

Integers II

CSE 351 Winter 2022

Instructor:

Sam Wolfson

Teaching Assistants:

Angela Xu

Anirudh Kumar

Catherine Guevara

Dara Stotland

Harrison Bay

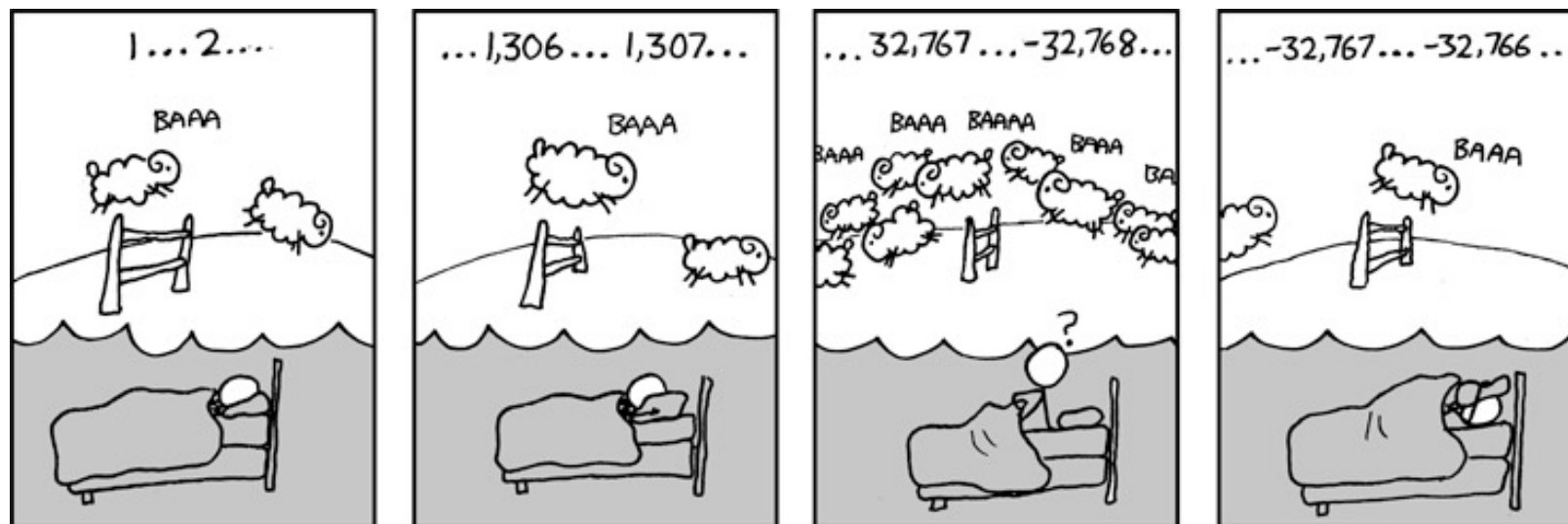
Ian Hsiao

Kevin Wang

Mara Kirdani-Ryan

Nick Durand

Sanjana Sridhar



<http://xkcd.com/571/>

Relevant Course Information

- ❖ hw4 due tonight, hw5 due Friday (1/14)
- ❖ Lab 1a due Wednesday (1/19)
 - Use `ptest` and `d1c.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 released today, due 1/21
 - Bit manipulation on a custom encoding scheme
 - Bonus slides at the end of today's lecture have relevant examples

Reading Review

❖ Terminology:

- U_{\min} , U_{\max} , T_{\min} , T_{\max}
- 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

❖ Questions from the Reading?

Review Questions

- ❖ What is the value (and encoding) of **TMin** for a fictional 6-bit wide integer data type?
- ❖ For `unsigned char uc = 0xA1;`, what are the produced data for the cast **(unsigned short)uc**?
- ❖ What is the result of the following expressions?
 - **(signed char)uc >> 2**
 - **(unsigned char)uc >> 3**

Why Does Two's Complement Work?

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

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

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

$$\begin{array}{r} 00000001 \\ + \text{ ? ? ? ? ? ? ? ? } \\ \hline 00000000 \end{array}$$

$$\begin{array}{r} 00000010 \\ + \text{ ? ? ? ? ? ? ? ? } \\ \hline 00000000 \end{array}$$

$$\begin{array}{r} 11000011 \\ + \text{ ? ? ? ? ? ? ? ? } \\ \hline 00000000 \end{array}$$

Why Does Two's Complement Work?

