Tipi di dato

Tipi di dato in C

Ogni variabile in C ha associato un tipo, questo permette di:

- Riservare memoria per la codifica dei valori che può assumere
- Rilevare errori legati all'uso di operatori non definiti per quel tipo (es sommare due stringhe)
- Interpretare in modo corretto il suo contenuto

Introduciamo adesso le caratteristiche fondamentali dei tipi C e i dettagli dei vari tipi

Tipi di dato in C

- Per tipo di dato si intende un insieme di valori ed un insieme di operazioni che possono essere applicate su essi

  Es: I numeri interi {..., -2, -1, 0, 1, 2, ... } e le usuali operazioni aritmetiche (somma, sottrazione, . . .) sono un tipo di dato

- Ogni tipo di dato ha una propria rappresentazione in memoria (codifica binaria) che utilizza un certo numero di celle

Tipi in C

- I tipi C si dividono in due gruppi:
- Tipi semplici: per rappresentare semplici informazioni numeriche
  - Esempio: una temperatura, una misura, una velocita, ecc.
  - In C anche i valori vero/falso sono rappresentati come interi (falso = 0, vero ≠ 0)
- Tipi strutturati: per rappresentare informazioni costituite dall'aggregazione di varie componenti
  - Esempio: una data, una fattura, ecc.

Tipi in C

- I tipi strutturati sono costutuiti da aggregazioni di tipi semplici o di altri tipi strutturati
  - Es: un valore di tipo data e costituito da tre valori (semplici),
- Il C mette a disposizione un insieme di tipi predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)

Tipi semplici in C

- Builtin
  - Interi: **`short int, int, long int, unsigned | signed ...`**
  - Caratteri: **`char, unsigned char`**
  - Reali: **`float, double, long double`**
- User defined:
  - tipi enumerati (**`enum`**)
  - tipi union
  - campi di bit

Gli interi

- Due tipi di interi: con segno (signed) e senza segno (unsigned)
- Descriviamo prima i 3 con segno:
  - **`short`** (o **`short int`**, **`signed short int`**)
  - **`int`**, **`signed int`**
  - **`long`** (o **`long int`**, **`signed long int`**)

La funzione `sizeof()`

- da la dimensione in byte di una variabile o di un tipo
- Usiamola per capire le dimensioni in byte dei diversi tipi interi signed
- Restituisce un valore **`long unsigned`**
  - Segnaposto **`%lu`**

Stampare le dimensioni dei tipi ...

```
#include <stdio.h>

int main(void) {

  printf("short è lungo %lu.\n", sizeof(short));
  printf("short int è lungo %lu.\n", sizeof(short int));
  printf("signed short è lungo %lu.\n",  sizeof(signed
short));
  printf("int è lungo %lu.\n", sizeof(int));
  printf("signed int è lungo %lu.\n", sizeof(signed
int));
  printf("long è lungo %lu.\n",  sizeof(long));
  printf("long int è lungo %lu.\n", sizeof(long int));
  printf("signed long è lungo %lu.\n", sizeof(signed
long));

  return 0;
}
```

Cosa otteniamo....

Compileremo ed eseguiremo il codice in laboratorio. In questo modo:

- Sapremo quanti byte occupano i vari tipi
- Verificheremo che alcune diciture sono equivalenti
- Verificheremo che lo standard sia rispettato cioè che la lunghezza dipende dal compilatore, ma è sempre vero che:

```
sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
```

Gli interi con segno

- Se *n* è il numero di bit su cui sono rappresentati (dipende dal compilatore)
  - Possono contenere tutti gli interi nell'intervallo $(-2^{n-1}, 2^{n-1}\text{-}1)$
- I valori limite sono contenuti nel file limits.h, che definisce le costanti:

`SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX`

Gli interi con segno

Costanti: in decimale: 0, 10, -10, . . .

Operatori: +, -, *, /, %, ==, !=, <, >, <=, >=

N.B.: l'operatore di uguaglianza si rappresenta con == (mentre = è utilizzato per il comando di assegnamento!)

Placeholder per printf scanf

`%hd` per short

`%d` per int

`%ld` per long (**`l`** minuscola)

Gli interi senza segno

- Sono i seguenti
  - **`unsigned short`** (o **`unsigned short int`**)
  - **`unsigned`**, **`unsigned int`**,
  - **`unsigned long`** (o **`usigned long int`**)
- Si può modificare il programma precedente che usa sizeof() per vedere le dimensioni

Gli interi senza segno

- Se n è il numero di bit su cui sono rappresentati (dipende dal compilatore)
  - Possono contenere tutti gli interi nell'intervallo $(0, 2^n\text{-}1)$
- I valori limite sono contenuti nel file limits.h, che definisce le costanti:

  `USHRT_MAX, UINT_MAX, ULONG_MAX`

Gli interi senza segno

Costanti: tre tipi diversi

- decimale: 0,1,...10,
- esadecimale: 0xA, 0x2F4B, . . .
- ottale: 012, 027513, ...

