C Programming: Memory programming patterns and antipatterns

Yipeng Huang

Rutgers University

October 2, 2026

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

C Programming: Memory programming patterns and antipatterns

Announcements

- Canvas timed quiz 4 and programming assignment 2

Stack data structure: `struct`, `push()`, `pop()`

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

Bugs and debugging related pointers, malloc, free

- Failure to free
- Use after free
- Pointer aliasing
- Pointer typing

Bugs and debugging related C memory model

- Non existent memory
- Returning null pointer

Bits and bytes

- Why binary
- Decimal, binary, octal, and hexadecimal
- Representing characters
- Bitwise operations

Canvas timed quiz 4 and programming assignment 2

Quiz 4

1. Spanning Friday 10/2 - Friday 10/16.
2. 60 minutes.
3. Three tries.
4. Memory bugs and bits, bytes, and integers.

Programming assignment 2

1. Due Tuesday 10/13.
2. Data structures mastery.

Programming Patterns vs. Anti-Patterns

- **Programming Patterns:** Recurring structural symmetries in code that signal readability, maintainability, and correctness.
- **Anti-Patterns:** Coding practices that introduce subtle bugs, memory leaks, or unmaintainable logic.
- **The Memory Ownership Pattern:**
  - Whoever allocates heap memory must be responsible for freeing it.
  - Prevents "ownership confusion" across function boundaries.
- **Refactoring for Symmetry:** Extracting duplicated code (e.g., repeating file-reading logic for two matrices) into a single reusable helper function.

Software Engineering in the Age of AI

- **The Shifting Role of Engineers:** As AI tools generate code syntax, the primary responsibilities shift from *writing* to *specifying* and *reviewing*.
- **Precise Specification:** Defining exact preconditions, postconditions, and edge-case behaviors before code is generated.
- **Code Review Methodologies:**
  - **Unit Testing & Randomized Validation:** Testing functions against random input generators.
  - **Verification:** Proving mathematically that preconditions entail postconditions.
  - **Memory Checking:** Verifying zero leaks or dangling pointers.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

struct

arrays vs structs

- Arrays group data of the same type. The `[]` operator accesses array elements.
- Structs group data of different type. The `.` operator accesses struct elements.

These are equivalent; the latter is shorthand:

```
BSTNode* root;
```

- `(*root).key = key;`
- `root->key = key;`

When structs are passed to functions, they are passed BY VALUE.

Syntactic Sugar: Structs vs. Classes

- **Structs vs. Java Classes:**
  - C `struct`s pack fields contiguously in memory, but contain no methods/member functions.
  - Java classes extend this layout by bundling function scopes within the data structure.
- **Arrow Notation (–>) as Syntactic Sugar:**
  - `bracket->data` is shorthand for `(*bracket).data`.
  - Dereferences the pointer and accesses the member field in a single, readable operator.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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.

Pointer Indirection: Why `struct stack**` is Required

- **The Golden Rule:** In C, everything is passed by value. To modify a caller's variable, you must pass a pointer to it.
- **Updating a Pointer Variable:**
  - `push(..., struct stack s)`: Modifies local copy; loses heap address → **Memory Leak**.
  - `push(..., struct stack* s)`: Modifies local pointer copy; caller's pointer unchanged.
  - `push(..., struct stack** s)`: Passes pointer to caller's pointer → **Correctly updates caller's root pointer**.
- *"Any problem in computer science can be solved by another layer of indirection."*

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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.

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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.

C Programming: Memory programming patterns and antipatterns

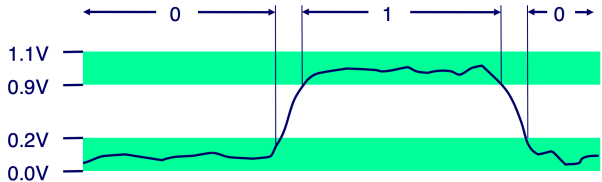
- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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



C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

Decimal, binary, octal, and hexadecimal

Decimal	Binary	Octal	Hexadecimal	Decimal	Binary	Octal	Hexadecimal
0	0b0000	000	0x0	8	0b1000	0010	0x8
1	0b0001	001	0x1	9	0b1001	0011	0x9
2	0b0010	002	0x2	10	0b1010	0012	0xA
3	0b0011	003	0x3	11	0b1011	0013	0xB
4	0b0100	004	0x4	12	0b1100	0014	0xC
5	0b0101	005	0x5	13	0b1101	0015	0xD
6	0b0110	006	0x6	14	0b1110	0016	0xE
7	0b0111	007	0x7	15	0b1111	0017	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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row ↓ Column →	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

C Programming: Memory programming patterns and antipatterns

- Announcements
  - Canvas timed quiz 4 and programming assignment 2
- Stack data structure: `struct`, `push()`, `pop()`
- Understanding pass-by-value and pass-by-reference
- Bugs and debugging related pointers, malloc, free
  - Failure to free
  - Use after free
  - Pointer aliasing
  - Pointer typing
- Bugs and debugging related C memory model
  - Non existent memory
  - Returning null pointer
- Bits and bytes
  - Why binary
  - Decimal, binary, octal, and hexadecimal
  - Representing characters
  - Bitwise operations

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

$$
\begin{aligned}
a &= 0b1000_0000 \\
\tilde{}a &= \tilde{}0b1000_0000 \\
&= 0b0111_1111 \\
&= 127
\end{aligned}
$$

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

ˆ: bitwise XOR

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

a	b	a ˆ 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)