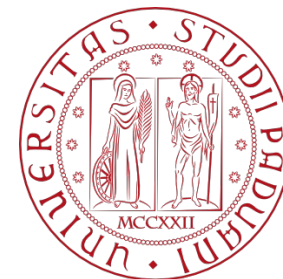


# Comandi di Base



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Comandi di base del linguaggio C:

- Espressioni
- Variabili
- Comandi di scelta
- Iterazione

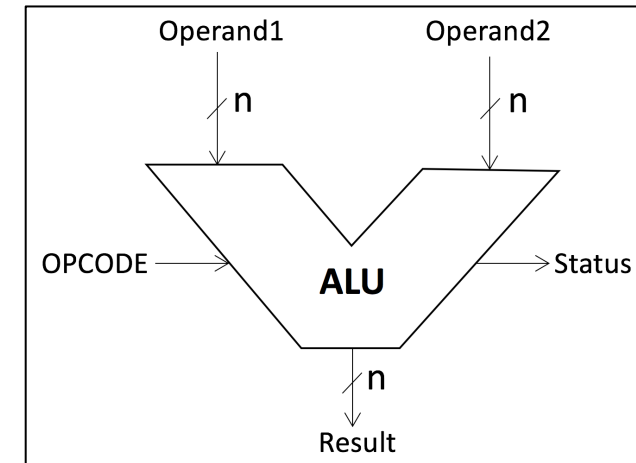
- Problema: vogliamo poter calcolare in un programma C espressioni aritmetiche, ad es.  $7(2+5)-23$

| Operazione in C | Operatore aritmetico | Espressione algebrica              | Espressione in C   |
|-----------------|----------------------|------------------------------------|--------------------|
| Addizione       | +                    | $f + 7$                            | <code>f + 7</code> |
| Sottrazione     | -                    | $p - c$                            | <code>p - c</code> |
| Moltiplicazione | *                    | $bm$                               | <code>b * m</code> |
| Divisione       | /                    | $x/y$ o $\frac{x}{y}$ o $x \div y$ | <code>x / y</code> |
| Resto           | %                    | $r \bmod s$                        | <code>r % s</code> |