Operatori: +, -, *, /, %, ==, !=, <, >, <=, >=

Come gli altri interi

Gli interi senza segno

Placeholder per printf scanf

`%u` in decimale

`%o` in ottale

`%x` esadecimale minuscolo con cifre 0, . . . , 9, a, . . . , f

`%X` esadecimale maiuscolo 0, . . . , 9, A, . . . , F

Per interi short si antepone **`h`** long si antepone **`l`** (minuscola)

Esempio: stampa ottale ed esadecimale

```
#include <stdio.h>

int main(void) {
  unsigned x = 210;
  printf("decimale %u.\n", x);
  printf("ottale %o.\n", x);
  printf("esadecimale %x.\n", x);
  return 0
}
```

Caratteri

- Rappresentati su un singolo byte
  - Rappresentano caratteri alfanumerici attraverso il codice ASCII (American Standard Code for Information Interchange)
- Esempi di valori:

carattere	'0'	...	'9'	':'	';'	'<'
Val. decimale	48		57	58	59	60
carattere	'a'	...	'z'	'{'	'|'	'}'
Val. decimale	97	...	122	123	124	125
carattere	'A'	...	'Z'			
Val. decimale	65	...	90			

Caratteri

- Sono degli interi a tutti gli effetti
  - Rappresentati su un byte
  - Possono essere **`signed`** ed **`unsigned`**
  - Si possono usare come caratteri o esattamente come interi
- Possono essere stampati con placeholder
  - **`%c`** stampa il carattere o
  - **`%d`** o **`%u`** come interi

Esempio: stampa ASCII maiuscole

```
#include <stdio.h>

int main(void) {
  char a;
  for ( a = 'A' ; a <= 'Z'; a++)
     printf("lettera %c codice %d.\n", a, a);
  return 0
}
```

Funzioni stampa/lettura caratteri

- **`stdio.h`** fornisce due funzioni specifiche per i caratteri
  - **`c = getchar()`** elegge un nuovo carattere dallo standard input e lo restituisce come valore (assegnato a **`c`**)
  - **`putchar(c)`** stampa **`c`** come carattere sullo standard output

Reali

- Rappresentati in virgola mobile
  - **`float, double, long double ..`**
- I limiti specifici della rappresentazione si trovano nel file **`float.h`**
  - Es : **`FLT_MAX`** e **`FLT_MIN`** sono il massimo e minimo rappresentabile del tipo float, **`FLT_MANT_DIG`** da il numero di cifre della mantissa etc...
  - Le costanti nel file sono moltissime e specificano anche il tipo di arrotondamento usato nelle varie operazioni
- La rappresentazione dipende dall'implementazione ma anche per i reali deve valere

```
sizeof(float)<=sizeof(double) <= sizeof(long double)
```

Reali

- Le costanti si possono esprimere in due modi: con il classico punto e con la notazione esponenziale :

```
double x, y, z, w;
x = 123.45;
y = 0.0034; /* oppure y = .0034 */
z = 34.5e+20; /* oppure z = 34.5E+20 */
w = 5.3e-12;
```

- Gli operatori sono gli stessi degli interi (tranne `%`)

Reali

- Stampa con **`printf()`** si hanno più opzioni
  - Virgola fissa

```
float x = 123.45;
/* uso valori di cifre predefiniti */
printf("%f", x);  /* stampa "123.4499997" */
/* uso 8 cifre di cui 3 decimali */
printf("%8.3f", x); /* "  123.450" (2 spazi
  all'inizio) */
```

  - Esponenziale

```
float x = 123.45;
/* uso valori di cifre predefiniti */
printf("%e", x);  /* stampa "1.234500e+02" */
/* uso 10 cifre di cui 3 decimali */
printf("%10.3e", x); /* stampa " 1.234e+02" */
```

Reali

- Riassunto specificatori

	float	double	long double
printf()	`%f %e`	`%f %e`	`%Lf %Le`
scanf()	`%f %e`	`%lf %le`	`%Lf %Le`

La libreria matematica

- Libreria matematica (**`libm`**) fornisce tutte le più comuni operazioni matematiche per operare sui reali
  - es. `sqrt(),ceil(), tanh(), pow(),...`
  - per utilizzarla includere il file
    - `#include <math.h>`
  - compilare con

    `gcc … … … -lm`

    che richiede il collegamento del codice relativo alle funzioni di libreria

La libreria matematica

- Permette di verificare varie anomalie nelle computazioni con i reali
  - È possibile capire se si tratta di un infinito o di un numero vero
    - `fpclassify(), isinf(), isnan()...`
  - Possiamo utilizzarle quando non siamo sicuri
    - Per capire se il risultato di una operazione è ancora un numero reale rappresentato correttamente
    - Per verificare se si è verificato un overflow/errore di cancellazione....
  - Le sperimenteremo in laboratorio

