

Unità didattica: Operatori logici e operatori bitwise

[01-AC]

Titolo: Operatori logici e operatori bitwise in C

Argomenti trattati:

- ✓ Tipo logico (o booleano) in C
- ✓ Tavole di definizione degli operatori logici in C
- ✓ Operatori bitwise in C
- ✓ Differenza tra operatori logici e bitwise

Prerequisiti richiesti: **fondamenti del linguaggio C**

Tipo di dato

=

- **criterio** di rappresentazione in memoria dei dati
- **definizione** delle operazioni consentite sui dati del tipo

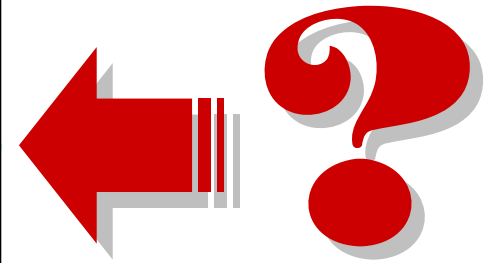
Tipi di dati scalari predefiniti

- Tipo logico o booleano
- Tipi carattere e stringa
- Tipi numerici (intero e reale)

In **memoria** tutte le *informazioni* (*istruzioni* e *dati*) sono codificate in **binario** (*sistema di numerazione in base 2*)

memoria

...	1	1	0	0	0	1	0	1	...
...	1	0	0	1	0	0	0	1	...
...	0	0	0	0	1	1	1	1	...
...	1	1	1	1	0	1	1	1	...



Cosa c'è in questi byte?

**È importante conoscere la
rappresentazione binaria**

provare...

```
...  
void main()  
{  
    printf("sizeof(char) = %d\n", sizeof(char));  
    printf("sizeof(short) = %d\n", sizeof(short));  
    printf("sizeof(long) = %d\n", sizeof(long));  
    printf("sizeof(char*) = %d\n", sizeof(char*));  
}
```

output

```
sizeof(char) = 1  
sizeof(short) = 2  
sizeof(long) = 4  
sizeof(type*) = 4
```

lunghezza in byte
1 byte = 8 bit

Tipo "puntatore"

perché?

Operatori logici in C (logical operators)

Agiscono su operandi di tipo logico (valori={true, false}={vero, falso}).

In C non esiste il tipo predefinito logico (o booleano),

ma ad ogni variabile è associato un valore di verità come segue:

- ❑ il valore **0** corrisponde a falso (**F**);
- ❑ un valore **≠0** (o **1** se risultato) corrisponde a vero (**V**).

op. in C	OPERATORE LOGICO
!A	not (\neg = negazione)
A&&B	and (\wedge = congiunzione)
A B	or (\vee = disgiunzione)

Tavole di definizione								
!		&&			 			
F	V	F	F	V	F	F	V	
V	F	V	F	V	V	V	V	

Esempio di variabili logiche: provare...

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

```
void main()
```

```
{ float F; short A, B, C, D, notA, notB, notC, notD, notF;
```

```
  A = 5 == 5; /* A è true */
```

```
  B = 5 == 6; /* B è false */
```

```
  C=8; D=-7; F=1.5f; // C,D,F ?
```

```
  printf("\nA = %d\tB = %d\tC = %d\tD = %d\t F = %f\n",A,B,C,D,F);
```

```
  notA=!A; notB=!B; notC=!C; notD=!D; notF=!F;
```

```
  printf("\nnotA=%d\tnotB=%d\tnotC=%d\tnotD=%d\tnotF=%d\n",  
        notA,notB,notC,notD,notF);
```

```
}
```

come sono considerate?

output

A = 1 B = 0 C = 8 D = -7 F = 1.500000

not A = 0 not B = 1 not C = 0 not D = 0 not F = 0

dove si usano?

```
...
A=-2; B=5;
if ( A<B && A<0 )
{...}
else
{...}...
```

vero!

Che valori hanno le seguenti espressioni?

(5==5) && (6!=2)

V

?

(5>1) && (6<1)

F

(5==5) || (6!=2)

V

?

