Linguaggio C: Espressioni

Moreno Marzolla
Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
https://www.moreno.marzolla.name/

Copyright © Mirko Viroli
http://mirkoviroli.apice.unibo.it/
Copyright © 2008 Stefano Mizzaro
http://users.dimi.uniud.it/~stefano.mizzaro/dida/Prog0708/
Copyright © 2017, 2020 Moreno Marzolla
https://www.moreno.marzolla.name/teaching/FINFA/

This work is licensed under the Creative Commons Attribution-Non Commercial 2.0 (CC BY-NC 2.0) License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/2.0/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Ringraziamenti

- prof. Mirko Viroli, Università di Bologna
  - http://mirkoviroli.apice.unibo.it/
- prof. Stefano Mizzaro, Università di Udine
  - http://users.dimi.uniud.it/~stefano.mizzaro/

Operatori unari, binari e ternari

- Operatori unari:
  - Si applicano ad un parametro e restituiscono un risultato
  - Esempi: `!` (not logico), `~` (not bit a bit), `-` (usato per invertire il segno, es. `a = -b;`), ...
- Operatori binari:
  - Si applicano a due parametri e restituiscono un risultato
  - Esempi: `+` (somma), `*` (prodotto), `<` (minore di), `<=` (minore o uguale di), ...
- Operatori ternari:
  - Ce n'è uno solo: l'operatore condizionale `?`
  - `(2 == 3) ? 3 : 9` si valuta in 9
  - `(2 == 2) ? 3 : 9` si valuta in 3

(cond) ? Exp$_1$: Exp$_2$ $\longrightarrow$ R

$$R = \begin{cases} Exp_1 & \text{se cond é vera} \neq \emptyset \\ Exp_2 & \text{se cond é falsa} = \emptyset \end{cases}$$

Operazioni aritmetiche

- Se v e w hanno entrambi tipo *T* (**`char`**, **`int`**, **`float`**, **`double`**), allora le seguenti espressioni hanno tipo *T*
  - +v, -v
  - v + w, v - w, v * w
  - v / w (quoziente: 10/3 vale 3; 10.0/3.0 vale 3.33333)
  - v % w (resto: 10%3 vale 1; non si può applicare ad argomenti float o double)

- ...e le seguenti hanno tipo **`int`** e valgono 0 oppure 1:
  - v > w, v < w, (maggiore, minore)
  - v >= w, v <= w, (maggiore o uguale, minore o uguale)
  - v == w, v != w, (uguale, diverso)

Esercizio di programmazione

- Dato un numero intero *n*, $0 \leq n \leq 999$, determinare i valori c2, c1, c0 delle tre cifre della rappresentazione decimale di *n*
  - c0 è la cifra delle unità, c1 delle decine, c2 delle centinaia