Conversioni di tipo

- Ci sono situazioni in in cui si hanno conversioni di tipo:
  - *Conversione implicita degli operandi*: quando in un'espressione compaiono operandi di tipo diverso, ad esempio

    ```
    double x = 123.45; int s=4;
    x = x / s;
    ```

    `s` viene prima converito (implicitamente) a double ed applicata la divisione fra **`double`**
  - *Conversione implicita durante un'assegnamento* **`x = y`**, quando il tipo di **`y`** e diverso da quello di **`x`**
  - *Conversione esplicita con l'operatore di cast* es: **`(int)(double)`**

    ```
    double x = 123.45, int s=4;
    s = (int) x ;
    ```
  - *Conversione implicita* nel passaggio dei parametri/risultato alle funzioni (ne parliamo più avanti)

Conversioni implicite degli operandi

- Quando un'espressione del tipo *x op y* coinvolge operandi di tipo diverso, avviene una conversione implicita secondo le seguenti regole:

1. ogni valore di tipo **`char`** o **`short`** viene convertito in **`int`**
2. se dopo il passo 1. l'espressione è ancora eterogenea si converte l'operando di tipo inferiore facendolo divenire di tipo superiore secondo la seguente gerarchia:

`int` → **`long`** → **`float`** → **`double`** → **`long double`**

Esempio: **`int x; double y;`**

Nel calcolo di **`(x+y)`**: 1. **`x`** viene convertito in **`double`**

2. viene effettuata la somma tra valori di tipo **`double`**

3. il risultato e di tipo **`double`**

Conversioni implicite nell'assegnamento

- Quando un assegnamento $x = y$ coinvolge espressioni di tipo diverso
  - La conversione avviene sempre con il tipo della variabile a sinistra,
  - Si può perdere informazione, es:

```
int i;
float x = 2.1, y = 4.5;
i = x + y;
printf("%d", i); /* stampa 6 */
```

  - In questi casi ci possiamo fare avvertire dal compilatore con opportuni warning

Conversioni di tipo

- Una conversione può o meno coinvolgere un cambiamento nella rappresentazione del valore.
  - da **`short`** a **`long`** SI(dimensioni diverse)
  - da **`int`** a **`float`** SI (anche se stessa dimensione)
  - Da **`int`** a **`unsigned`** NO (anche se cambia l'interpretazione!)

Conversioni di tipo

- Ci sono situazioni in cui è fondamentale usare la conversione di tipo per avere un risultato corretto degli operatori:
  - ad esempio, supponiamo di voler calcolare la media (con eventuali decimali) di tre numeri interi usando il seguente frammento di codice

```
int a = 3, b = 3, c = 4, somma;
double media;
somma = a + b + c;
media = somma / 3;
printf("%f\n",media);
```

  - In questo caso il risultato stampato è 3, invece di 3,333....

Che cosa è successo ?

Conversioni di tipo

- È stato usato l'operatore di divisione (/) intera in quanto a destra e a sinistra ci sono due valori interi!
  - Overloading: lo stesso simbolo viene utilizzato per denotare più operazioni diverse ed il compilatore sceglie quello giusto in base al contesto in cui è applicato
  - Per avere il funzionamento corretto dobbiamo rendere artificialmente reale uno dei due operando con l'operazione di casting :

```
int a = 3, b = 3, c = 4, somma;
double media;
somma = a + b + c;
media = somma / (double) 3; /* oppure somma/3. */
printf("%f\n",media);
```

  - In questo caso il risultato stampato è correttamente 3,333....

Valori booleani

- Quelli che abbiamo già usato nelle condizioni di **`if, for`**
- In C, questi valori sono interi

$$0 \leftrightarrow FALSO$$

$$N \neq 0 \leftrightarrow VERO$$

- Gli operatori di confronto **`(<, <=, ==`** etc ...) restituiscono 0 se la condizione è falsa e 1 se è vera
  - Es: **`(2<3)`** vale 1 mentre **`(4<1)`** vale 0

Operatori booleani

- Sono gli operatori logici che permettono di combinare fra loro i valori di verità
  - Negazione **(!)**
  - Congiunzione o AND **(&&)**
  - Disgiunzione o OR **(||)**

a	b	a && b	a || b	!a
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Operatori booleani

- Ad esempio:
  - `(a >= 10) && (a <= 20)`

  è vero (1) se **a** è compreso fra 10 e 20
  - `(b <= -5) || (b >= 5) || (b == 0)`

  vero se **b** è maggiore di 5 o minore di -5 oppure b è 0
- La valutazione di questi operatori è "lazy" letteralmente *pigra*, cioè vengono valutati da sinistra a destra e ci si ferma appena si sa già il risultato
  - Ad esempio, se b è uguale a -6 il valore di || è già 1 (vero) dopo il primo confronto e gli altri confronti non servono
  - Nel && ci possiamo fermare e restituire 0 (falso) appena uno dei valori è falso