(5>1) || (6<1)

V

(5==4) || ! (5!=4)

F

In **C** si può **creare il tipo logico** mediante ...

```
enum boolean {false, true};
```

tipo enumerativo:

i valori in parentesi sono associati con i numeri naturali 0,1,2,...

```
typedef enum boolean logical;
```

definisce il nome del nuovo tipo

```
logical bol;
```

dichiara una variabile del nuovo tipo

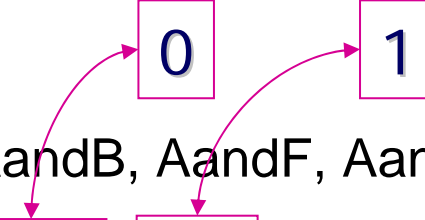
```
bol=true;
```

definisce la variabile

Esempio: provare...

01_01.9

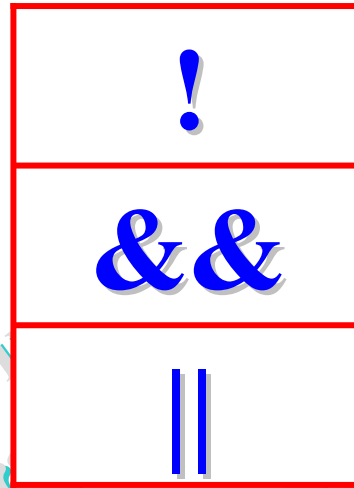
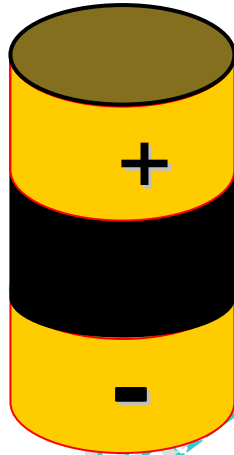
```
#include <stdio.h>
void main()
{float F; short A, B, AandB, AandF, Aandfalso;
enum boolean {false, true}; typedef enum boolean LOGICAL;
LOGICAL falso, vero;
A = 5==5; B = 5==6; C = 8; D = -7; F = 1.5f;
falso = false; vero = true;
printf("\nA = %d\tB = %d\tF = %f\tfalso = %d\tvero = %d\n",A,B,F,falso,vero);
AandB=A&&B; AandF=A&&F; Aandfalso=A&&falso;
printf("\nAandB=%d\tAandF=%d\tAandfalso=%d\n",AandB, AandF,Aandfalso);
}
```



output

```
A = 1      B = 0      F = 1.500000      falso = 0      vero = 1
AandB = 0      AandF = 1      Aandfalso = 0
```