$(a != b) \equiv !(a == b)$

12.3 / 7.92 ✓

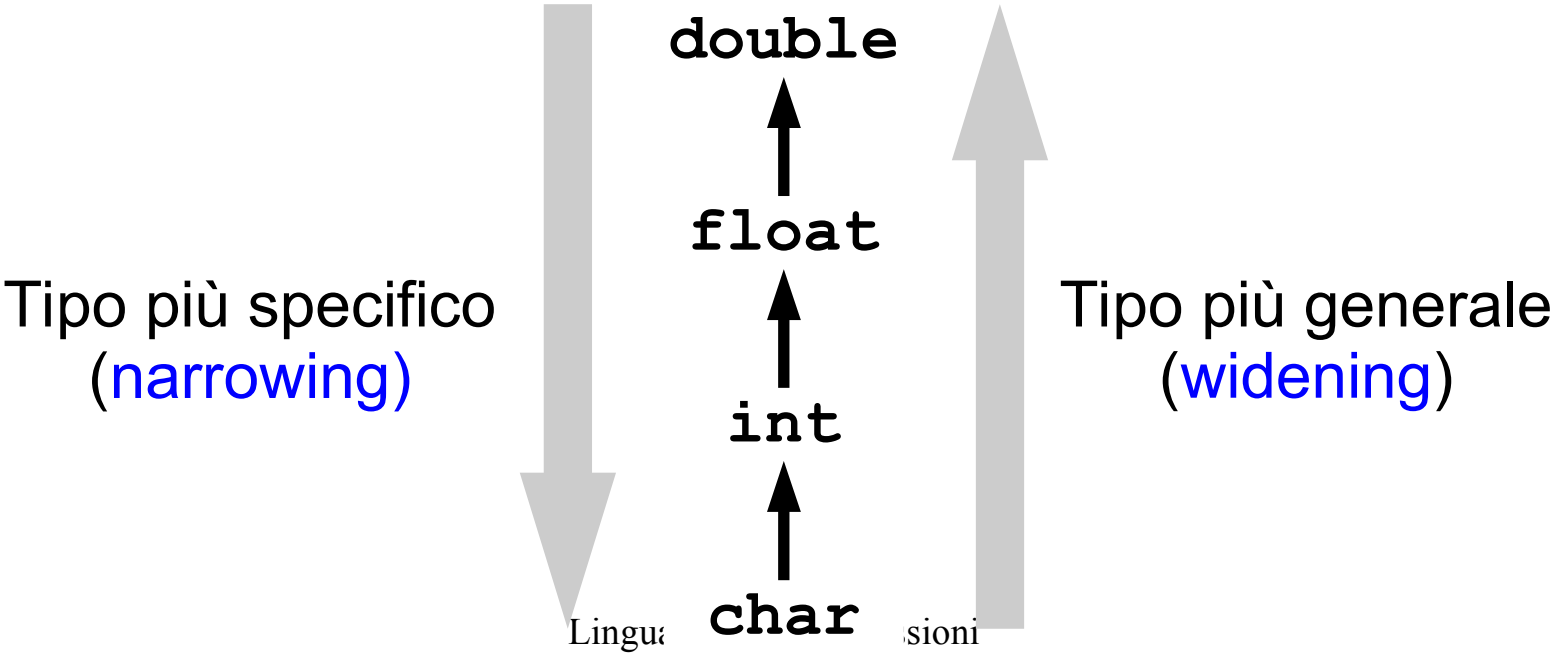
12.3 % 7.92 ✗

12 / 7 ✓

12 % 7 ✓

Conversioni numeriche

- È ammessa la conversione tra qualsiasi coppia di tipi numerici
- Si stabilisce un ordine di generalità dei tipi numerici
  - **`double`** è più generale (*wide*) di **`int`**
  - **`int`** è più specifico (*narrow*) di **`double`**



Conversioni di tipo

- Che succede se applichiamo un operatore (es., la somma) a valori di tipo diverso?
  - Es., voglio sommare un **`int`** e un **`double`**
- Il compilatore effettua una conversione di tipo (detta anche cast o type-cast) verso il tipo più generico
  - **`(5 + 10.0)`** ha tipo **`double`** e vale 15.0
  - **`(10.0f + 5)`** ha tipo **`float`** e vale 15.0f
  - **`(16 / 2.0)`** ha tipo **`double`** e vale 8.0
- È possibile richiedere una conversione esplicita
  - **`(double)2`** converte 2 (intero) in **`double`**
  - **`(float)15.0`** converte 15.0 (double) in **`float`**
  - **`(double)'A'`** converte 'A' (o meglio, il suo codice ASCII che vale 65) in double (!!!!)

```
int a=7, b=2;
double c;
c = a / (double)b;
```

oppure

```
c = ((double)a) / b;
```

→ `c = (double)2 / b`

$$\frac{2+3*7}{2+(3*7)}$$

c → 3.5

(double)(a/b) ⇒ 3.0

3

(int) 3.7 ⇒ int 3

float x = 13.7;

⇓

double

float x = (float) 13.7; ✗

Conversioni numeriche

- Attenzione a risultati "inaspettati"

```
int a = 18, b = 5;
double q1 = a / b; /* q1 vale 3.0 */
double q2 = a / ((double)b); /* q2 vale 3.6 */
double q3 = ((double)a) / b; /* q3 vale 3.6 */
```

Pre/post incremento/decremento

- Gli operatori ++ e -- sono operatori unari che si applicano a variabili di tipo numerico
- Data una variabile a:
  - **`a++`** incrementa a di 1, e restituisce il vecchio valore di a
  - **`a--`** decrementa a di 1, e restituisce il vecchio valore di a
  - **`++a`** incrementa a di 1, e restituisce il nuovo valore di a
  - **`--a`** decrementa a di 1, e restituisce il nuovo valore di a

Esempio

```c
/* pre-post-incr-decr.c – Cosa stampa questo programma? */
#include <stdio.h>

int main( void )
{
    int a = -5;
    int b = a++;
    int c = b--;
    int d = ++c;
    int e = --d;
    printf("a=%d b=%d c=%d d=%d e=%d\n",
           a, b, c, d, e);
    return 0;
}
```

Esempio

```c
/* pre-post-incr-decr.c – Cosa stampa questo programma? */
#include <stdio.h>

int main( void )
{
    int a = -5;
    int b = a++; /* b vale -5, a vale -4      */
    int c = b--; /* c vale -5, b vale -6      */
    int d = ++c; /* d vale -4, c vale -4      */
    int e = --d; /* e vale -5, d vale -5      */
    printf("a=%d b=%d c=%d d=%d e=%d\n",
           a, b, c, d, e);
    return 0;
}
```

```
a=-4 b=-6 c=-4 d=-5 e=-5
```

```
int a = 7;
int b = 3;
int c = (a++) + (--b);
```

✓

7 2

a 8

b 2

c 9

INDEFINITA

```
int d = (++a) + 2*(++a)
```

(1) (2)

a = a+1; ✓

a++; ✓

(2) (1)

```
int a = 7;
int b = (a++) - 5;
          7
```

a 8

b 2 ? ERRORE GRAVE

```
int c = (a++) + (++a);
```

INDEFINITO

Uso

- Il 99% delle volte gli operatori di auto incremento e decremento si usano in combinazione con cicli “for”

```c
/* Cosa stampa questo programma? */
#include <stdio.h>
int main( void )
{
    int i = 0;
    while (i<10) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