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

$$\frac{\text{bit representation of } x + \text{bit representation of } -x}{0} \quad (\text{ignoring the carry-out bit})$$

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

$$\begin{array}{r} 00000001 \\ + 11111111 \\ \hline 100000000 \end{array}$$

$$\begin{array}{r} 00000010 \\ + 11111110 \\ \hline 100000000 \end{array}$$

$$\begin{array}{r} 11000011 \\ + 00111101 \\ \hline 100000000 \end{array}$$

These are the bitwise complement plus 1!

$$-x == \sim x + 1$$

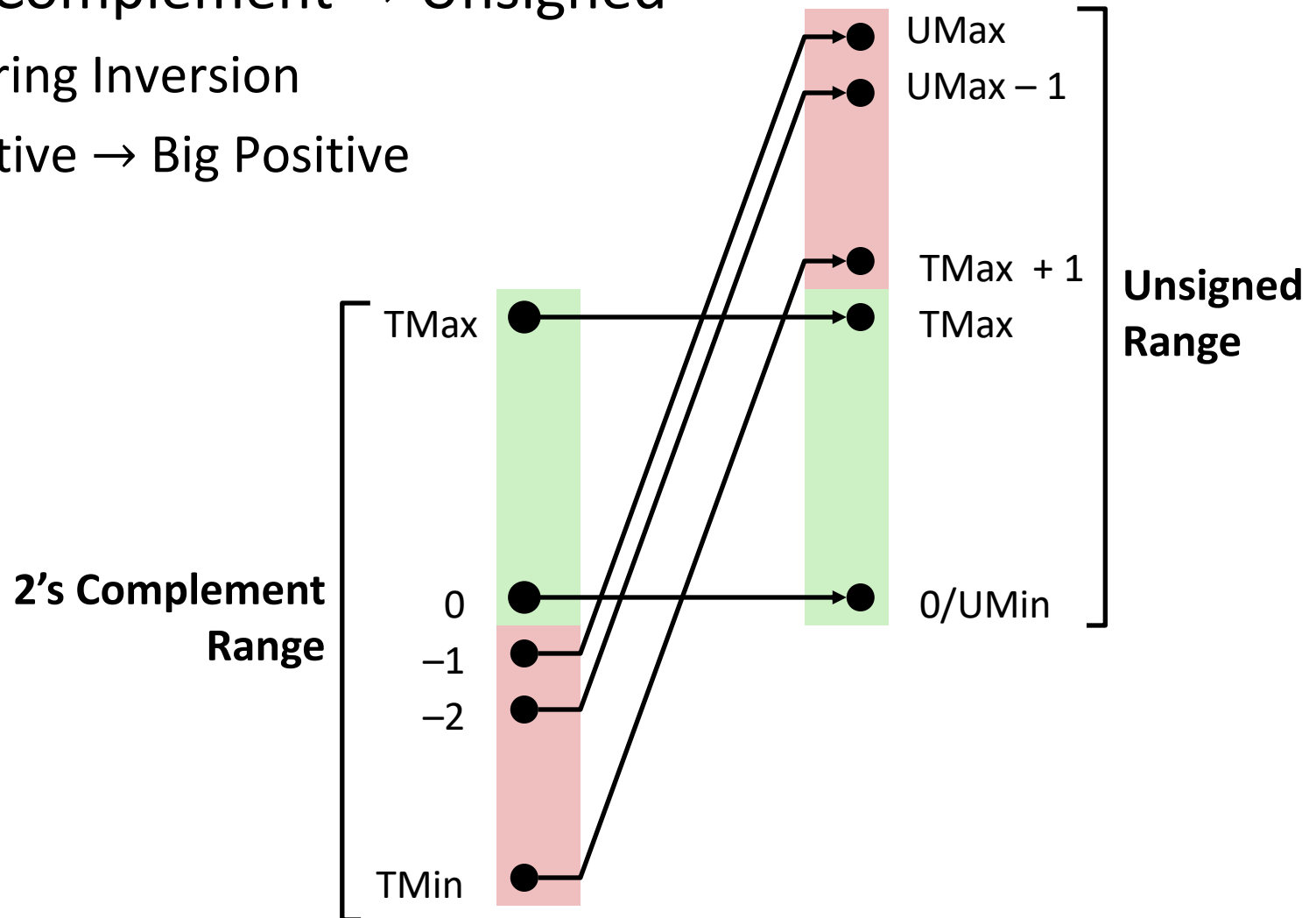
Integers

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

Signed/Unsigned Conversion Visualized

❖ Two's Complement \rightarrow Unsigned

- Ordering Inversion
- Negative \rightarrow Big Positive



Values To Remember (Review)

