Data representation: Bits, Bytes, Integers

Yipeng Huang

Rutgers University

September 29, 2025

Table of contents

Important change to recitation schedule

Rationale

1. Section 6 and 8 recitations were scheduled at the same time in different rooms.
2. Students should feel free to attend any and all recitations and office hours for help with assignments.
3. Good to spread out opportunities for homework help across more days of the week.

The plan

1. Jedrik's recitation on Tuesdays 7:45 PM - 8:40 PM in Chemical Biology CCB-1303 (capacity 117) remains in place.
2. Zirui's recitation moves to Wednesday 5:55 PM - 6:50 PM in CoRE 301.
3. There is no more recitation in SEC-118. For help on Tuesday evenings, go to CCB-1303.

Canvas timed quiz 3 and programming assignment 1

Programming assignment 2

1. Due Friday 10/10.
2. Hash tables, graph algorithms, and recursion.

Quiz 3

1. Spanning Wednesday 9/25 - Wednesday 10/1.
2. 60 minutes.
3. Two tries.
4. Arrays, pointers, structs, and memory

Table of contents

Understanding pass-by-value and pass-by-reference

In this section, we study the `push()` function for a stack.

The `push()` function needs to make changes to the top of the stack, and return pointers to stack elements such that the elements can later be freed from memory.

We consider four function signatures for `push()` that are incorrect.

1. `void push ( char value, struct stack s );`
2. `void push ( char value, struct stack* s );`
3. `struct stack push ( char value, struct stack s );`
4. `struct stack push ( char value, struct stack* s );`

And we consider two function signatures for `push()` that are correct.

5. `void push ( char value, struct stack** s );`
6. `struct stack* push ( char value, struct stack* s );`

Understanding pass-by-value and pass-by-reference

```
1  void push ( char value, struct stack
       s ) { // bug in signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = &s;
6
7      s = *bracket;
8
9      return;
10 }
```

```
1 int main () {
2     struct stack s;
3     push( 'S', s );
4     printf ("s.data = %c\n", s.data)
          ;
5 }
```

Version 1. An incorrect function signature for `push()`.

This version of `push()` completely passes-by-value and has no effect on `struct stack s` in `main()`, so `s.data` is uninitialized.

Understanding pass-by-value and pass-by-reference

```
1  void push ( char value, struct stack
       * s ) { // bug in signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return;
10 }
```

```
1  int main () {
2      struct stack s;
3      push( 'S', &s );
4      push( 'C', &s );
5      // printf ("s = %p\n", s);
6      struct stack* pointer = &s;
7      printf ("pop: %c\n", pop(&
           pointer));
8      printf ("pop: %c\n", pop(&
           pointer));
9  }
```

Version 2. An incorrect function signature for `push()`.

This version of `push()` also has no effect on `struct stack s` in `main()`.

Understanding pass-by-value and pass-by-reference

```
1  struct stack push ( char value,
       struct stack s ) { // bug in
       signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = &s;
6
7      s = *bracket;
8
9      return s;
10 }
```

Version 3. An incorrect function signature for `push()`.

Here, we try returning an updated stack data structure via the return type of `push()`. Lines 3, 7, and 9 will lead to a memory leak (pointer is lost). Line 5 assigns the `next` pointer to an address `&s` which will be out of scope in `main()`.

Understanding pass-by-value and pass-by-reference

```
1  struct stack push ( char value,
       struct stack* s ) { // bug in
       signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return *s;
10 }
```

```
1 int main () {
2     struct stack s;
3     s = push( 'S', &s );
4     printf ("s.data = %c\n", s.data)
          ;
5     s = push( 'C', &s );
6     printf ("s.data = %c\n", s.data)
          ;
7 }
```

Version 4. An incorrect function signature for `push()`.

Here, we again try returning an updated stack data structure via the return type of `push()`. Lines 3, 7, and 9 will still lead to a memory leak (pointer is lost).