```c
/* Cosa stampa questo programma? */
#include <stdio.h>
int main( void )
{
    int i;
    for (i=0; i<10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

- Qui scrivere **`i++`** è un modo sintetico per scrivere **`i=i+1`**

Operatori logici

- `&&` (*and* logico), `||` (*or* logico), `!` (*not* logico)
- Ricordiamo che in C non esiste un tipo “booleano”
  - Al suo posto usa **`int`** o **`char`**
  - 0 rappresenta *false,* qualsiasi valore diverso da zero (tipicamente, 1) rappresenta *true*
- Esempio: a = 18, b = 0, c = -7

`a && b`	0
`a || b`	1
`a && c`	1
`a || c`	1
`!a`	0
`!b`	1

Short-circuit evaluation per operatori logici

- Per gli operatori `&&` e `||` si applica la valutazione short-circuit
  - Si valuta l'espressione, da sinistra verso destra, “quel tanto che basta” per determinarne il valore
- **`p && q`**
  - valuta p
  - se p è *falso* (== 0), restituisci 0 e termina la valutazione
  - altrimenti, valuta q: se q è *vero* (!= 0) restituisci 1, altrimenti restituisci 0
- **`p || q`**
  - valuta p
  - se p è *vero* (!= 0), restituisci 1 e termina la valutazione
  - altrimenti, valuta q: se q è *vero* (!= 0) restituisci 1, altrimenti restituisci 0

Short-circuit evaluation

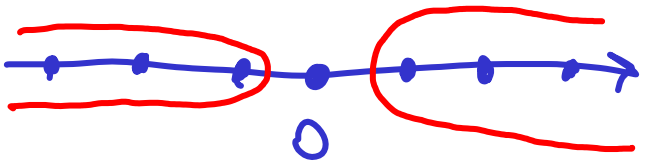
- La valutazione short-circuit risulta molto utile

```
/* short-circuit.c */
#include <stdio.h>

int main( void )
{
    int a = 0, b = 13;
    if ((a != 0) && (b/a > 2)) {
        printf("Condizione vera\n");
    } else {
        printf("Condizione falsa\n");
    }
    return 0;
}
```

int x, y;

Esercizio



- Per quali valori delle variabili intere **x**, **y** queste espressioni sono vere?

1 - `(x > 0) && (x < 0)` → sempre false $\forall x$

2 - `(x > 0) || (x < 0)` → $\forall x \neq 0$

3 - `(x >= 0) && (x <= 0)` → vere solo se $x = 0$

4 - `(x >= 0) && ( (x < 5) || (x > 7) )`

5 - `x && (! y)` $\{(x,y) \mid x \neq 0 \wedge y = 0\}$

6 - `!( (x < 0) || (y < 0) )` $\{(x,y) \mid x \geq 0 \wedge y \geq 0\}$

⑥ $\neg((x<0) \vee (y<0))$

$\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$

(*) $\neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$

$\neg((x<0) \vee (y<0)) \equiv$

$(\neg(x<0)) \wedge (\neg(y<0))$ ho applicato (*)

$(x \geq 0) \wedge (y \geq 0)$

$!(x != 0) \equiv (x == 0)$

(5) $x \wedge (\neg y)$

(1) (2)