❖ Unsigned Values

- UMin = 0b00...0
= 0
- UMax = 0b11...1
= $2^w - 1$

❖ Two's Complement Values

- TMin = 0b10...0
= -2^{w-1}
- TMax = 0b01...1
= $2^{w-1} - 1$
- -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
TMax	9,223,372,036,854,775,807	7F FF FF FF FF FF FF FF
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
0	0	00 00 00 00 00 00 00 00

In C: Signed vs. Unsigned (Review)

❖ Casting

- Bits are unchanged, just interpreted differently!
 - `int tx, ty;`
 - `unsigned int ux, uy;`
- *Explicit* casting
 - `tx = (int) ux;`
 - `uy = (unsigned int) ty;`
- *Implicit* casting can occur during assignments or function calls
 - `tx = ux;`
 - `uy = ty;`



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

❖ Expression Evaluation

- When you mixed unsigned and signed in a single expression, then **signed values are implicitly cast to unsigned**
- Including comparison operators $<$, $>$, $==$, $<=$, $>=$

Integers

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

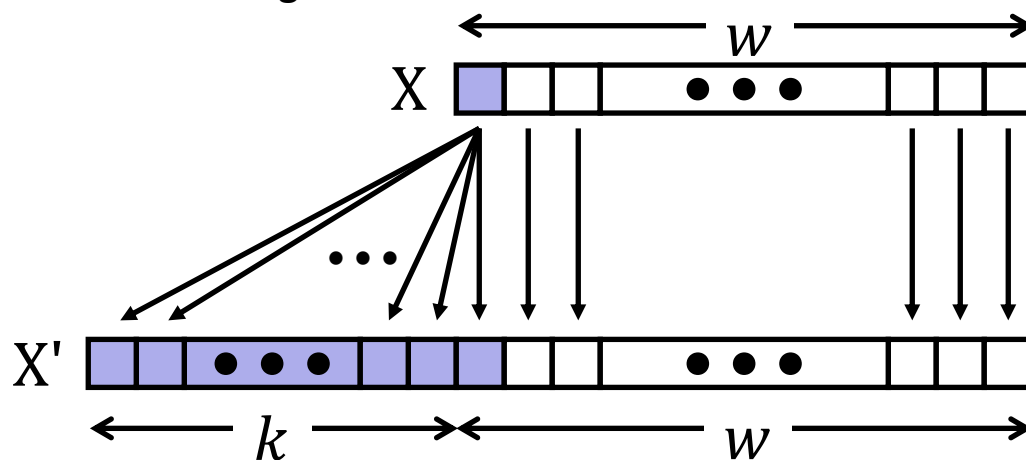
Sign Extension (Review)

❖ **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

■ Let x_i be the i -th digit of X in binary

■ $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \dots, x_1, x_0}_{\text{original } X}$



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 *modulo* 2^w

Arithmetic Overflow (Review)

Bits	Unsigned	Signed
0000	0	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	-1

- ❖ When 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
- ❖ C and Java ignore overflow exceptions
 - You end up with a bad value in your program and no warning/indication... oops!



Sam is just unlucky.