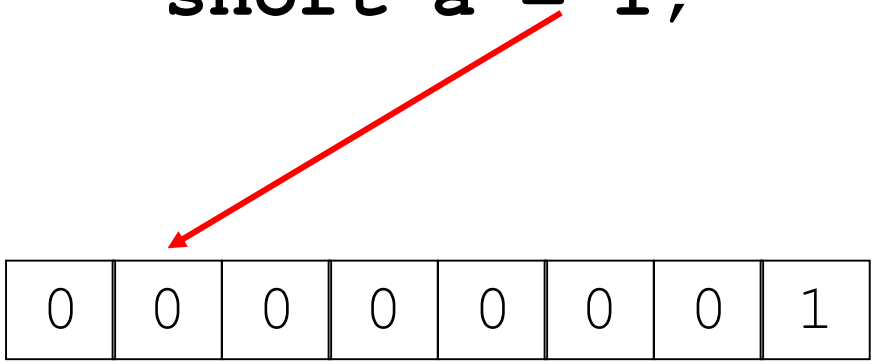
Gli operatori bit a bit

- Lavorano sugli interi e i caratteri con segno e senza segno
- & (and), | (or), ^ (xor), ~ (complemento)
  - Lavorano sui bit corrispondenti dei valori coinvolti

A_i	B_i	(A | B)_i	(~A)_i	(A&B)_i	(A^B)_i
0	0	0	1	0	0
0	1	1	1	0	1
1	0	1	0	0	1
1	1	1	0	1	0

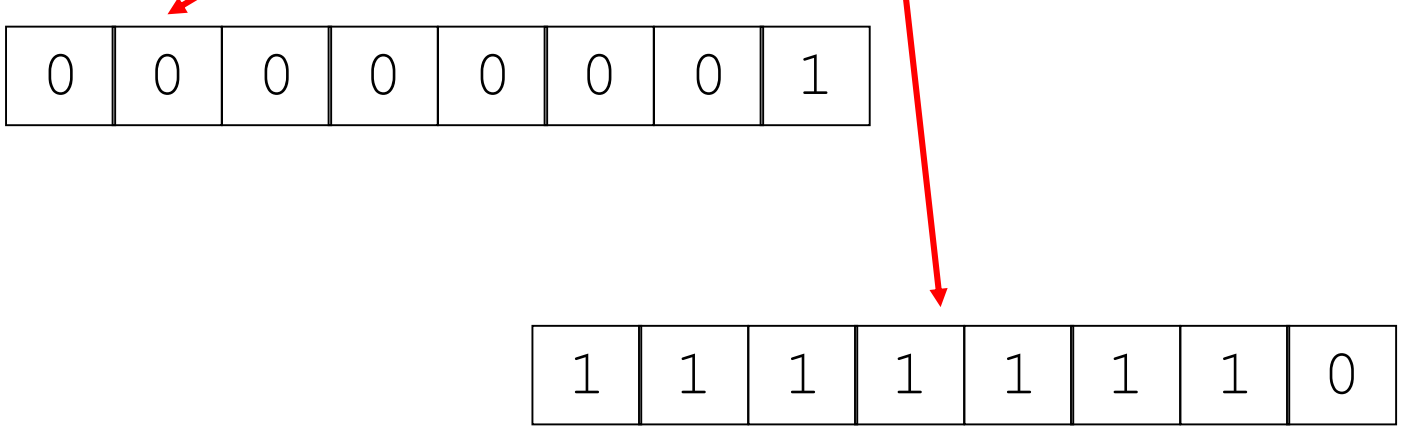
Facciamo qualche esempio

```
short a = 1;
```