(1) vero $x \neq 0$

(2) vero $y = 0$

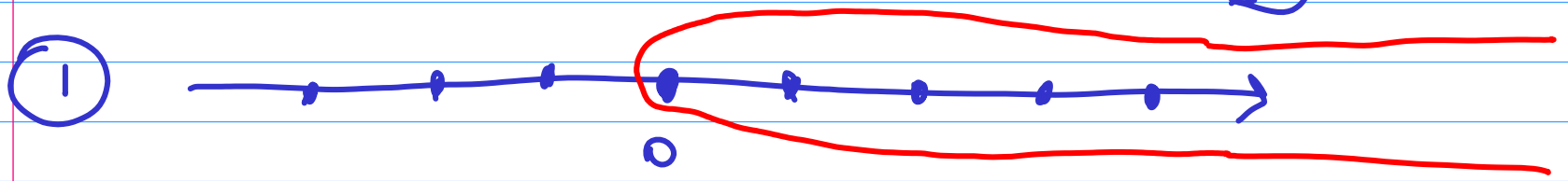
$x \wedge (\neg y)$ è vera

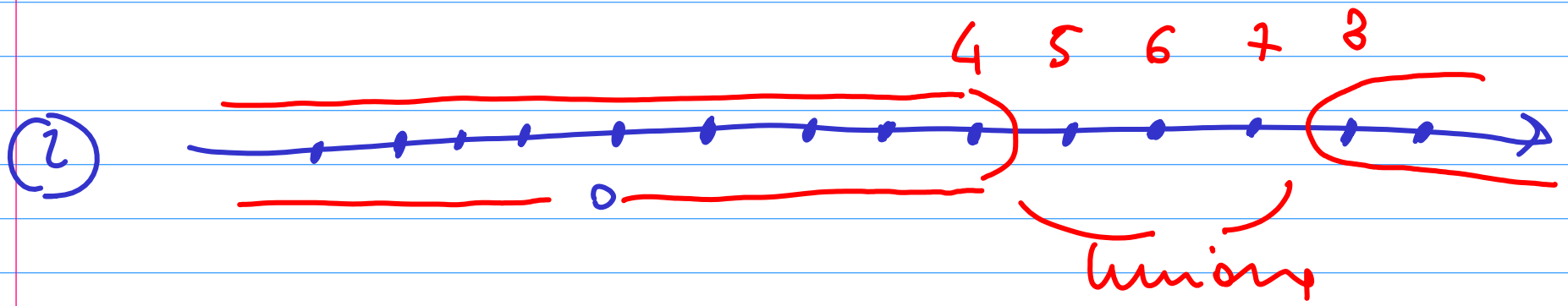
se $\{ x \neq 0 \wedge y = 0 \}$

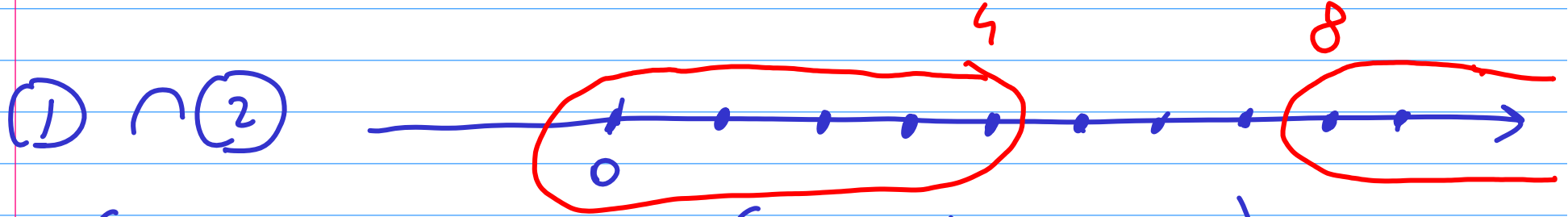
4

$$(x \geq 0) \wedge ((x < 5) \vee (x > 7))$$

① ②







$$\underbrace{((x \geq 0) \wedge (x < 5))}_{x = 0,1,2,3,4} \vee \underbrace{((x \geq 0) \wedge (x > 7))}_{x > 7}$$

Operazioni sui bit

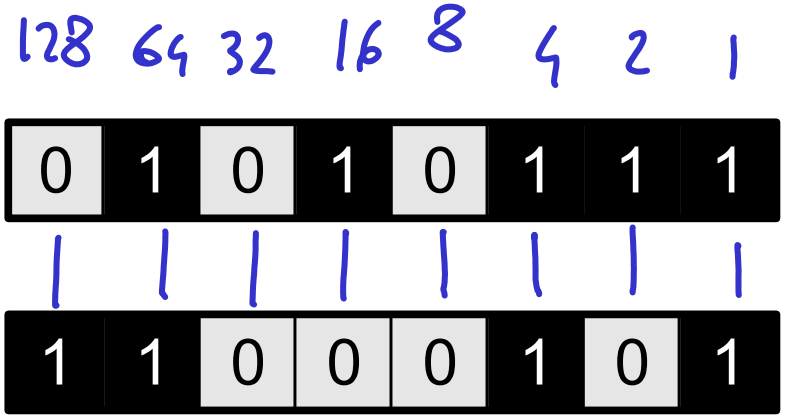
- Se **`v`** e **`w`** hanno tipo **`char`** o **`int`**, allora le seguenti espressioni hanno tipo **`char`** o **`int`**
  - **`v & w`** AND bit-a-bit
  - **`v | w`** OR bit-a-bit
  - **`v ^ w`** XOR ("OR esclusivo") bit-a-bit
  - **`~v`** NOT bit-a-bit
  - **`v << w`** SHIFT a sinistra di w bit
  - **`v >> w`** SHIFT a destra di w bit
- È consigliato usare tali operatori solo su tipi *unsigned*

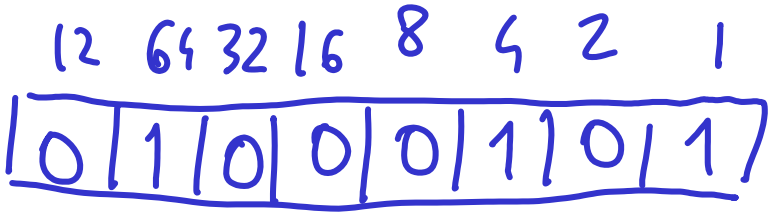
  - Il risultato delle operazioni di shift nel caso in cui **`v`** sia *signed* dipende dal compilatore usato