Precedenza tra gli operatori logici in C



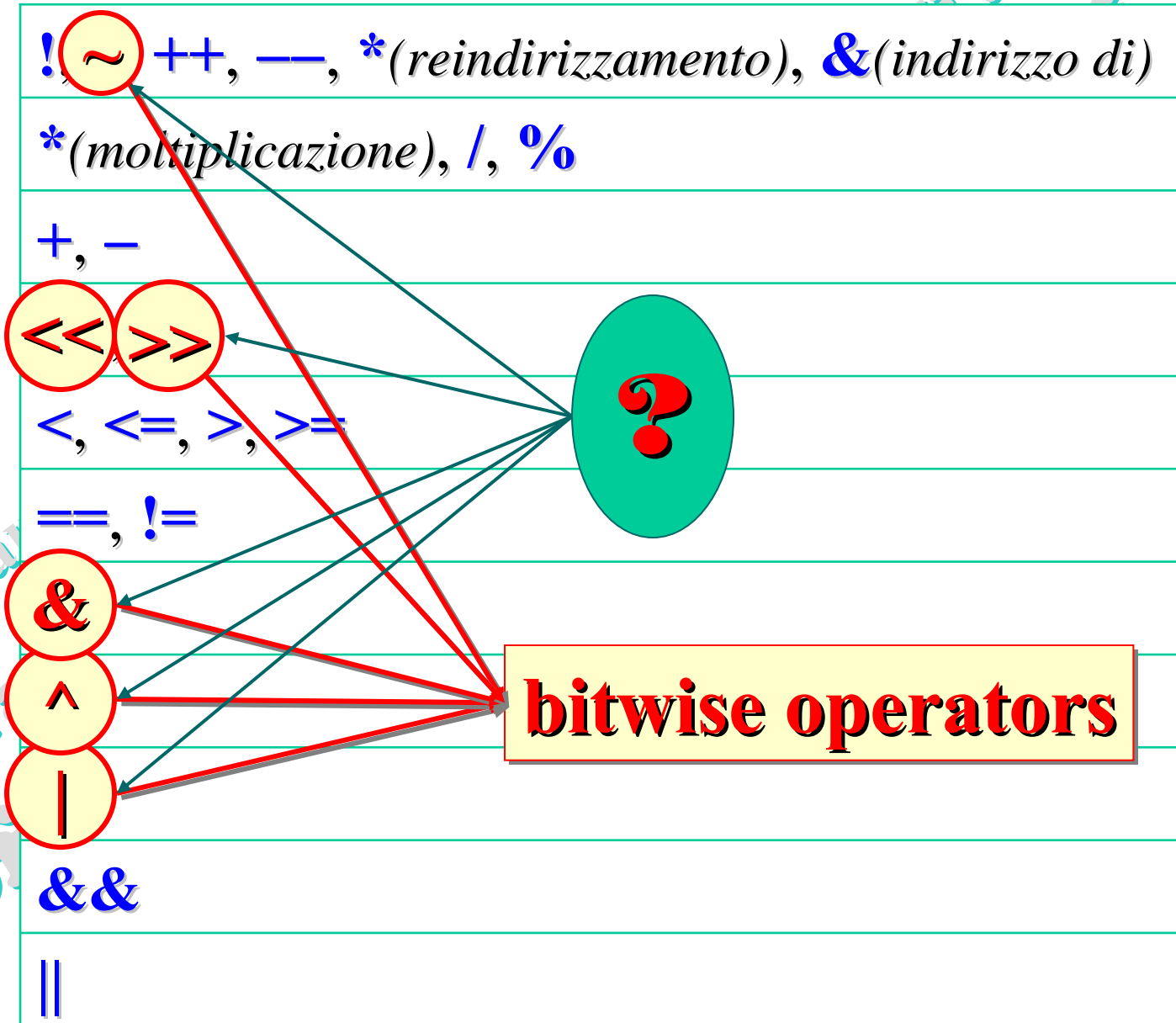
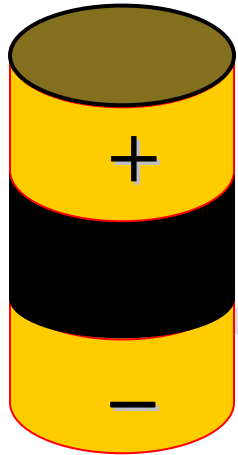
`a < b || a < c && c < d`

ordine di esecuzione ?

`a < b || (a < c && c < d)`

consiglio: ...usare comunque le parentesi!!!

Precedenza tra operatori



Operatori bit a bit (bitwise operators) in C

Il linguaggio **C** mette a disposizione la possibilità, utile nelle applicazioni, di *intervenire direttamente sulla rappresentazione interna binaria* (in memoria) dei dati mediante alcuni *operatori orientati ai singoli bit*.

Tali operatori agiscono sui bit degli operandi di **tipo intero** (**char**, **short int** e **long int**) con o senza segno (rispettivamente **signed** e **unsigned**).

in C	OPERATORE
$\sim A$	NOT (complemento)
$A \& B$	AND
$A B$	OR (OR inclusivo)
$A \wedge B$	XOR (OR esclusivo)

in C	OPERATORE
$A \ll n$	shift a sinistra di n bit
$A \gg n$	shift a destra di n bit