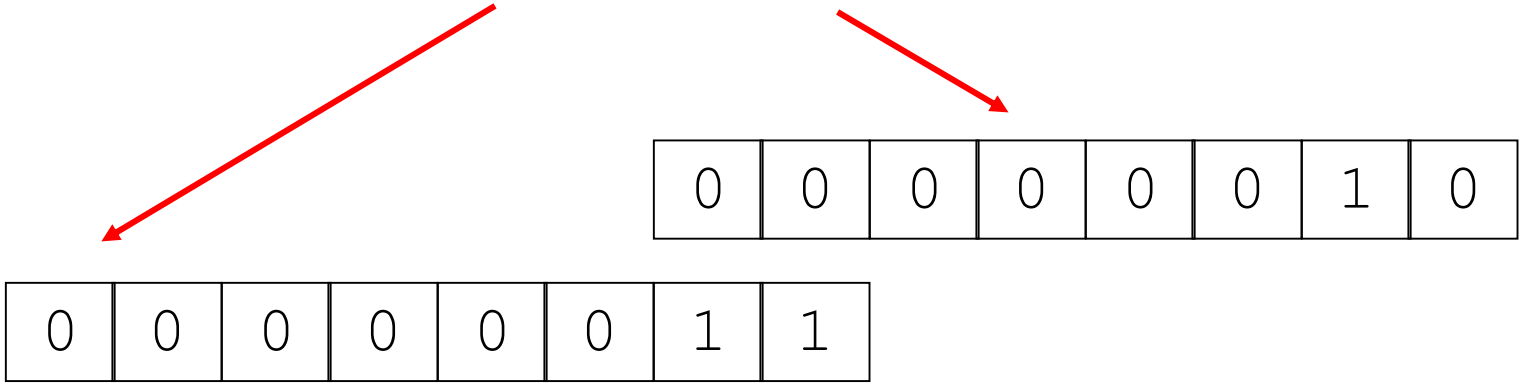
Facciamo qualche esempio

```
short a = 1, b = ~a;
```



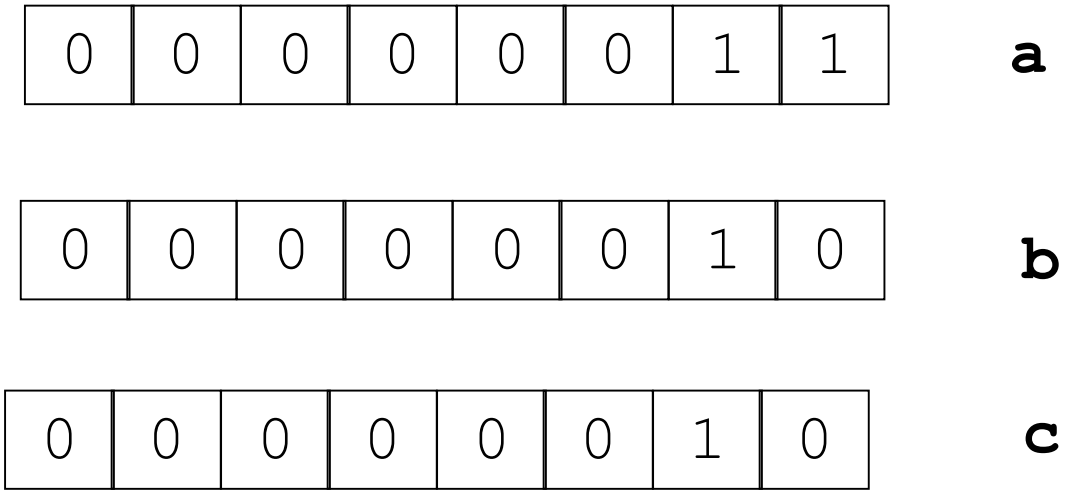
Facciamo qualche esempio

```
short a = 3, b = 2;
```



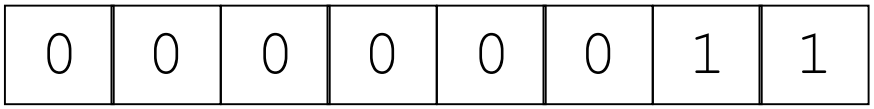
Facciamo qualche esempio

```
short a = 3, b = 2;
short c = a & b;
```

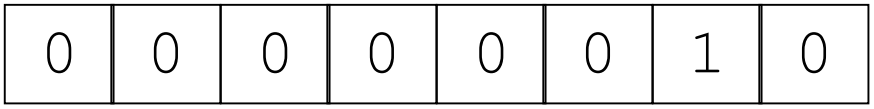


Facciamo qualche esempio

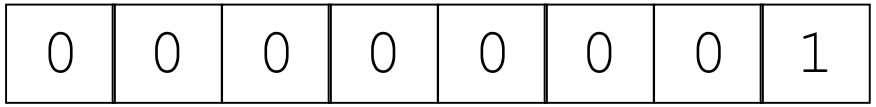
```
short a = 3, b = 2;
short c = a ^ b;
```