AND bit a bit

&

```
unsigned char a = 87;
unsigned char b = 197;
unsigned char c = a & b;
```





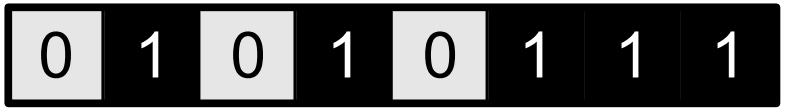
c vale 69

64 + 4 + 1 = 69

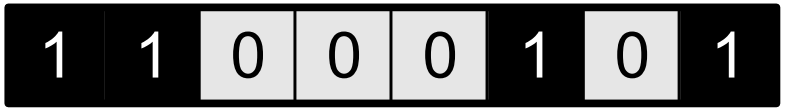
AND bit a bit

&

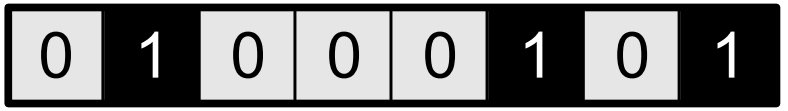
```
unsigned char a = 87;
```




```
unsigned char b = 197;
```




```
unsigned char c = a & b;
/* c vale 69 */
```

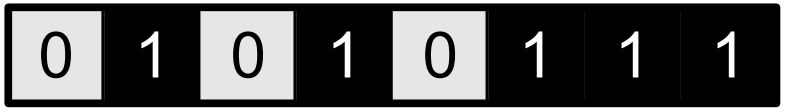


Slide opzionale

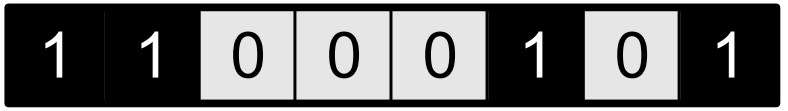
OR bit a bit

|

```
unsigned char a = 87;
```




```
unsigned char b = 197;
```



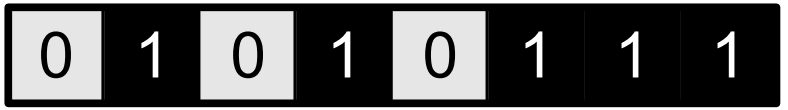

```
unsigned char d = a | b;
```

Slide opzionale

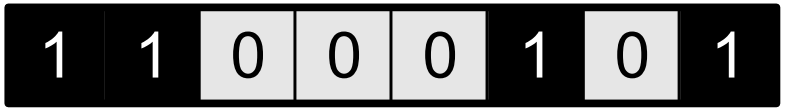
OR bit a bit

|

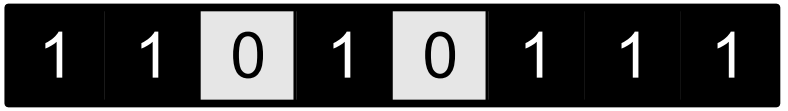
```
unsigned char a = 87;
```




```
unsigned char b = 197;
```



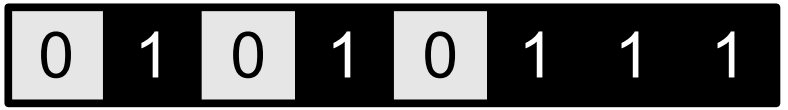

```
unsigned char d = a | b;
/* d vale 215 */
```



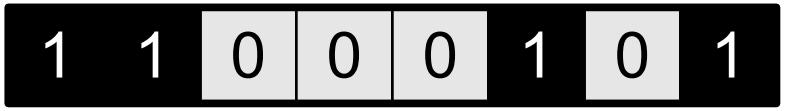
Slide opzionale

XOR bit a bit ^

```
unsigned char a = 87;
```




```
unsigned char b = 197;
```



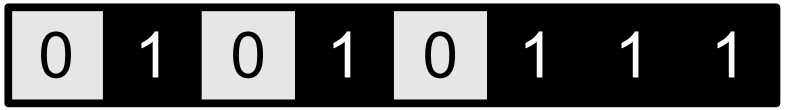

```
unsigned char e = a ^ b;
```

- Errore comune:
  - L'operatore ^ non fa l'elevamento a potenza!
  - Non esiste un operatore predefinito per l'elevamento a potenza in C; occorre realizzarlo tramite una funzione

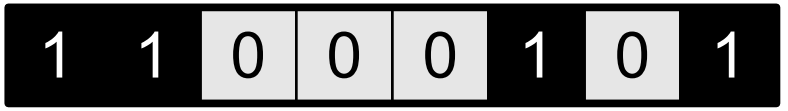
Slide opzionale

XOR bit a bit ^

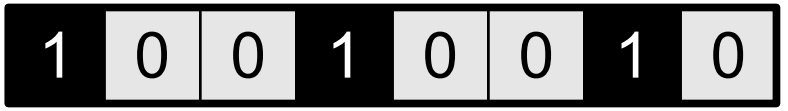
```
unsigned char a = 87;
```




```
unsigned char b = 197;
```




```
unsigned char e = a ^ b;
/* e vale 146 */
```