Tavole degli **operatori bitwise in C**

\sim	
0	1
1	0

$\&$	0	1
0	0	0
1	0	1

$ $	0	1
0	0	1
1	1	1

\wedge	0	1
0	0	1
1	1	0

logical operators

!		&&			 		
F	V	F	F	F	F	F	V
V	F	V	F	V	V	V	V

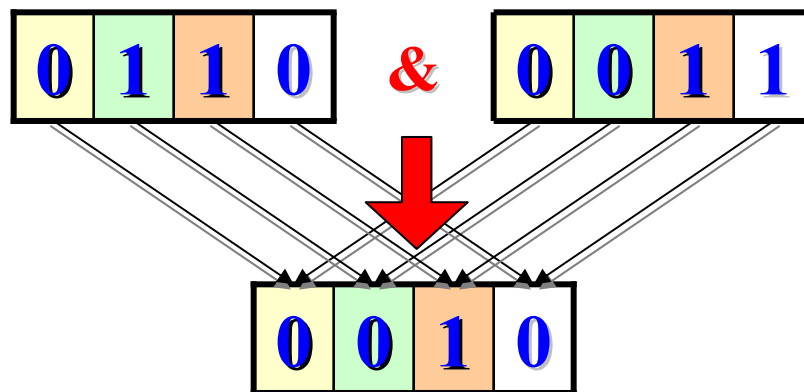
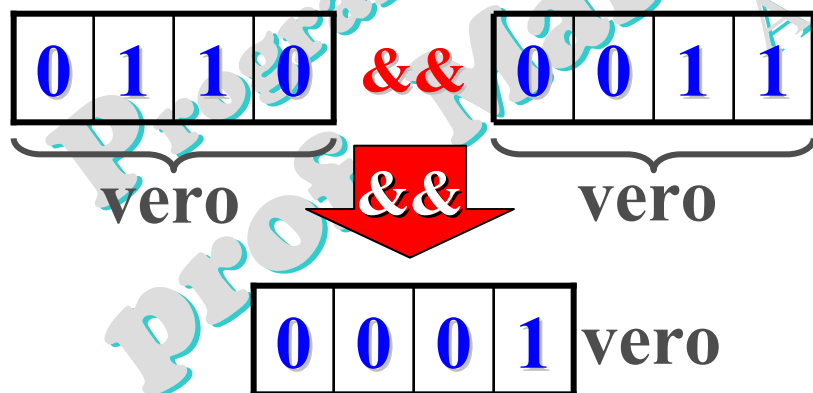
bitwise operators

~		&			 		
0	1	0	0	0	0	0	1
1	0	1	0	1	1	1	1

Che differenza c'è?

gli operandi sono le voci intere

gli operandi sono i singoli bit delle voci



Provare il seguente programma C...

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

```
void main()
```

```
{ char A, NAbitwise, NAbool; unsigned char uA, uNAbitwise, uNAbool;
puts( "\n carattere intero d esadecimale unsigned \n");
A = 'A'; uA = A;
printf("\n A = '%c',int=%4d,hex = %04hx,uns=%4u\n", A, A, A, uA);
NAbitwise = ~A; uNAbitwise = NAbitwise;
printf("\n ~A = '%c',int=%4d,hex %04hx,uns=%4u\n",
      NAbitwise, NAbitwise, NAbitwise, uNAbitwise);
NAbool = !A; uNAbool = NAbool;
printf("\n !A = '%c',int=%4d,hex = %04hx,uns=%4u\n",
      NAbool, NAbool, NAbool, uNAbool);
NAbool = !NAbool; uNAbool = NAbool;
printf("\n !A = '%c',int=%4d,hex = %04hx,uns=%4u\n",
      NAbool, NAbool, NAbool, uNAbool);
}
```

a che serve?

	%c	%d	%x	binario
A	'A'	65	41	0100 0001
~A	'¥'	-66	BE	1011 1110
!A	' '	0	0	0000 0000
!(A)	'☺'	1	1	0000 0001

bitwise
logico

$$190 = 256 - 66$$

aritmetica
modulo 2⁸



Programma C completo

```
#include <stdio.h>
void main()
{ unsigned char A,B,NAbitwise,NAbool,LA,RA,AeB,AoB,AxB;
  puts( " carattere intero u  esadecimale ");
  A = 'A';
  printf("\n  A = '%c',int=%4d, hex = %04hx",A,A,A);
  B = 'B';
  printf("\n  B = '%c',int=%4d, hex = %04hx",B,B,B);
  puts("\n-----");
  NAbitwise = ~A;
  printf("\n  ~A = '%c',int=%4d, hex = %04hx",NAbitwise,NAbitwise,NAbitwise);
  NAbool = !A;
  printf("\n  !A = '%c',int=%4d, hex = %04hx",NAbool,NAbool,NAbool);
  puts("\n-----");
  AeB = A&B;      // and
  printf("\n A&B = '%c',int=%4u, hex = %04hx",AeB,AeB,AeB);
  AoB = A|B;      // or
  printf("\n A|B = '%c',int=%4u, hex = %04hx",AoB,AoB,AoB);
  AxB = A^B;      // xor
  printf("\n A^B = '%c',int=%4u, hex = %04hx",AxB,AxB,AxB);
  LA = A<<2;      // left shift
  printf("\n A<<2 = '%c',int=%4u, hex = %04hx",LA,LA,LA);
  RA = A>>1;      // right shift
  printf("\n A>>1 = '%c',int=%4u, hex = %04hx",RA,RA,RA);
}
```


