

Operatori bitwise

Liceo G.B. Brocchi
Classi prime Scientifico - opzione scienze applicate
Bassano del Grappa, Aprile 2023

Gli operatori *bitwise*

- Il C mette a disposizione 6 operatori per la manipolazione dei bit, applicabili su operandi *integral*, ossia su valori di tipo:

int
char
short
long

- Gli operatori in questione sono:

&	bitwise AND (da non confondere con &&)
	bitwise inclusive OR (da non confondere con)
^	bitwise exclusive OR (XOR/EXOR)
<<	left shift
>>	right shift
~	complemento a uno (<i>one's complement</i>)

AND bitwise


- L'AND bitwise non è altro che l'*and* bit a bit di due stringhe binarie
- Con questi operatori bitwise si utilizza spesso il termine *maschera* (*mask*): una maschera è una stringa binaria che serve per nascondere o estrarre alcuni bit da un'altra stringa binaria
- Ad esempio, per estrarre i 4 bit più significativi da una stringa binaria A di 8 bit, e azzerare i 4 bit meno significativi, si può utilizzare l'operatore bitwise &, utilizzando come maschera 1111 0000
- Supponendo di operare su 8 bit (1 byte), l'operazione è questa:

```
0 0 1 0 0 1 1 1
      &
1 1 1 1 0 0 0 0
      =
0 0 1 0 0 0 0 0
```

AND bitwise

A: 0 0 1 0 0 1 1 1

 &
MASK: 1 1 1 1 0 0 0 0
 =
 0 0 1 0 0 0 0 0



notate come abbiamo *attivato* i 4 bit più significativi, e *disattivato* i 4 meno significativi

AND bitwise (a & b)

- In C non possiamo scrivere direttamente costanti binarie. Possiamo però utilizzare costanti esadecimali, che sono comodamente mappabili su stringhe binarie
- Ad ogni cifra esadecimale corrisponde infatti un *nibble*, ossia una sequenza di 4 bit che rappresenta la conversione in binario della cifra esadecimale in questione
- Quindi,
 - a = 0010 0111 diventa 0x27, perché 0010→0x2 e 0111→0x7
 - b = 1111 0000 diventa 0xF0

il risultato di a & b è 0x20, ossia 0010 0000

OR inclusivo bitwise

- L'OR inclusivo bitwise può essere utilizzato per *attivare*, ossia per settare a 1 alcuni bit di una stringa binaria

0x	0	0	0	A	B	C	D	E
0x	F	F	F	0	0	0	0	0
			=					
0x	F	F	F	A	B	C	D	E

L'OR inclusivo bitwise (a | b)

- L'OR inclusivo bitwise può essere utilizzato per *attivare*, ossia per settare a 1 alcuni bit di una stringa binaria

a in 8-digit hexadecimal and 32-bit binary is:	0x000abcde	0000 0000 0000 1010 1011 1100 1101 1110
b in 8-digit hexadecimal and 32-bit binary is:	0xffff0000	1111 1111 1111 0000 0000 0000 0000 0000
a b in 8-digit hexadecimal and 32-bit binary is:	0xffffabcde	1111 1111 1111 1010 1011 1100 1101 1110

L'OR inclusivo è un *attivatore* di bit, proprio per la natura dell'OR. Se un bit della maschera vale 1, il risultato corrispondente a quel bit è sicuramente 1,

L'AND invece è un *selezionatore*. Se un bit della maschera vale 0, allora sicuramente il risultato corrispondente a quel bit è 0. Se invece è 1, il risultato è uguale al bit della stringa di partenza

Il *left shift* ($a \ll b$)