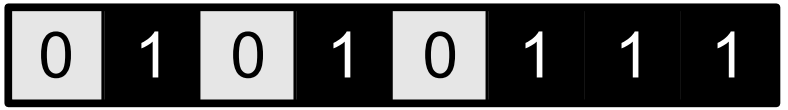
- Errore comune:
  - L'operatore ^ non fa l'elevamento a potenza!
  - Non esiste un operatore predefinito per l'elevamento a potenza in C; occorre realizzarlo tramite una funzione

Slide opzionale

NOT bit a bit

~

```
unsigned char a = 87;
```



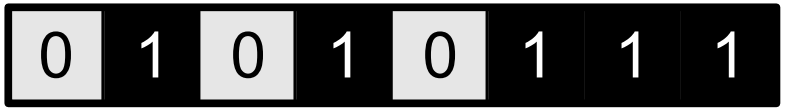

```
unsigned char f = ~a;
```

Slide opzionale

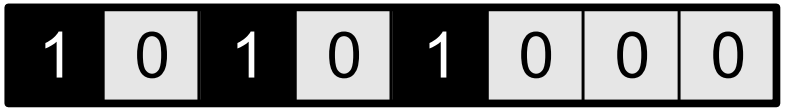
NOT bit a bit

~

```
unsigned char a = 87;
```




```
unsigned char f = ~a;
/* f vale 168 */
```



Slide opzionale

Operatori logici vs operatori bit a bit

- Gli operatori logici **&&**, **||** non vanno confusi con gli operatori bit a bit **&**, **|**

```
int a = 1;
int b = 2;

int ex1 = a && b;    // vale 1 (true)
int ex2 = a & b;     // vale 0
```

- Ricordare inoltre che:
  - Per gli operatori logici si applica la short-circuit evaluation
  - Per gli operatori bit-a-bit NO: le espressioni vengono sempre valutate per intero

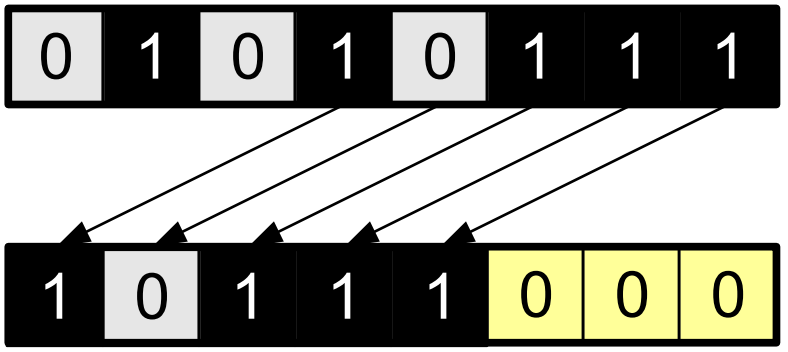
Slide opzionale

Operazioni sui bit

- `v << w` effettua lo shift a sinistra di *w* posizioni della sequenza di bit che rappresenta il valore *v*
  - I *w* bit a destra del risultato sono posti a zero

```
unsigned char a = 87;

unsigned char g = a << 3;
/* g vale 184 */
```

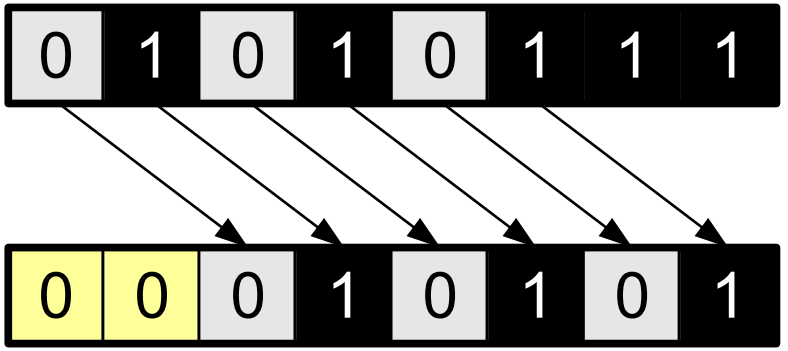


Slide opzionale

Operazioni sui bit

- **`v >> w`** effettua lo shift a destra di *w* posizioni della sequenza di bit che rappresenta il valore *v*
  - Se *v* è *unsigned*, i *w* bit a sinistra del risultato sono zero
  - Se *v* è *signed*, il risultato è dipendente dal compilatore

```
unsigned char a = 87;
```