a



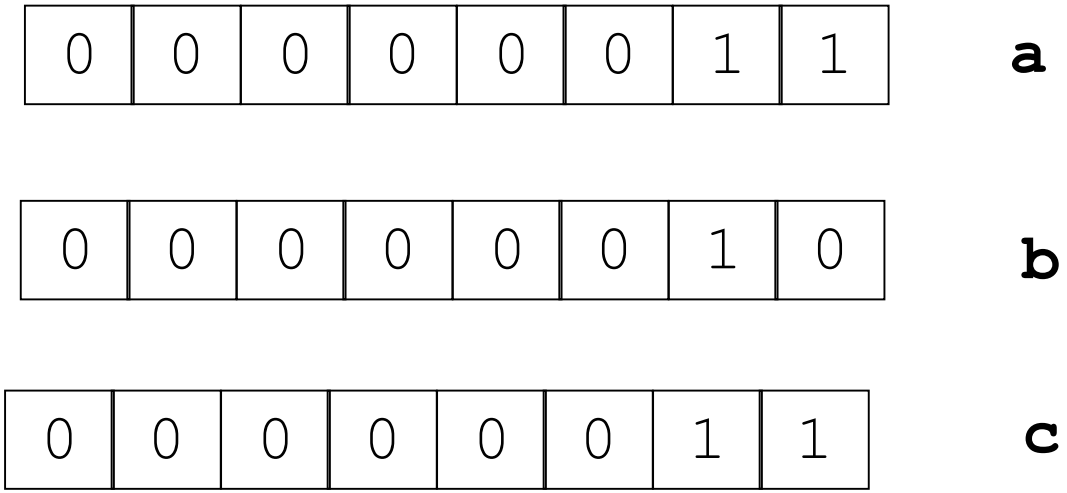
b



c

Facciamo qualche esempio

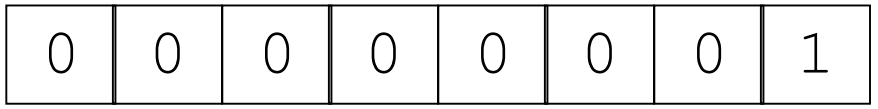
```
short a = 3, b = 2;
short c = a | b;
```



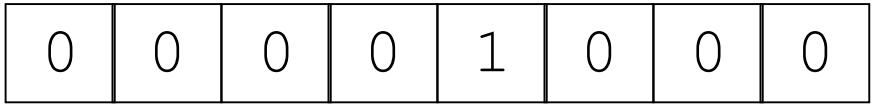
Gli operatori shift (dx e sx)

- << (lshift), >> (rshift)
  - Spostano verso destra o verso sinistra la rappr. binaria ad esempio

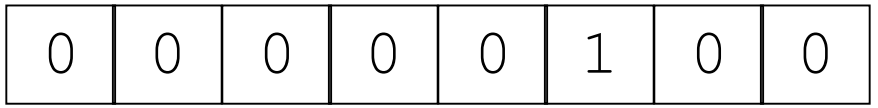
```
int b = 1;
```




```
b =  b <<  3;
```

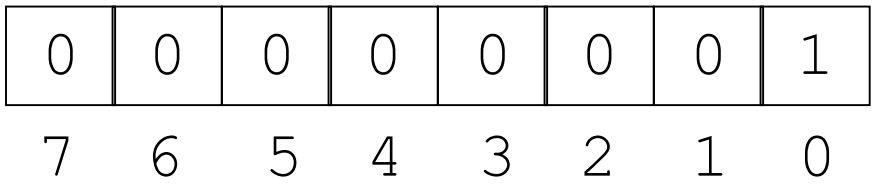



```
b =  b >>  1;
```



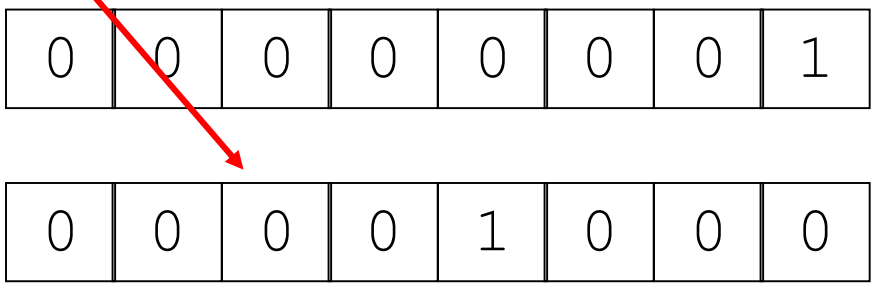
Es: ?

- a = a & ~(1 << 3)



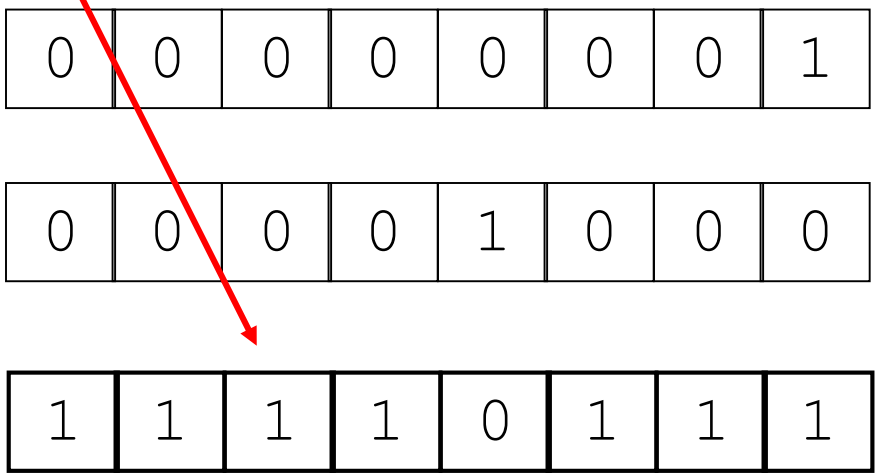
Es ?