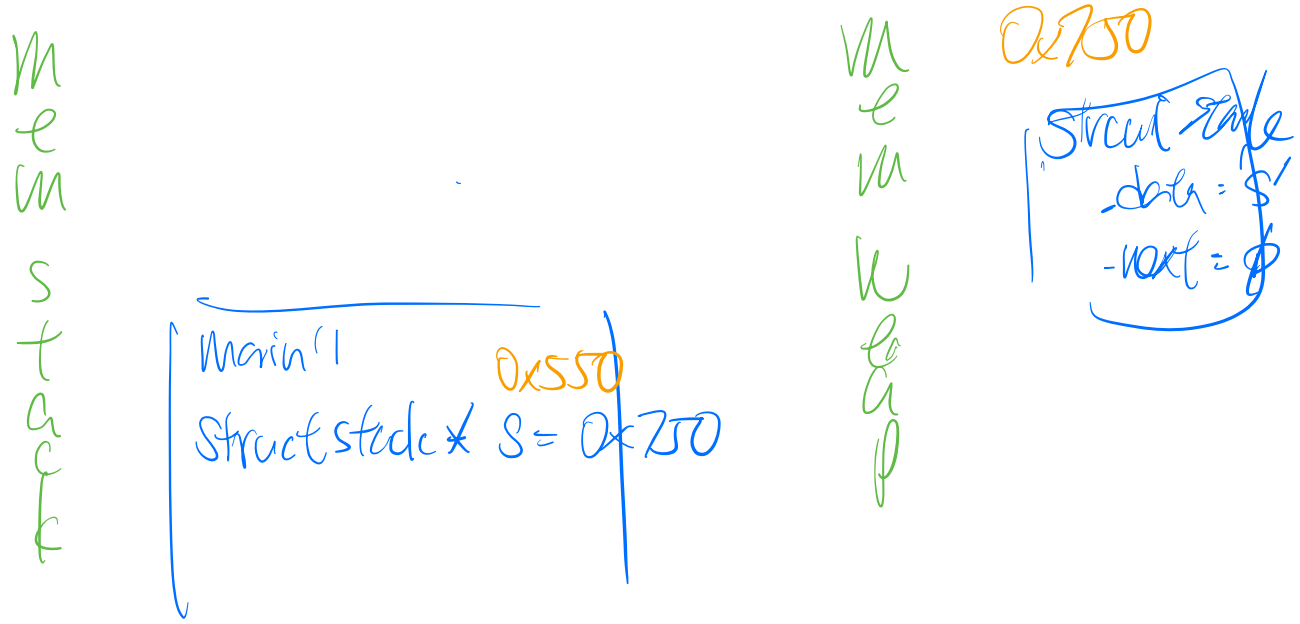
Understanding pass-by-value and pass-by-reference

```
1  void push ( char value, struct stack
       ** s ) {
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = *s;
6
7      *s = bracket;
8
9      return;
10 }
```

```
1  int main () {
2      struct stack* s;
3      push( 'S', &s );
4      push( 'C', &s );
5      printf ("pop: %c\n", pop(&s));
6      printf ("pop: %c\n", pop(&s));
7  }
```

Version 5. A correct function signature for `push()`.

`struct stack* s` in `main()` updates by passing the `struct stack *` parameter via pass-by-reference, leading to the `push()` signature that you see here. This matches the signature that you see for the `pop()` function.


mem stack
main()
0x550
struct stude * S = 0x250
mem heap
0x250
struct stude
-data = $
-next = ∅

Understanding pass-by-value and pass-by-reference

```
1  struct stack* push ( char value,
       struct stack* s ) {
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return s;
10 }
```

```
1 int main () {
2     struct stack* s;
3     s = push( 'S', s );
4     s = push( 'C', s );
5     printf ("pop: %c\n", pop(&s));
6     printf ("pop: %c\n", pop(&s));
7 }
```

Version 6. A correct function signature for `push()`.

`struct stack* s` updates via the return type of `push()` in `main()`, lines 3 and 4. Side note, `pop()` needs to return the character data, so `pop()` cannot have a similar function signature.