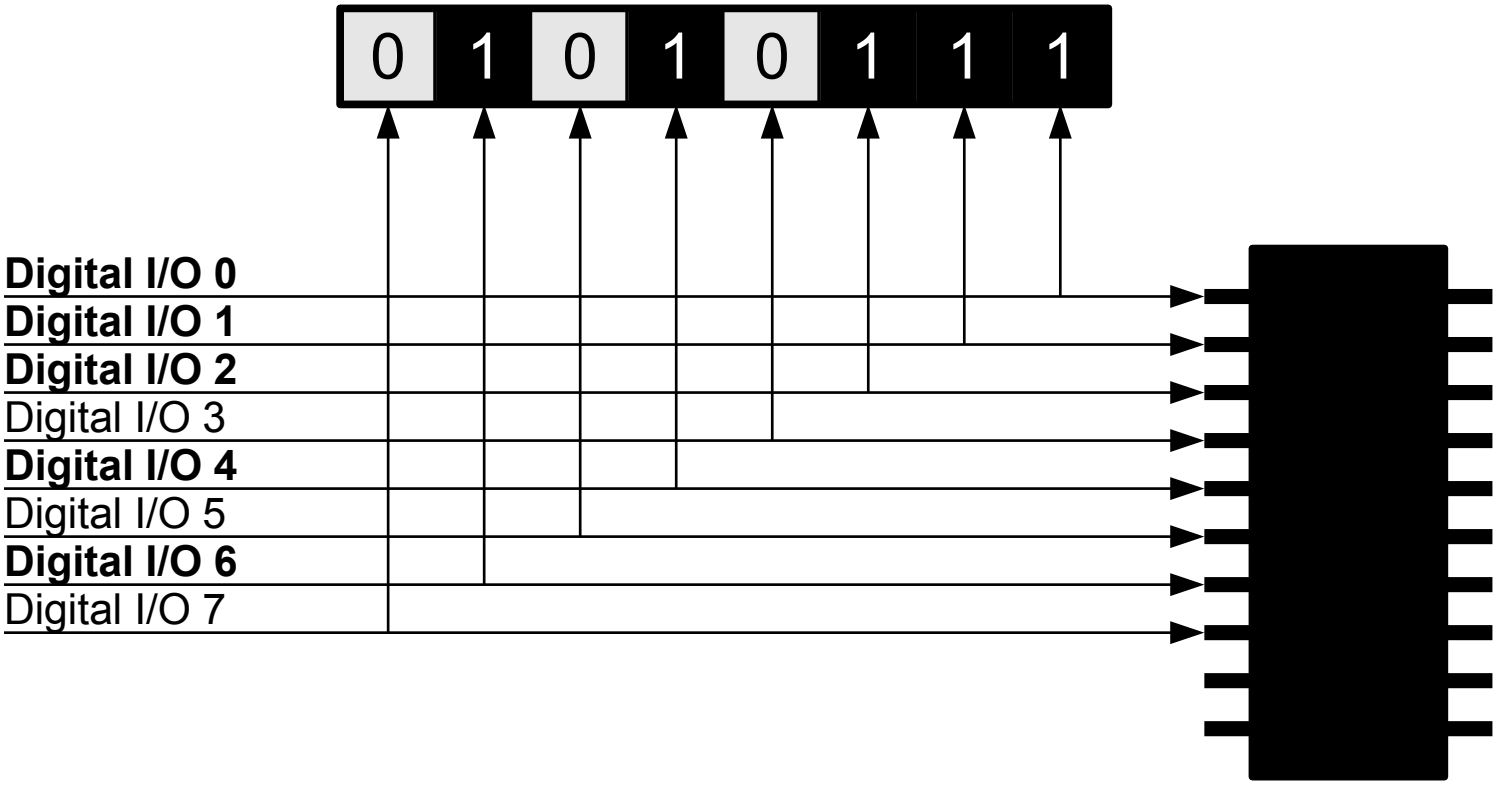



```
unsigned char h = a >> 2;
/* h vale 21 */
```

Slide opzionale

Operazioni sui bit

- Le operazioni sui bit sono fondamentali per programmare microcontrollori



Slide opzionale

Esercizio

- Si consideri una variabile `x` di tipo **`unsigned char`**
- Quanto vale il bit 3 della rappresentazione binaria di `x`?
  - Il bit più a destra è il bit 0
- Che valore si ottiene ponendo a 1 il bit 3 della rappresentazione binaria di `x`?
- Che valore si ottiene ponendo a 0 il bit 3 della rappresentazione binaria di `x`?

Slide opzionale

```
/* bit-op.c - dimostrazione degli operatori bit-a-bit */
#include <stdio.h>

int main( void )
{
    unsigned char a = 87;
    unsigned char b = 197;
    unsigned char c = a & b;
    unsigned char d = a | b;
    unsigned char e = a ^ b;
    unsigned char f = ~a;
    unsigned char g = a << 3;
    unsigned char h = a >> 2;

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("a & b = %d\n", c);
    printf("a | b = %d\n", d);
    printf("a ^ b = %d\n", e);
    printf("~a = %d\n", f);
    printf("a << 3 = %d\n", g);
    printf("a >> 2 = %d\n", h);
    return 0;
}
```

```
a = 87
b = 197
a & b = 69
a | b = 215
a ^ b = 146
~a = 168
a << 3 = 184
a >> 2 = 21
```

Priorità degli operatori

- Il linguaggio C definisce in maniera rigorosa la priorità di ciascun operatore
- *, / hanno priorità maggiore di +, -
  - `a+b*c` si valuta come `a+(b*c)`
- Purtroppo, esistono casi controintuitivi
  - `a & b == 7` viene valutata come `a & (b == 7)`
  - `a + b == 7` viene valutata come `(a + b) == 7`
- Non perdete tempo a imparare la precedenza degli operatori: usate sempre le parentesi!