- a = a & ~(1 << 3)



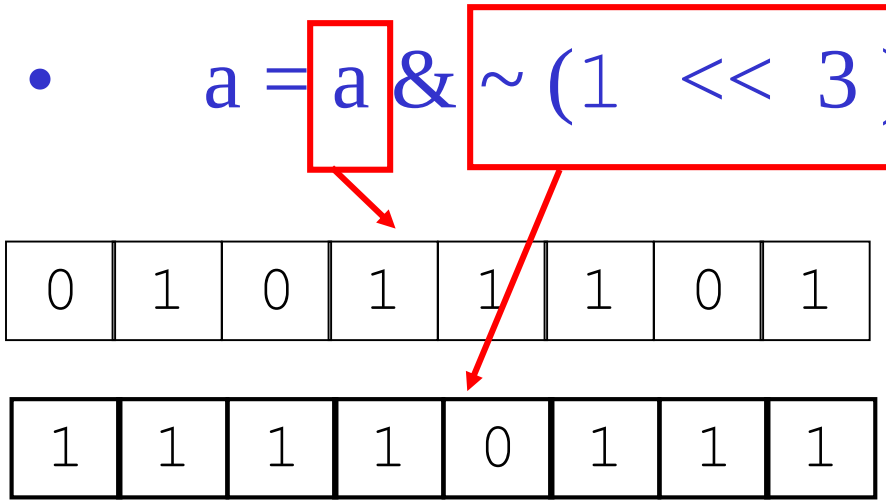
Es ?

- a = a & ~ (1 << 3)



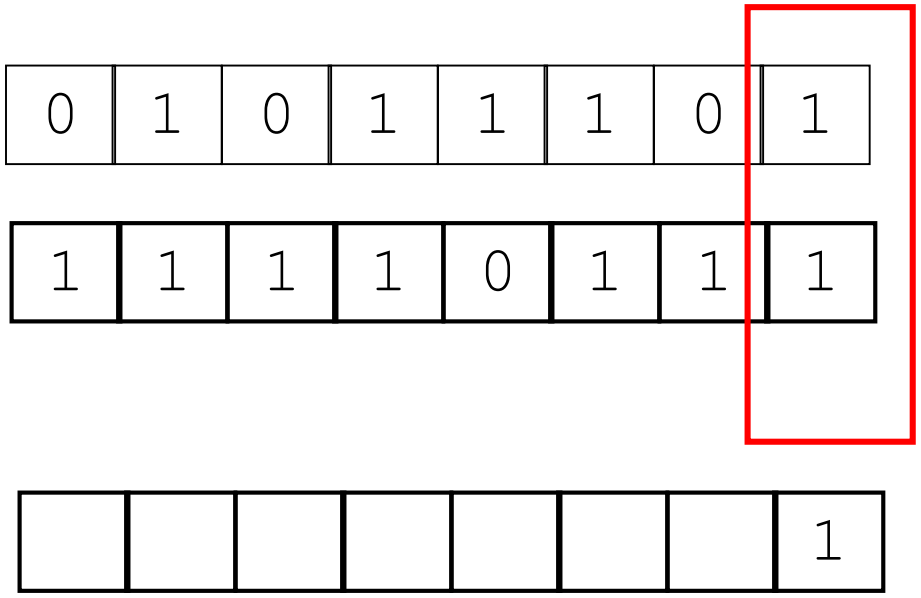
Es ?

- a = a & ~ (1 << 3)



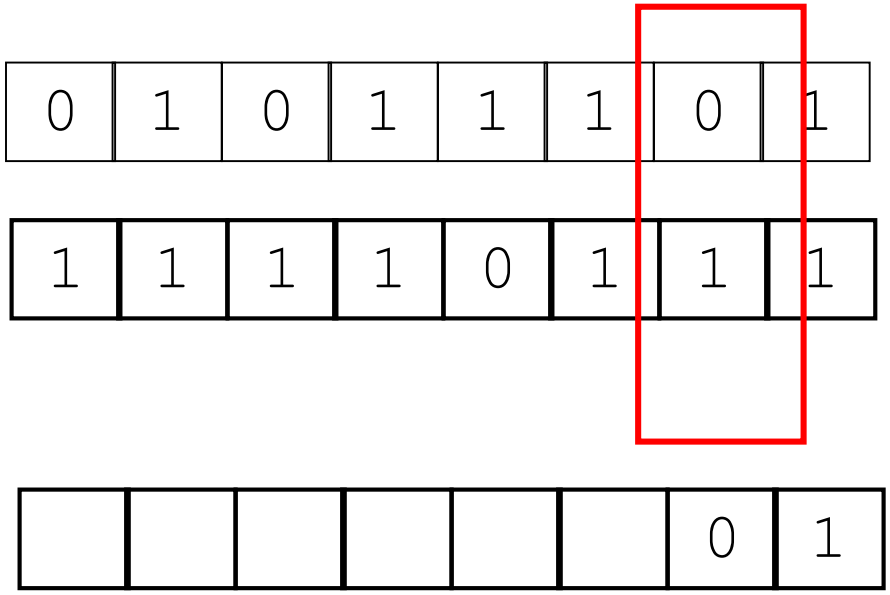
Es ?