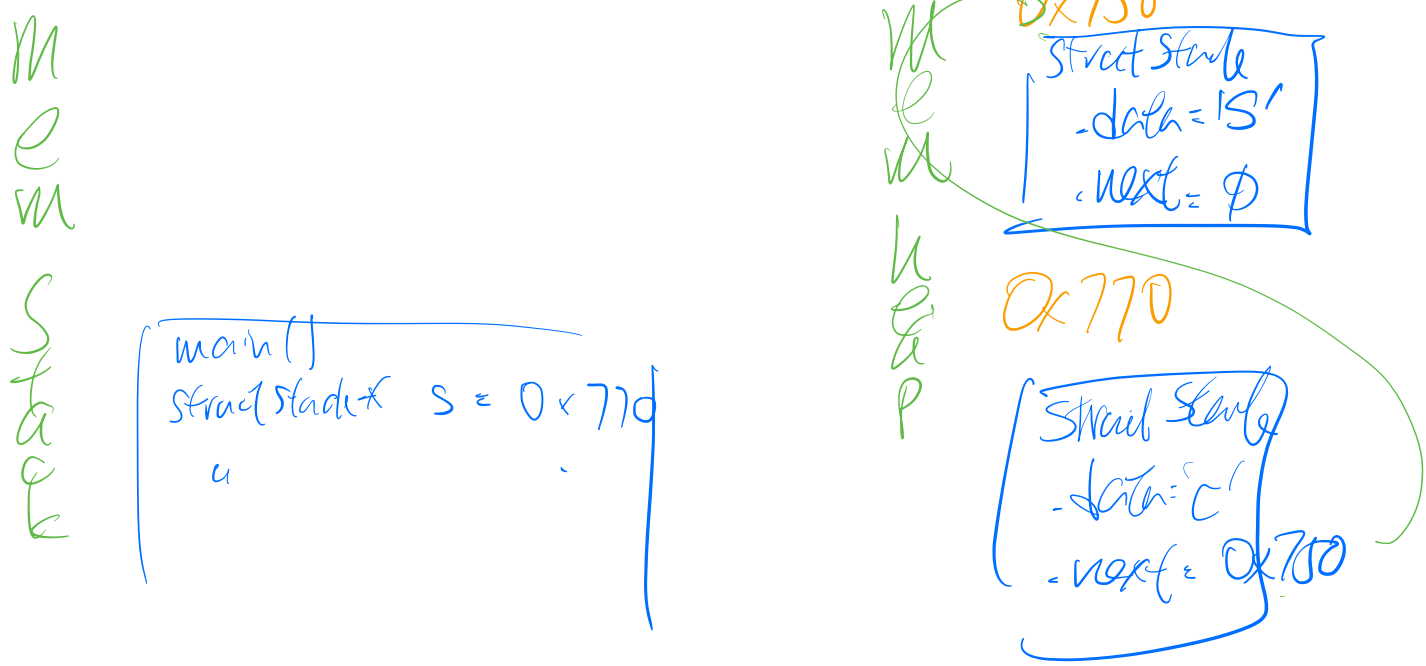

mem
Stack
main()
struct Stud* s = 0x770
mem
Heap
0x750
struct Stud
-data = 'S'
-next = ϕ
0x770
struct Stud
-data = 'C'
-next = 0x750

Table of contents

Failure to free

```c
#include <stdlib.h>
#include <stdio.h>

int main () {

    int* pointer0 = malloc(sizeof(int));
    *pointer0 = 100;
    printf("*pointer0 = %d\n", *pointer0);

}
```

Note: calloc() functions like malloc(), but calloc() initializes memory to zero while malloc() offers no such guarantee.

Memory leaks

Have you ever had to restart software or hardware to recover it?

Debug by compilation in GCC, running with Valgrind, Address Sanitizer

Use after free

```
1    int* pointer0 = malloc(sizeof(int));
2
3    printf("pointer0 = %p\n", pointer0);
4    *pointer0 = 100;
5    printf("*pointer0 = %d\n", *pointer0);
6
7    free(pointer0);
8    pointer0 = NULL;
9
10   printf("pointer0 = %p\n", pointer0);
11   *pointer0 = 10;
12   printf("*pointer0 = %d\n", *pointer0);
```

Dangling pointers

- One defensive programming style is to set any freed pointer to NULL.
- Debug by running with Valgrind, Address Sanitizer.

Pointer aliasing

```
1     int* pointer0 = malloc(sizeof(int));
2     int* pointer1 = pointer0;
3
4     *pointer0 = 100;
5     printf("*pointer1 = %d\n", *pointer1);
6
7     *pointer0 = 10;
8     printf("*pointer1 = %d\n", *pointer1);
9
10    free(pointer0);
11    pointer0 = NULL;
12
13    *pointer1 = 1;
14    printf("*pointer1 = %d\n", *pointer1);
```

Debug by running with Valgrind, Address Sanitizer

Pointer aliasing and overhead of garbage collection

- Java garbage collection tracks dangling pointers but costs performance.
- C requires programmer to manage pointers but is more difficult.

