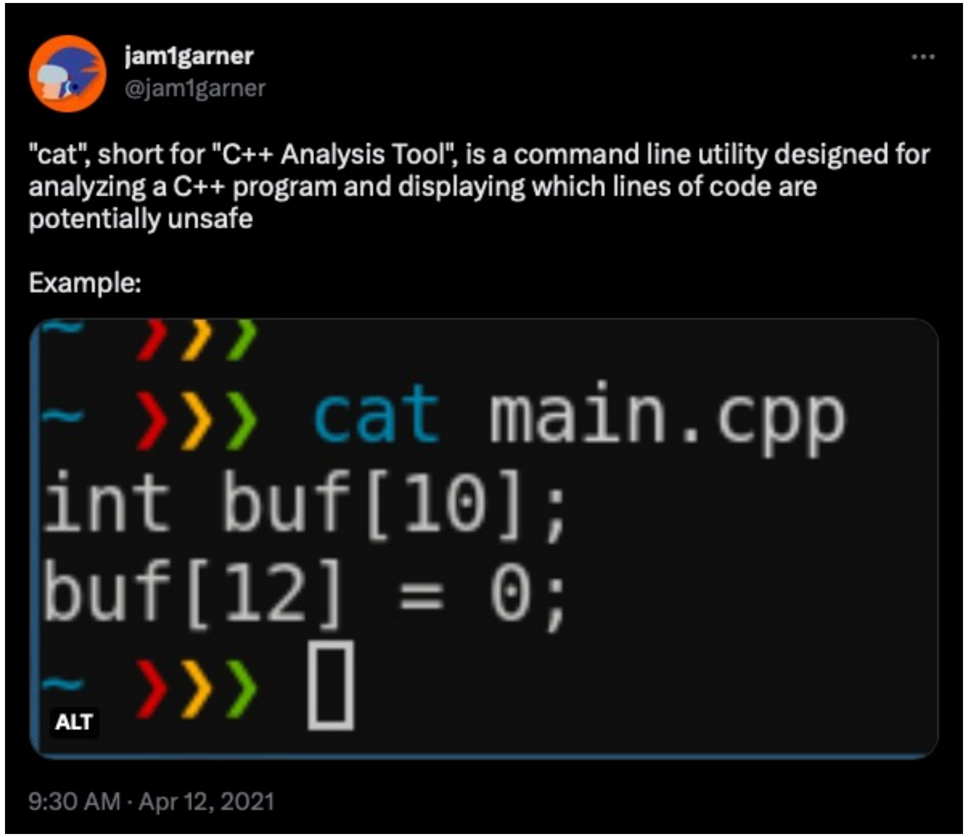
Perspectives on C

- **Minimalist**
  - Relatively small, can be described in a small space, and learned quickly (or so it's claimed)
  - "Only the bare essentials"
- **Rugged**
  - Close to the *hardware*
  - Shows what's *really happening*
- **Eliteness**
  - "Real programmers can do pointer arithmetic!"
  - Quickly slides into a "Back in my day!" situation…



Consequences of C

- "C is good for two things: being beautiful and creating catastrophic 0days in memory management." - Link to Medium Post
- "We shape our tools, and thereafter, our tools shape us." – John Culkin, 1967
- White House says no to C/C++! Is Joe Biden a rustacean?



Also applies to C, of course.

Maybe C is like… cilantro?

As a Latina, I love cilantro :)

- Maybe you love it!
- Maybe you hate it!
- Maybe your feelings are more complicated than that!

- We're not trying to force you one way or another, we only ask that you try to appreciate both its **benefits** and its **shortcomings**.
- Mainly using C as a tool to understand computers. ♡

BONUS SLIDES

Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1b.

- Extract the 2nd most significant byte of an `int`
- Extract the sign bit of a signed `int`
- Conditionals as Boolean expressions

Practice Question 1

- Assuming 8-bit data (*i.e.*, bit position 7 is the MSB), what will the following expression evaluate to?
  - UMin = 0, UMax = 255, TMin = -128, TMax = 127

- `127 < (signed char) 128u`

Practice Questions 2

- Assuming 8-bit integers:
  - `0x27` = 39 (signed) = 39 (unsigned)
  - `0xD9` = -39 (signed) = 217 (unsigned)
  - `0x7F` = 127 (signed) = 127 (unsigned)
  - `0x81` = -127 (signed) = 129 (unsigned)

- For the following additions, did signed and/or unsigned overflow occur?
  - **`0x27 + 0x81`**

Signed: $39_{10} + (-127)_{10} = -88_{10}$

no signed overflow

Unsigned: $39_{10} + 129_{10} = 168$

no unsigned overflow

  - **`0x7F + 0xD9`**

Signed: $127_{10} + (-39)_{10} = 88_{10}$

no signed overflow

Unsigned: $127_{10} + 217_{10} = 344_{10}$

unsigned overflow!!!

Exploration Questions

For the following expressions, find a value of `signed char x`, if there exists one, that makes the expression True.

- Assume we are using 8-bit arithmetic:

Expression	Example:	All solutions:
`x == (unsigned char) x`	$x = 0$	$\forall x$
`x >= 128U`	$x = -1$	$\forall x,\ x < 0$
`x != (x>>2)<<2`	$x = 3$	Any x where 2 LSB's aren't 0b00
`x == -x` • Hint: there are two solutions	$x = 0$	① x = 0b00...00 = 0 ② x = 0b100...0 = -128
`(x < 128U) && (x > 0x3F)`		Any x where 2 MSB's are 0b01

Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: `(x>>16) & 0xFF`

x	00000001 00000010 00000011 00000100
x>>16	00000000 00000000 00000001 00000010
0xFF	00000000 00000000 00000000 11111111
(x>>16) & 0xFF	00000000 00000000 00000000 00000010

  - Or first mask, then shift: `(x & 0xFF0000)>>16`

x	00000001 00000010 00000011 00000100
0xFF0000	00000000 11111111 00000000 00000000
x & 0xFF0000	00000000 00000010 00000000 00000000
(x&0xFF0000)>>16	00000000 00000000 00000000 00000010

Using Shifts and Masks

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask: `(x>>31) & 0x1`
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear `1`s possibly shifted in

`x`	`00000001 00000010 00000011 00000100`
`x>>31`	`00000000 00000000 00000000 00000000`
`0x1`	`00000000 00000000 00000000 00000001`
`(x>>31) & 0x1`	`00000000 00000000 00000000 00000000`

`x`	`10000001 00000010 00000011 00000100`
`x>>31`	`11111111 11111111 11111111 11111111`
`0x1`	`00000000 00000000 00000000 00000001`
`(x>>31) & 0x1`	`00000000 00000000 00000000 00000001`

Using Shifts and Masks

- Conditionals as Boolean expressions
  - For **`int`** `x`, what does `(x<<31)>>31` do?

`x=!!123`	`00000000 00000000 00000000 00000001`
`x<<31`	`10000000 00000000 00000000 00000000`
`(x<<31)>>31`	`11111111 11111111 11111111 11111111`
`!x`	`00000000 00000000 00000000 00000000`
`!x<<31`	`00000000 00000000 00000000 00000000`
`(!x<<31)>>31`	`00000000 00000000 00000000 00000000`

  - Can use in place of conditional:
    - In C: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z;`
    - `a=(((!!x<<31)>>31)&y) | (((!x<<31)>>31)&z);`