



C Integers in Binary

```
010100
001001
010100
```

- ❑ C offers integers in two flavors: **signed & unsigned**
- ❑ Signed integers use the **two's complements** system
 - ➔ **Positive numbers:** just their binary representation
 - ➔ **Negative numbers:** (1) Take the binary of the positive equivalent, (2) invert each bit, (3) and add 1
 - ➔ 0 is all zero bits 🥰

Note that **most significant bit** indicates sign (0 for +, 1 for -)

(1) 0101 +5
(2) 1010
(3) 1011 -5

- ❑ When casting from signed to unsigned, the underlying bits don't change – just the way they're interpreted (two's complement ↔ ordinary binary)
- ❑ When comparing signed and unsigned ints, C implicitly casts signed to unsigned

Signed	-1	11111111	0xff
Unsigned	255	11111111	0xff

Binary and Bit Vectors

Goal: To think about binary representations as a kind of **array data structure** – with each bit storing true/false information

Example: In *Flappy Bird*, we can represent a vertical barrier as a **64-bit number / vector**: **0 bit** for opening, **1 bit** otherwise

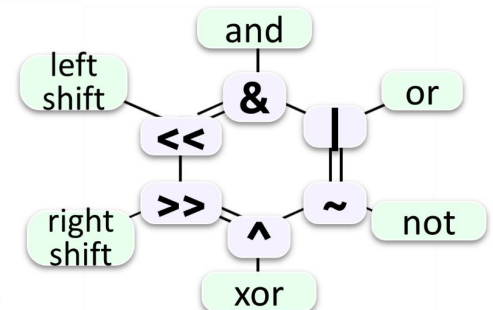
```
1 1 1 0 ... 0 1 1 1
long barrier = 0b1110...0111
```

Task: How to **modify / access** the data within?

👉 **6 key bitwise operators** — shown on the right!
🧠 Bitmasks are useful for **manipulating bits** (e.g., getting the lowest bit, flipping a specific bit OFF)

```
1 1 1 0 ... 0 1 0 1
```

Visually, this would be like opening a new hole in the barrier



Bitwise Operators

- ❑ AND, OR, XOR are binary operators (2 bits as input), NOT is unary (1 bit as input)
- ❑ When applied on numbers (multiple bits), **applied to corresponding bits** in each

AND	OR	XOR	NOT
<pre>1 1 0 0 & 1 0 1 0 ----- 1 0 0 0</pre>	<pre>1 1 0 0 1 0 1 0 ----- 1 1 1 0</pre>	<pre>1 1 0 0 ^ 1 0 1 0 ----- 0 1 1 0</pre>	<pre>~ 1 1 0 0 ----- 0 0 1 1</pre>

- ➔ The **AND** of two bits is **1** only if **both bits are 1**, and otherwise **0**.
- ➔ The **OR** of two bits is **1** if **either bit (or both) is a 1**, and otherwise **0**.

- ➔ The **XOR** (exclusive OR) of two bits is **1** if **exactly one of the bits is a 1**, and **0** if both bits or neither bit is a **1**.
- ➔ The **NOT** of a bit is its **exact opposite**: **1** if it's a **0**, and **0** if the bit's a **1**.



More Bit Operators (Shifting)



Left shifting (<<) involves sliding a bit pattern a **certain number of positions to the left** — filling in the new lower bits with 0s, and discarding any bits shifted off the end.

Left Shift
(fill with 0s)

$1 \ll 3 \rightarrow 1000$
3 new zero bits

$11 \ll 30 \rightarrow 11000...000$
30 new zero bits

Right shifting (>>) is similar, but slides to the right instead. However, **filling in the higher-order bits is trickier**, as we don't want to alter the sign bit / MSB.

Solution: For signed values, fill with **sign bit**; for unsigned, just fill with leading 0s.

Right Shift
num

Unsigned Values

1 0 ... 0 1 0

num >> 1

0 1 0 ... 0 1

Logical (fill with **leading 0s**)

Signed Values

0 0 ... 0 1 0

1 0 ... 0 1 0

0 0 0 ... 0 1

1 1 0 ... 0 1

Arithmetic (fill with **sign bit**)

Bitmasks to Modify Bits Strategically



To **manipulate bits in a certain way**, we often construct a help value / **bitmask**, e.g.,

Turn a bit on $\text{bit} \mid 1$

Invert/flip a bit $\text{bit} \wedge 1$

Turn a bit off $\text{bit} \& 0$

Retrieving the last bit $\text{num} \& 1$

To see how a bitmask works, I recommend trying it with **concrete examples**, e.g.,

Task: Getting the lowest bit (LSB)
Bitmask: AND (&) with 1

num 1 ... 0 1 1 0 1 1
bitmask & 0 ... 0 0 0 0 0 1
result 0 ... 0 0 0 0 0 1

Note that all bits are **switched off**, except for the **last one!** (preserved from original **num**)

Task: Turning on the lowest bit
Bitmask: OR (|) with 1

num 1 ... 0 1 0 0 0
bitmask 0 ... 0 0 0 0 1
result 1 ... 0 1 0 0 1

Note that all bits are **retained**, except the **lowest bit (LSB)** we just turned on!

Building Bitmasks



Bitmask Design

For **building bitmasks**, I recommend:

- ❖ Breaking it up in separate steps
- ❖ Constructing it **from bitmask(s) you already know** (e.g., -1 is all 1s)

👉 On the right, some useful ones to know!

Task: Turning off the lowest 4 bits simultaneously

num 1 ... 0 1 1 0 1 1
bitmask & 1 ... 1 1 0 0 0 0
result 1 ... 0 1 0 0 0 0

So a possible bitmask is AND (&) with $(-1L \ll 4)$

Bitmask building blocks / LEGOs

-1 1 1 1 ... 1 1 1 -1

1 0 ... 0 0 0 0 1 1

Note: 1L for signed long, 1UL for unsigned long

2^n 0 ... 0 1 0 0 0 $1 \ll n$

$2^n - 1$ 0 ... 0 0 1 1 1 $(1 \ll n) - 1$

How do we build this mask?

1 ... 1 1 1 1 1 1

Start with -1L (above) and shift 4 bits to the left!

1 ... 1 1 0 0 0 0

Alternatively, the bitmask from prior slide!