Pointer typing

```
1     unsigned char n = 2;
2     unsigned char m = 3;
3
4     unsigned char ** p;
5     p = calloc( n, sizeof(unsigned char) );
6
7     for (int i = 0; i < n; i++)
8         p[i] = calloc( m, sizeof(unsigned char) );
9
10    for (int i = 0; i < n; i++)
11        for (int j = 0; j < m; j++) {
12            p[i][j] = 10*i+j;
13            printf("p[%d][%d] = %d\n", i, j, p[i][j]);
14        }
```

Defend using explicit pointer casting.

Table of contents

Non existent memory

```c
#include <stdlib.h>
#include <stdio.h>

int main () {

  int **x = malloc(sizeof(int*));
  **x = 8;
  printf("x = %p\n", x);
  printf("*x = %p\n", *x);
  printf("**x = %d\n", **x);
  fflush(stdout);

}
```

Debug by running with Valgrind, Address Sanitizer

Returning null pointer

```
1
2  int* returnsNull () {
3    int val = 100;
4    return &val;
5  }
6
7  int main () {
8
9    int* pointer = returnsNull();
10   printf("pointer = %p\n", pointer);
11   printf("*pointer = %d\n", *pointer);
12
13 }
```

Prevent using -Werror compilation flag.

Table of contents

Future computer architectures: A quantum pointer?

For $n = 1$, four possibilities

	f_0	f_1	f_2	f_3
f(0)	0	0	1	1
f(1)	0	1	0	1
	f is constant 0	f is balanced	f is balanced	f is constant 1

A simple physics experiment that classical computing cannot replicate

Algorithm

David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. 1992.

Implementation

- Mach-Zehnder interferometer implementation.
  `https://www.st-andrews.ac.uk/physics/quvis/simulations_html5/sims/SinglePhotonLab/SinglePhotonLab.html`

Mathematical description of the algorithm

$$|0\rangle \xrightarrow{H} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{cases} \xrightarrow{I} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |0\rangle \\ \xrightarrow{Z} |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |1\rangle \\ \xrightarrow{-Z} -|-\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|1\rangle \\ \xrightarrow{-ZZ=-I} -|+\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|0\rangle \end{cases}$$

The binary abstraction: classical and quantum

High, low voltage

Adds resilience against noise.

Representation as a state vector

- $\begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$ We pronounce this "ket" 0
- $\begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$ We pronounce this "ket" 1

The Hadamard gate

Matrix representation of Hadamard operator: $H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$

- $H|0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$
- $H|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$

Superposition

Single qubit state

- $\alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$
- Amplitudes $\alpha, \beta \in \mathbb{C}$
- $|\alpha|^2 + |\beta|^2 = 1$
- The above constraints require that qubit operators are unitary matrices.

Many physical phenomena can be in superposition and encode qubits

- Polarization of light in different directions
- Electron spins (Intel solid state qubits)
- Atom energy states (UMD, IonQ ion trap qubits)
- Quantized voltage and current (IBM, Google superconducting qubits)

If multiple discrete values are possible (e.g., atom energy states, voltage and current), we pick (bottom) two for the binary abstraction.