output

A = 'A' ; B = 'B' ⇒

	%c	%d	%x	binario
A	'A'	65	41	0100 0001
B	'B'	66	42	0100 0010
~A	'\ufffd'	190	BE	1011 1110
!A	' '	0	0	0000 0000
A&B	'@'	64	40	0100 0000
A B	'C'	67	43	0100 0011
A^B	'♥'	3	3	0000 0011
A<<2	'♦'	4	4	0000 01<u>00</u>
A>>1	' '	32	20	<u>00</u>10 0000

unsigned A

$$190 = 256 - 66$$

aritmetica
modulo 2⁸

Esempio: long int (32 bit)

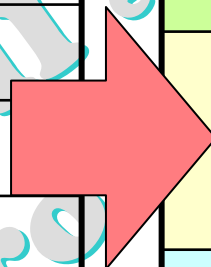
variabile	%d	binario (da %x)
A	5	0000 0000 0000 0000 0000 0000 0000 0101
B	7	0000 0000 0000 0000 0000 0000 0000 0111

risultato		
~A	-6	1111 1111 1111 1111 1111 1111 1111 1010
!A	0	0000 0000 0000 0000 0000 0000 0000 0000
A&B	5	0000 0000 0000 0000 0000 0000 0000 0101
A B	7	0000 0000 0000 0000 0000 0000 0000 0111
A^B	2	0000 0000 0000 0000 0000 0000 0000 0010
A>>3	0	<u>0000</u> 0000 0000 0000 0000 0000 0000 0000
A<<2	20	0000 0000 0000 0000 0000 0000 0001 01 <u>00</u>

Laboratorio:

Scrivere un programma C che legge un carattere maiuscolo (A...Z) e lo trasforma in minuscolo (a...z) tramite operatori bitwise. Realizzare anche il viceversa.

char	ASCII	char	ASCII
A	65	a	97
B	66	b	98
C	67	c	99
D	68	d	100
E	69	e	101
F	70	f	102
G	71	g	103
...		...	



char	ASCII	binario
A	65	0100 0001
a	97	0110 0001
D	68	0100 0100
d	100	0110 0100
G	71	0100 0111
g	103	0110 0111



Operatori bitwise

(prg zardi)

Quale operatore \otimes applicare a ...

... per ottenere

$$\begin{array}{rcl}
 \text{A} & & 01000001 \\
 2^5 & \otimes & 00100000 \\
 \hline
 = a & & 01100001
 \end{array}$$

$$\begin{array}{rcl}
 a & & 01100001 \\
 2^5 & \otimes & 00100000 \\
 \hline
 = A & & 01000001
 \end{array}$$

Laboratorio:

Scrivere un programma C per ruotare di n bit, verso sinistra o verso destra (stabilito in input), il contenuto di una variabile V (mediante gli operatori bitwise).

	binario
V	0110 0110

$W = V \gg 2$

	binario	
W	<u>0001</u> 1001	10

persi!

	binario
V	0110 0110

$W = V \text{rot} > 2$

	binario	
W	<u>1001</u> 1001	10

	binario
V	0110 01 <u>10</u>
	0000 0010
	1000 0000

← salvare!

	binario
V	0110 0110
	<u>0001</u> 1001
	1000 0000
	<u>1001</u> 1001

Esercizi

1 Scrivere una function C

char low_upp(char ch)

che cambia il carattere in input da minuscolo a maiuscolo e viceversa *automaticamente*.

2 Scrivere una function C

char rotate(char ch, char n_bit)

per ruotare di n bit, verso sinistra o verso destra (rispettivamente **n_bit<0** e **n_bit>0**), il contenuto di una variabile char (mediante gli operatori bitwise).