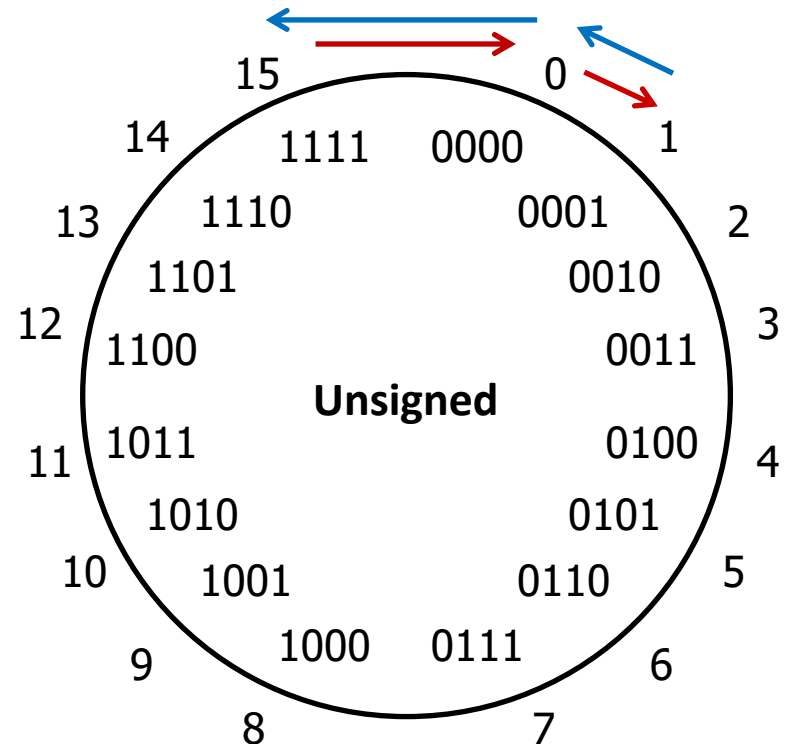
Overflow: Unsigned

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

15	1111
+ 2	+ 0010
17	1 0001
1	

- ❖ **Subtraction:** borrow ($+2^N$)

1	10001
- 2	- 0010
-1	1111
15	



$\pm 2^N$ because of
modular arithmetic

Overflow: Two's Complement

❖ **Addition:** $(+) + (+) = (-)$ result?

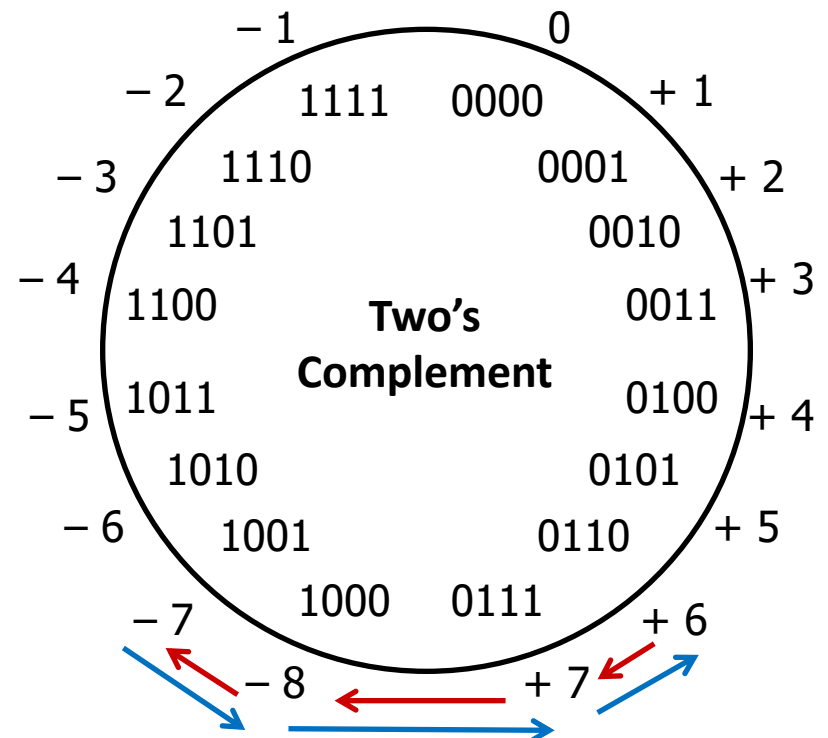
$$\begin{array}{r} 6 \\ + 3 \\ \hline \cancel{9} \\ -7 \end{array}$$

$$\begin{array}{r} 0110 \\ + 0011 \\ \hline 1001 \end{array}$$

❖ **Subtraction:** $(-) + (-) = (+)$?

$$\begin{array}{r} -7 \\ - 3 \\ \hline \cancel{-10} \\ 6 \end{array}$$

$$\begin{array}{r} 1001 \\ - 0011 \\ \hline 0110 \end{array}$$



For signed: overflow 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**

Shift Operations (Review)