- Il left shift di **b** posizioni di una stringa binaria **a** consiste nello spostare a sinistra di **b** posizioni i bit di **a**
- Dovete immaginare che per ogni spostamento a sinistra, in corrispondenza del bit meno significativo (più a destra) resta una buca, che viene riempito con 0. Il bit più significativo, invece, «cade»
- $0010 \ll 1 = 0100$
- $00001100 \ll 2 = 00110000$
- Shiftare a sinistra di n posizioni equivale a moltiplicare per 2^n
- La questione è del tutto analoga in base 10, solo che qui lo shift a sinistra equivale ad una moltiplicazione per 10^n

$$000543 \ll 3 = 543000$$

|| *left shift* (a << b)

a in 8-digit hexadecimal and 32-bit binary is:	0x0000001f	0000 0000 0000 0000 0000 0000 0001 1111
a << 2 in 8-digit hexadecimal and 32-bit binary is:	0x0000007c	0000 0000 0000 0000 0000 0000 0111 1100
a in decimal is:	31	
a << 2 in decimal is:	124 (31*2^2)	

a in 8-digit hexadecimal and 32-bit binary is:	0x00000008	0000 0000 0000 0000 0000 0000 0000 1000
a << 4 in 8-digit hexadecimal and 32-bit binary is:	0x00000080	0000 0000 0000 0000 0000 0000 1000 0000
a in decimal is:	8	
a << 4 in decimal is:	128 (8 * 2^4)	

a in 8-digit hexadecimal and 32-bit binary is:	0x00000064	0000 0000 0000 0000 0000 0000 0110 0100
a << 5 in 8-digit hexadecimal and 32-bit binary is:	0x00000c80	0000 0000 0000 0000 0000 1100 1000 0000
a in decimal is:	100	
a << 5 in decimal is:	3200 (100 * 2^5)	

a in 8-digit hexadecimal and 32-bit binary is:	0x0fffffff	0000 1111 1111 1111 1111 1111 1111 1111
a << 5 in 8-digit hexadecimal and 32-bit binary is:	0x1fffffff	0001 1111 1111 1111 1111 1111 1111 1110
a in decimal is:	268435455	
a << 1 in decimal is:	536870910 (268435455 * 2^1)	

Il *right shift* ($a \gg b$)

- Il right shift di **b** posizioni di una stringa binaria **a** consiste nello spostare a destra di **b** posizioni i bit di **a**
- Dovete immaginare che per ogni spostamento a destra, in corrispondenza del bit più significativo (più a sinistra) resta una buca, che viene riempita con 0. Il bit meno significativo, invece, «cade»
- $0010 \gg 1 = 0001$
- $00001100 \gg 2 = 00000011$
- Shiftare a destra di n posizioni equivale a dividere per 2^n . Il resto della divisione «cade» a destra
- La questione è del tutto analoga in base 10, solo che qui lo shift a destra equivale ad una divisione per 10^n
$$000543 \gg 2 = 000005 \text{ (resto 43)}$$

Il complemento a uno ($\sim a$)

- L'operatore di **complemento a uno** di una stringa binaria **a** effettua la complementazione a 1 di ciascun bit di a, ossia, inverte ciascun bit (*gli 0 diventano 1 e gli 1 diventano 0*)
- $\sim 0010 = 1101$
- $\sim 00001100 = 11110011$
- Se effettuate la somma binaria di a e $\sim a$, ad esempio su 8 bit, il risultato sarà: 11111111

Il complemento a uno ($\sim a$)

a in 8-digit hexadecimal and 32-bit binary is:	0x0000cafe	0000 0000 0000 0000 1100 1010 1111 1110
$\sim a$ in 8-digit hexadecimal and 32-bit binary is:	0xffff3501	1111 1111 1111 1111 0011 0101 0000 0001

a in 8-digit hexadecimal and 32-bit binary is:	0xffffffff	1111 1111 1111 1111 1111 1111 1111 1111
$\sim a$ in 8-digit hexadecimal and 32-bit binary is:	0x00000000	0000 0000 0000 0000 0000 0000 0000 0000

a in 8-digit hexadecimal and 32-bit binary is:	0xffff0000	1111 1111 1111 1111 0000 0000 0000 0000
$\sim a$ in 8-digit hexadecimal and 32-bit binary is:	0x0000ffff	0000 0000 0000 0000 1111 1111 1111 1111

a in 8-digit hexadecimal and 32-bit binary is:	0xdeadbeef	1101 1110 1010 1101 1011 1110 1110 1111
$\sim a$ in 8-digit hexadecimal and 32-bit binary is:	0x21524110	0010 0001 0101 0010 0100 0001 0001 0000