

# Operatori bitwise

Matteo Spanio

Giulio Pitteri

17 aprile 2025

In questa lezione si approfondisce la conoscenza degli operatori bitwise in C.

## Operazioni bitwise

Gli operatori bitwise permettono di effettuare calcoli al livello dei bit delle variabili. Questi operatori sono molto utili quando si deve lavorare con i registri di un microcontrollore o con i dati grezzi provenienti da un sensore.

Gli operatori bitwise in C sono: - & AND - | OR - ^ XOR - ~ NOT - << Shift a sinistra - >> Shift a destra

## Tabelle di verità

Questi operatori applicano le regole dell'algebra booleana, pertanto sarà importante conoscere le tabelle di verità delle operazioni AND, OR e XOR.

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

## Rappresentazione in base 2 in C

In elettronica digitale, spesso si utilizzano serie di bit per rappresentare lo stato di un sistema. Ad esempio, si può utilizzare un byte per rappresentare lo stato di 8 interruttori. Supponiamo che l'interruttore 3 sia acceso e gli altri spenti. Per rappresentare lo stato degli interruttori si può utilizzare una variabile di tipo `char`:

```
unsigned char switches = 0b00001000;
// oppure
unsigned char switches = 8;
```

In questa configurazione il quarto bit interruttore è acceso, mentre gli altri sono spenti.

Ipotizzando di voler accendere un altro interruttore, si può utilizzare l'operatore OR:

```
// switches = 0b00001000
switches = switches | 0b00000100;
// oppure
switches = switches | 4;
```

Ora il valore di `switches` sarà `0b00001100`. Il che rappresenta il fatto che gli interruttori 3 e 4 sono accesi.

Per spegnere tutti gli interruttori si può utilizzare l'operatore AND:

```
switches = switches & 0b00000000;
// oppure
switches = switches & 0;
// switches: 0b00000000
```

## Esempio di esercizio

Scrivere un programma che stampi a monitor il valore del bit meno significativo di un numero intero.

```
#include <stdio.h>
void print_lsb(int n);

int main(void)
{
    int a = 127, b = 128;

    print_lsb(a);
    print_lsb(b);
}

void print_lsb(int n)
{
    printf("The least significant bit of %d is: ", n);
    printf("%d\n", n & 1);
}
```

## XOR

L'operatore XOR è molto utile per invertire lo stato di un bit.

### **i** Nota

Eseguire l'operazione di XOR due volte su un bit restituisce il valore originale.

```
char c = 'A';  
c = c ^ 'h';  
c = c ^ 'h';  
// c: 'A'
```

### Esempio

Vediamo ora un esempio di aritmetica con XOR: volendo fare encoding e decoding di un carattere, possiamo utilizzare l'operatore XOR. Supponiamo di voler cifrare il carattere 'A' utilizzando la chiave 'I'. La chiave è un carattere ASCII, quindi possiamo rappresentarla in binario:

A -> 65: 0b01000001

I -> 73: 0b01001001

08: 0b00001000

$65 \wedge 73 = 8$

$8 \wedge 73 = 65$

'A'  $\wedge$  'I' = '\b'

Ora possiamo implementare un semplice programma di crittografia in C.

```
#include <stdio.h>  
#define KEY 'h'  
  
int main(void)  
{  
    char c = 'A';  
    c = c ^ KEY;  
    printf("%c\n", c);  
  
    c = c ^ KEY;  
    printf("%c\n", c);  
}
```

## Shift

Gli operatori di shift permettono di spostare i bit di una variabile a sinistra o a destra.

```
unsigned char c = 0b00000001;
c = c << 1;
// c: 0b00000010
c = c << 3;
// c: 0b00010000
c = c >> 2;
// c: 0b00000100
```

## Esempio Shift

Scrivere un programma che moltiplichi un numero intero per 2 utilizzando l'operatore di shift.

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    n = n << 1;
    printf("%d\n", n);
}
```

## Double Linked List

Abbiamo visto le liste concatenate in precedenza, uno dei limiti di questa struttura dati è che non permette di navigare la lista in entrambe le direzioni. Per superare questo limite, è possibile implementare una lista doppiamente concatenata, in cui ogni nodo contiene un puntatore al nodo successivo e un puntatore al nodo precedente. In questo modo è possibile navigare la lista in entrambe le direzioni.

```
typedef struct node {
    struct node *prev;
    struct node *next;
    int data;
} Node;
```

Ovviamente questo approccio ha un costo in termini di memoria, poiché ogni nodo deve memorizzare due puntatori invece di uno.

## XOR Linked List

Nei sistemi con poca disponibilità di memoria, è possibile implementare una lista doppiamente concatenata utilizzando l'operatore XOR. In questo modo si può risparmiare memoria, poiché non è necessario memorizzare i puntatori `next` e `prev` per ogni nodo.

Ogni nodo della lista contiene un campo `link` che è il risultato dell'operazione XOR tra i puntatori `next` e `prev`. Per calcolare il puntatore `next` o `prev` di un nodo, è sufficiente eseguire l'operazione XOR tra il puntatore `link` e il puntatore del nodo corrente.

```
typedef struct node {
    struct node *link;
    int data;
} Node;
```

## Esercizi

1. Scrivere una funzione che dato un numero intero  $n$ , restituisca `true` se  $n$  è pari, `false` altrimenti. Usare l'operatore AND per verificare se un numero è pari.

```
#include <stdio.h>
#include <stdbool.h>
bool is_even(int n);

int main(void) {
    int a = 55, b = 48;

    printf("%d is even: %d\n", a, is_even(a));
    printf("%d is even: %d\n", b, is_even(b));
}

bool is_even(int n) {
    // TODO
}
```

2. Scrivere un programma di semplice crittografia in grado di cifrare e decifrare una stringa utilizzando l'operatore XOR.
3. Scrivere un programma che stampi a monitor la codifica in binario di un unsigned char. Ad esempio, se il valore di `c` è 5, il programma dovrà stampare 00000000 00000000 00000000 00000101.
4. Immaginando che una serie di 8 bit rappresenti lo stato di accensione di 8 led, scrivere un programma che:
  1. Accenda il led più a destra;
  2. Accenda il led più a sinistra;

3. Inverta lo stato di tutti i led;
4. Spenga tutti i led.