- a = a & ~ (1 << 3)



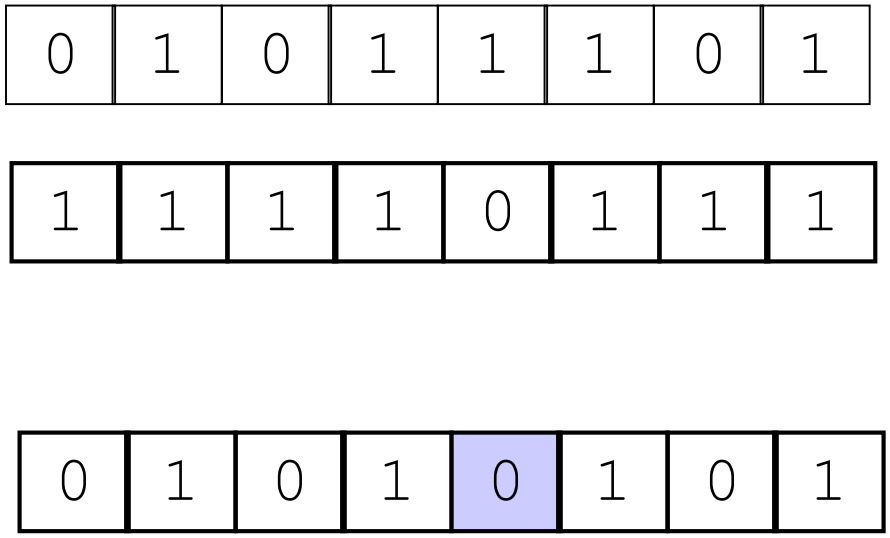
Es ?

- a = a & ~ (1 << 3)



Es ?

- a = a & ~ (1 << 3)

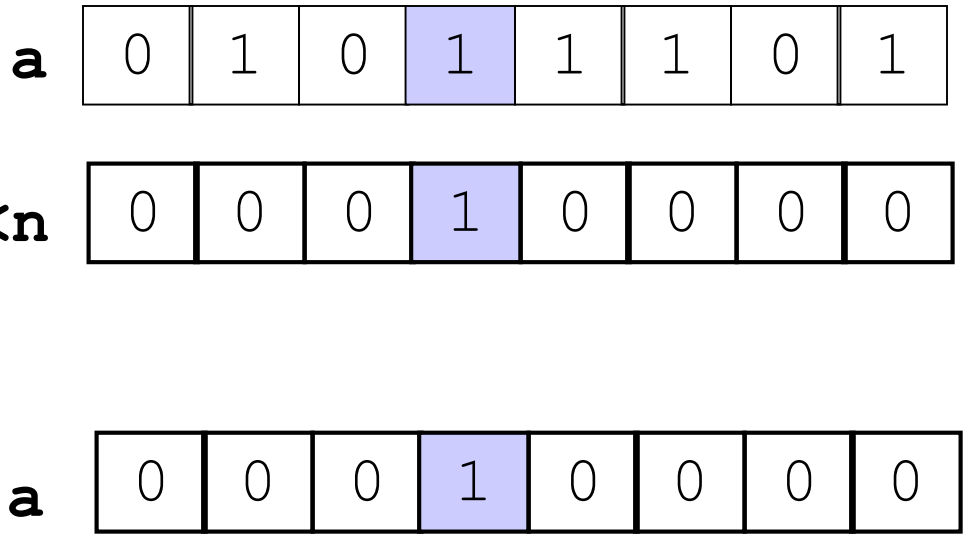


Nuovo valore di a =85

Abbiamo azzerato il bit 3

Selezionare l'n-esimo bit di un intero

```
int a=93; int n=4; int i;
int bit_n;
if (((1<<n) & a)!= 0)
     bit_n = 1;
  else
     bit_n = 0;
```



Stampare i K bit meno significativi di un intero

```
int a = 93, i;

for (i=0; i<K; i++)
  if (((1<<i) & a)!= 0)
     printf(”1”);
  else
     printf(”0”);
```