Mathematical description of the algorithm

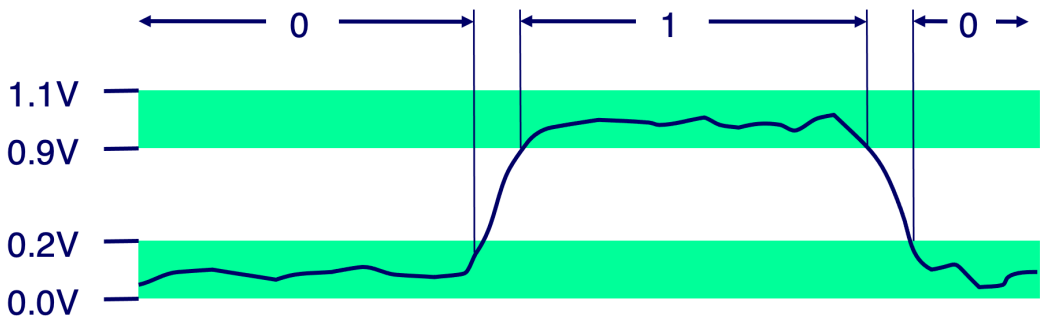
$$|0\rangle \xrightarrow{H} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \begin{cases} \xrightarrow{I} |+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |0\rangle \\ \xrightarrow{Z} |-\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} |1\rangle \\ \xrightarrow{-Z} -|-\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|1\rangle \\ \xrightarrow{-ZZ=-I} -|+\rangle = \begin{bmatrix} \frac{-1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} & \xrightarrow{H} -|0\rangle \end{cases}$$

Table of contents

Why binary

Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



Decimal, binary, octal, and hexadecimal

Decimal	Binary	Octal	Hexadecimal
0	0b0000	0o0	0x0
1	0b0001	0o1	0x1
2	0b0010	0o2	0x2
3	0b0011	0o3	0x3
4	0b0100	0o4	0x4
5	0b0101	0o5	0x5
6	0b0110	0o6	0x6
7	0b0111	0o7	0x7

Decimal	Binary	Octal	Hexadecimal
8	0b1000	0o10	0x8
9	0b1001	0o11	0x9
10	0b1010	0o12	0xA
11	0b1011	0o13	0xB
12	0b1100	0o14	0xC
13	0b1101	0o15	0xD
14	0b1110	0o16	0xE
15	0b1111	0o17	0xF

In C, format specifiers for printf() and fscanf():

1. decimal: '%d'
2. binary: none
3. octal: '%o'
4. hexadecimal: '%x'

Decimal, binary, octal, and hexadecimal

How to represent the range of unsigned char in each?

Unsigned char is one byte, 8 bits.

1. decimal: 0 to 255
2. binary: 0b0 to 0b11111111
3. octal: 0 to 0o377 (group by 3 bits)
4. hexadecimal: 0x00 to 0xFF (group by 4 bits)

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

#000000	#FFFFFF	#6A757C	??? #CC0033

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

#000000	#FFFFFF	#6A757C	#CC0033

Representing characters

- char is a 1-byte, 8-bit data type.
- ASCII is a 7-bit encoding standard.
- "man ascii" to see Linux manual.
- Compile and run ascii.c to see it in action.
- Some interesting characters: 7 (bell), 10 (new line), 27 (escape).

USASCII code chart

$b_7 b_6 b_5$ → / Bits / $b_4 b_3 b_2 b_1$ ↓ / Row ↓ Column →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Figure: ASCII character set. Image credit Wikimedia

Bitwise operations

Why are bitwise operations important?

- Network and UNIX settings using bit masks (e.g., umask)
- Hardware and microcontroller programming (e.g., Arduinos)
- Instruction set architecture encodings (e.g., ARM, x86)

Bitwise operations

~: bitwise NOT

unsigned char a = 128

$$
\begin{aligned}
a &= 0b1000\_0000 \\
\sim a &= \sim 0b1000\_0000 \\
&= 0b0111\_1111 \\
&= 127
\end{aligned}
$$

b	~ b
0	1
1	0

Bitwise operations

&: bitwise AND

$$\begin{aligned} 3\&1 &= 0b11\&0b01 \\ &= 0b01 \\ &= 1 \end{aligned}$$

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise operations

| : bitwise OR

$$\begin{aligned} 3|1 &= 0b11|0b01 \\ &= 0b11 \\ &= 3 \end{aligned}$$

$$\begin{aligned} 2|1 &= 0b10|0b01 \\ &= 0b11 \\ &= 3 \end{aligned}$$

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise operations

^: bitwise XOR

$$\begin{aligned} 3 \wedge 1 &= 0b11 \wedge 0b01 \\ &= 0b10 \\ &= 2 \end{aligned}$$

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

`inplaceSwap.c`: Swapping variables without temp variables.

How does it work?

Don't confuse bitwise operators with logical operators

Bitwise operators

- ~
- &
- |
- ^

Logical operators

- !
- &&
- ||
- != (for bool type)