- ❖ Throw away (drop) extra bits that “fall off” the end
- ❖ Left shift ($x \ll n$) bit vector x by n positions
 - Fill with 0's on right
- ❖ Right shift ($x \gg n$) bit-vector x by n positions
 - Logical shift (for **unsigned** values)
 - Fill with 0's on left
 - Arithmetic shift (for **signed** values)
 - Replicate most significant bit on left (maintains sign of x)

	x	0010	0010
	$x \ll 3$	0001	0 000
logical:	$x \gg 2$	00 00	1000
arithmetic:	$x \gg 2$	00 00	1000

	x	1010	0010
	$x \ll 3$	0001	0 000
logical:	$x \gg 2$	00 10	1000
arithmetic:	$x \gg 2$	11 10	1000

Shift Operations (Review)

❖ Arithmetic:

- Left shift ($x \ll n$) is equivalent to multiply by 2^n
- Right shift ($x \gg n$) is equivalent to divide by 2^n
- Shifting is faster than general multiply and divide operations!

❖ Notes:

- Shifts by $n < 0$ or $n \geq w$ (w is bit width of x) are *undefined*
- **In C:** behavior of \gg is determined by the compiler
 - In gcc / C lang, depends on data type of x (signed/unsigned)
- **In Java:** logical shift is \ggg and arithmetic shift is \gg

Left Shifting Arithmetic 8-bit Example

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

		Signed	Unsigned
$x = 25;$	00011001 =	25	25
$L1 = x \ll 2;$	0001100100 =	100	100
$L2 = x \ll 3;$	00011001000 =	-56	200
$L3 = x \ll 4;$	000110010000 =	-112	144

signed overflow

unsigned overflow

Right Shifting Arithmetic 8-bit Examples

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

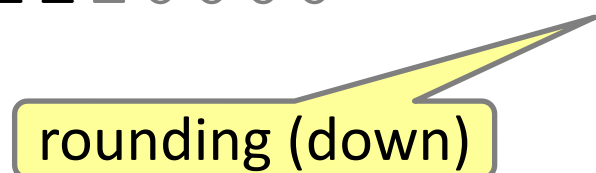
`xu = 240u;` 11110000 = 240

`R1u=xu>>3;` 00011110000 = 30



`R2u=xu>>5;` 0000011110000 = 7

rounding (down)



Right Shifting Arithmetic 8-bit Examples

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

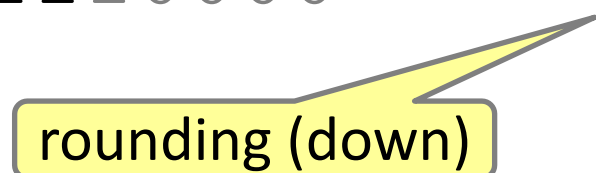
`xs = -16;` 11110000 = -16

`R1s = xu >> 3;` 11111110000 = -2



`R2s = xu >> 5;` 1111111110000 = -1

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

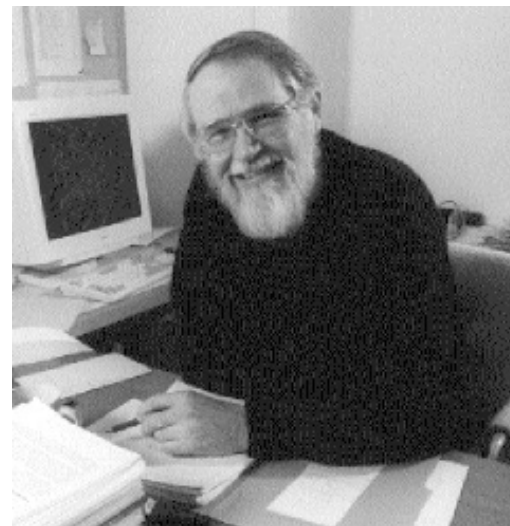
- ❖ How much **undefined behavior** have we talked about in just the past 5 lectures?
 - Integer overflow
 - Pointers, pointers, pointers
 - Mystery data in unassigned variables
 - ...and there will be more (I promise)



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

C language (1978)

- ❖ Created in 1972, “standardized” in 1978
 - Goal of writing Unix (precursor to Linux, macOS and others)
 - Different time, greater performance and resource limits
- ❖ Explicit Goals:
 - Portability, performance (better than B, it's C!)