Assegnamento

- Assegnare un valore a una variabile

  variabile = *espressione*;

  - Prima si valuta l'espressione
  - poi, il valore dell'espressione è assegnato alla variabile
- Esempi
  - `contatore = 0;`
  - `area = base * altezza / 2;`
  - `contatore = contatore + 1;`
- L'assegnamento è a sua volta una espressione!
  - “x = a” è una espressione che ha come valore a

```
int a, b;
a = b = 3; /* a e b valgono entrambi 3
              si valuta come a = (b = 3) */
```

l'operatore = è associativo a DESTRA

$a = b = c = d = 7;$

$a = (b = (c = (d = 7)));$

(1)

(2)

(3)

(4)

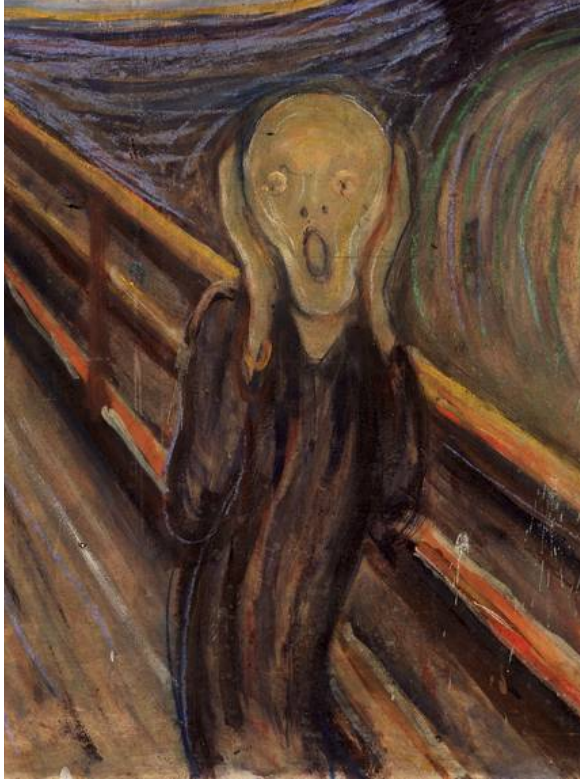
$a + b + c + d$

$\Rightarrow ((a + b) + c) + d$

+ è associativo a sinistra

*/

Errore comune



- Confondere l'operatore di assegnamento `=` con quello di uguaglianza `==`
  - Suggerimento: per confrontare il valore di una variabile (es. `x`) con una costante (es. `3`), meglio scrivere `(3 == x)` anziché `(x == 3)`

```c
#include <stdio.h>
int main( void )
{
    int x = 0;
    if (x = 3) {
        printf("ramo TRUE\n");
    } else {
        printf("ramo FALSE\n");
    }
    return 0;
}
```

Compila, ma probabilmente non fa quello che ci aspettiamo

```c
#include <stdio.h>
int main( )
{
    int x = 0;
    if (3 = x) {
        printf("ramo TRUE\n");
    } else {
        printf("ramo FALSE\n");
    }
    return 0;
}
```

Non compila *(errore di sintassi)*

Errore comune

- Cosa stampa il frammento di codice seguente?

ATTENZIONE...

```
int x = 1;
if (3 < x < 10) {
   printf("VERO\n");
} else {
   printf("FALSO\n");
}
```

- Il modo corretto per verificare se (3 < *x* < 10) è

```
int x = 1;
if ((3 < x) && (x < 10)) {
   printf("VERO\n");
} else {
   printf("FALSO\n");
}
```

x = 1;

if (3 ≤ x ≤ 10) {

Verp

(3 < x) < 10

0 < 10

1 ⇒ Verp

3 ≤ x ≤ 10

if ((3 <= x) && (x <= 10)) {...

if ((x >= 3) && (x <= 10)) {

...

Compound assignment

Nome	Sintassi	Significato
Addition assignment	`a += b`	`a = a + b`
Subtraction assignment	`a -= b`	`a = a - b`
Multiplication assignment	`a *= b`	`a = a * b`
Division assignment	`a /= b`	`a = a / b`
Modulo assignment	`a %= b`	`a = a % b`
Bitwise AND assignment	`a &= b`	`a = a & b`
Bitwise OR assignment	`a |= b`	`a = a | b`
Bitwise XOR assignment	`a ^= b`	`a = a ^ b`
Bitwise left shift assignment	`a <<= b`	`a = a << b`
Bitwise right shift assignment	`a >>= b`	`a = a >> b`