Principles of C, viewed today

- ❖ “*Since C is relatively small, it can be described in small space, and learned quickly.*”
- ❖ “Only the bare essentials”
- ❖ “Close to the *hardware*”
- ❖ “Shows what’s *really happening*”
- ❖ “No one to help you”
- ❖ “You’re on your own”
- ❖ “I know what I’m doing, get out of my way”

Principles of C, viewed today

❖ Minimalist:

- “*Since C is relatively small, it can be described in small space, and learned quickly.*”
- “Only the bare essentials”

❖ Rugged:

- “Close to the *hardware*”
- “Shows what’s *really happening*”

❖ Individualistic

- “No one to help you”
- “You’re on your own”
- “I know what I’m doing, get out of my way”

**Minimalism, Rugged,
Individualistic...
Pioneers!**

Pioneers in the Wild West

- ❖ American Frontierism (~1800 – 1890)
 - Vast expansion westward, from original 13 colonies to Pacific Ocean
- ❖ Manifest Destiny
 - Burgeoning theory that White Americans were “destined” to connect from coast to coast
 - Cultural phenomenon, Indigenous genocide

Manifest Destiny

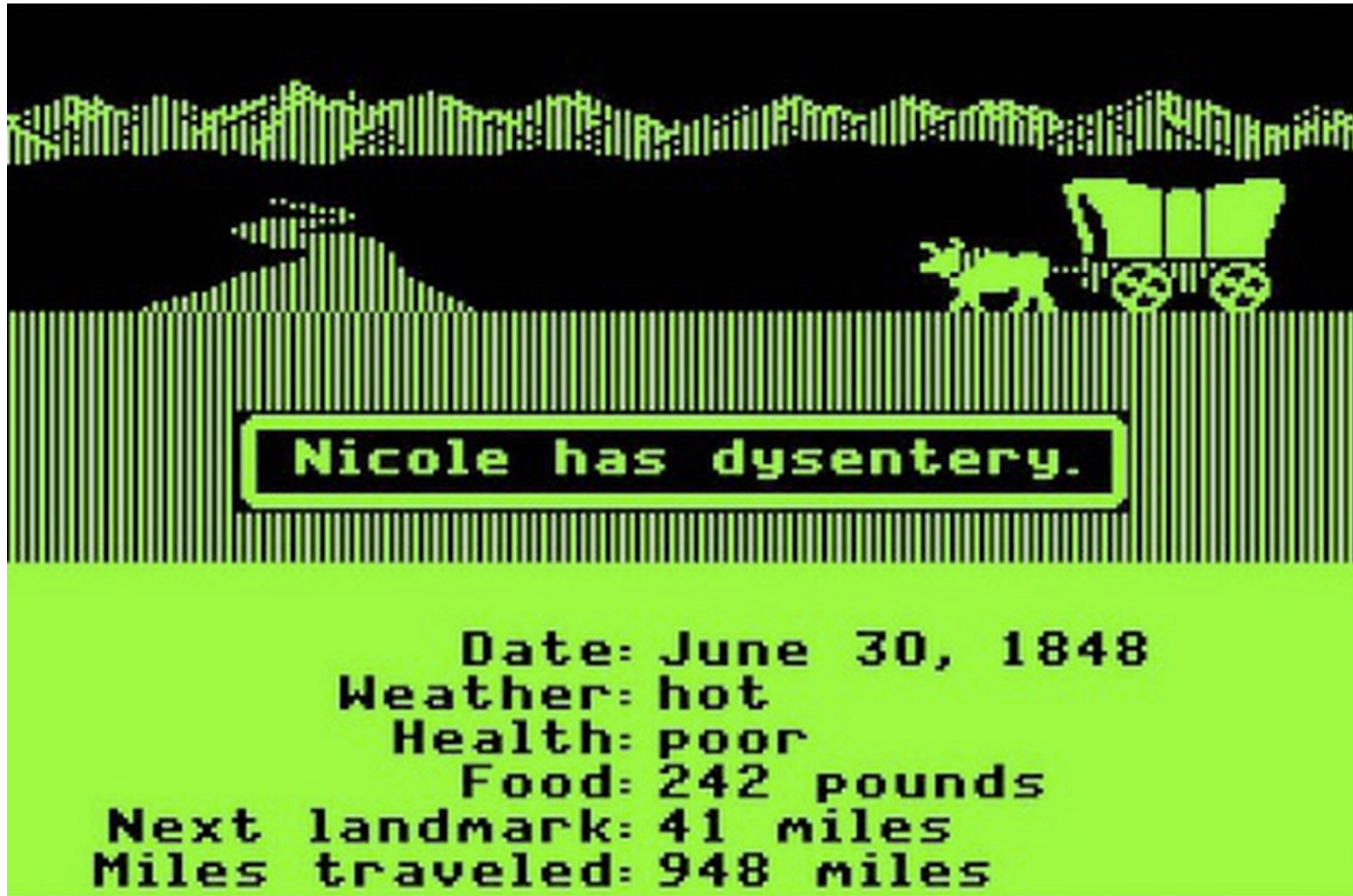


John Gast, *American Progress*, 1872

Immortalized in Popular Culture



Replicated in Computing Culture



Takeaways

- ❖ **C: Minimalistic, Rugged, Individualistic**
 - Embodied what was culturally valued at the time!
 - Frontierism! Moon landing was 1969!
- ❖ Explore the digital frontier!
 - Only carry the essentials!
 - American frontierism!
 - Manifest destiny (1800 – 1890), genocide
 - Glorified in popular culture: westerns, video games
- ❖ K&R didn't mean to do harm!
 - But they didn't question the values glorified by society
 - And C's laissez-faire attitude **has** caused harm...

**“C is good for two things:
being beautiful and
creating catastrophic 0days
in memory management.”**

<https://medium.com/message/everything-is-broken-81e5f33a24e1>

Ideology: You don't even need to ask

Contrast with: “best”, “better”, “more important”

**“We shape our tools, and
thereafter, our tools shape us”**

1967

“Reification”, if you want a single word. To make the abstract concrete.

**Computing is a tool, but a tool built by a distinctly non-neutral society!
We’ve always had values!**

C is like camping!



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

signed
 ❖ $127 < (\text{signed char}) 128u$
 0b01111111 0b10000000

← both sides are signed, so signed comparison

signed comparison:

0b01111111

127

<

False

0b10000000

-128

unsigned comparison:

127

<

True

128

(*e.g.*, if LHS was 127u)

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?
 - **$0x27 + 0x81$**

 - **$0x7F + 0xD9$**

Exploration Questions

$uMin = 0, uMax = 255$
 8-bits, so $TMin = -128, TMax = 127$

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:

■ $x \overset{\text{unsigned}}{==} (\text{unsigned char}) x$

Example:
 $x = 0$

All solutions:
 works for all x

■ $x \overset{\text{unsigned}}{>=} 128U$
 $0b1000\ 0000$

$x = -1$

any $x < 0$

■ $x \neq (x >> 2) << 2$

$x = 3$

any x where lowest two bits are not $0b00$

■ $x == -x$

$x = 0$

① $x = 0b0\dots0 = 0$
 ② $x = 0b10\dots0 = -128$

• Hint: there are two solutions

■ $(x < 128U) \ \&\& \ (x > 0x3F)$

any x where upper two bits are exactly $0b01$

Using Shifts and Masks

❖ Extract the 2nd most significant *byte* of an `int`:

- First shift, then mask: $(x \gg 16) \ \& \ 0xFF$

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: $(x \ \& \ 0xFF0000) \gg 16$

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: $(x \gg 31) \ \& \ 0x1$
 - Assuming arithmetic shift here, but this works in either case
 - Need mask to clear 1s possibly shifted in

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

x	10000001 00000010 00000011 00000100
x>>31	11111111 11111111 11111111 11111111 1
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?

<code>x=!!123</code>	00000000 00000000 00000000 00000000 1
<code>x<<31</code>	1 00000000 00000000 00000000 00000000
<code>(x<<31)>>31</code>	11111111 11111111 11111111 11111111
<code>!x</code>	00000000 00000000 00000000 00000000 0
<code>!x<<31</code>	0 00000000 00000000 00000000 00000000
<code>(!x<<31)>>31</code>	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);`