- la virgola nei numeri reali si esprime col . es. 2.1

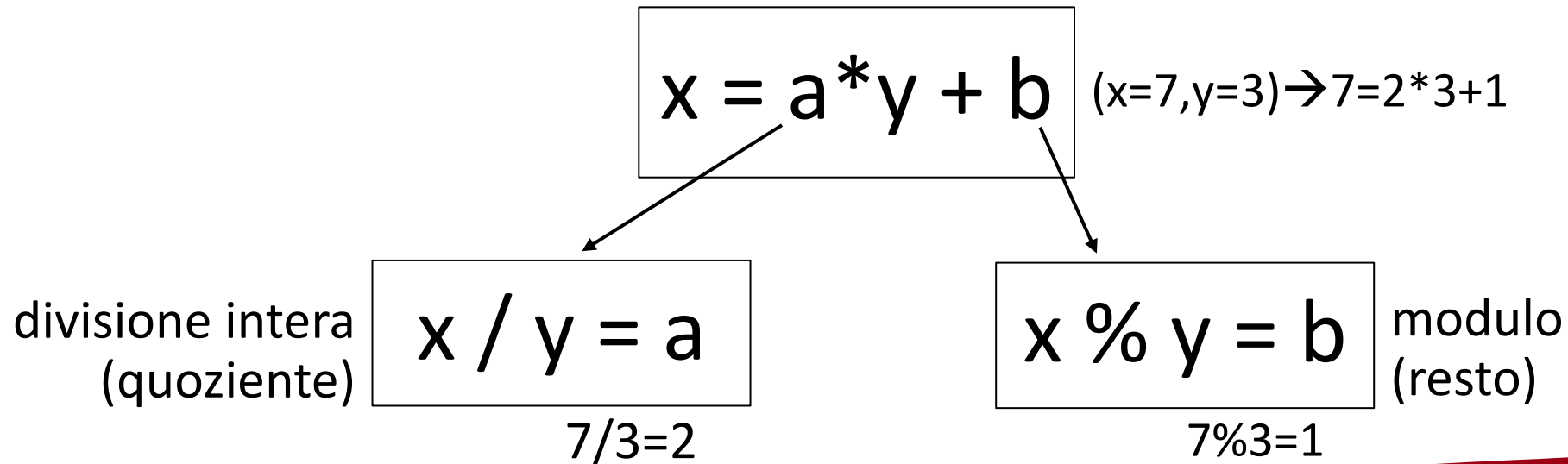
## Vincoli dal linguaggio macchina

- ogni circuito ha al massimo due ingressi dello stesso tipo (interi, reali)
  - quindi  $2+5+3$  dovremmo scomporlo in una serie di operazioni binarie:  $x=2+5$ ;  $x+3$
  - il C ci permette di scrivere espressioni con più termini, es.  $2+5+3$ , perché le traduce lui automaticamente (in modo non ambiguo) in sequenze di operazioni binarie: es.  $(2+5)+3$
  - per farlo il C definisce le seguenti priorità tra operatori
    1. gli operatori unari  $+$ ,  $-$  e le parentesi  $()$  hanno massima priorità
    2. poi vengono gli operatori  $*$ ,  $/$ ,  $\%$
    3. infine  $+$ ,  $-$  (somma e differenza)
  - gli operatori con la stessa priorità, si calcolano da sinistra e destra
  - l'ordine risultante è non ambiguo



## Vincoli dal linguaggio macchina

- circuiti diversi (operazioni diverse) per numeri interi e reali
- Cosa calcola un operatore dipende dal tipo degli operandi:
- divisione tra numeri reali:  $7.0/2.0 = 3.5$
- divisione tra interi,  $x$  e  $y$ . Esistono sempre  $a, b$  tali che



- l'aritmetica del C non è sempre uguale a come noi eseguiamo i conti (dobbiamo tenerne di conto). Es.

$$\frac{3}{2} * 2$$

- Come calcola l'espressione il C?

Ipotesi 1

$$\frac{3}{\cancel{2}} * \cancel{2} = 3 \text{ (si semplificano i 2)}$$

?

- l'aritmetica del C non è sempre uguale a come noi eseguiamo i conti (dobbiamo tenerne di conto). Es.

$$\frac{3}{2} * 2$$

- Come calcola l'espressione il C?

Ipotesi 1

$$\frac{3}{\cancel{2}} * \cancel{2} = 3 \text{ (si semplificano i 2)}$$

NO perché le espressioni non sono eseguite nell'ordine previsto:

$$(3/2)*2$$

- l'aritmetica del C non è sempre uguale a come noi eseguiamo i conti (dobbiamo tenerne di conto). Es.

$$\frac{3}{2} * 2$$

- Come calcola l'espressione il C?

Ipotesi 2

$$\left(\frac{3}{2}\right) * 2 = 1.5 * 2 = 3$$

?



- l'aritmetica del C non è sempre uguale a come noi eseguiamo i conti (dobbiamo tenerne di conto). Es.

$$\frac{3}{2} * 2$$

- Come calcola l'espressione il C?

Ipotesi 2

$$\left(\frac{3}{2}\right) * 2 = 1.5 * 2 = 3$$

NO perché  $3/2$  è un'operazione tra interi, non tra reali (i numeri non hanno la parte decimale, 3.0). L'espressione di seguito è corretta:

$$3.0/2.0*2.0=3$$

Quindi quanto fa  $3/2*2$ ?

- l'aritmetica del C non è sempre uguale a come noi eseguiamo i conti (dobbiamo tenerne di conto). Es.

$$\frac{3}{2} * 2$$

- Come calcola l'espressione il C?

Ipotesi 3

$$\left(\frac{3}{2}\right) * 2 = 1 * 2 = 2$$



- È possibile, anzi auspicabile, utilizzare parentesi non necessarie se queste migliorano la leggibilità del codice

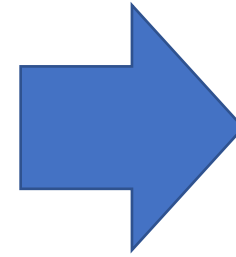
- In C gli operandi di un'espressione devono avere lo stesso tipo: cosa succede se proviamo a sommare un intero e un reale?

$$3.0 + 2 = ?$$

- Il C trasforma l'intero in un reale (è possibile forzare la trasformazione opposta, lo vedremo più avanti):

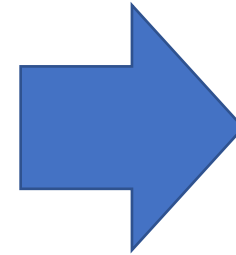
$$3.0 + 2.0 = 5.0$$

```
1  #include <stdio.h>
2
3  int main(void) {
4      |
5      printf("%d\n", 3/0);
6      |
7  }
```



gcc main.c

```
1  #include <stdio.h>
2
3  int main(void) {
4      |
5      printf("%d\n", 3/0);
6      |
7  }
```



gcc main.c



main.c:5:21: **warning:** division by zero is undefined [-Wdivision-by-zero]

```
printf("%d\n", 3/0);
```

^~

1 warning generated.

# Espressioni Condizioni (o Booleane)



- Espressioni condizionali (il risultato ha due valori possibili: Vero, Falso):

|    |                      |                |
|----|----------------------|----------------|
| == | uguale a             | 5 == 3 è Falso |
| >  | maggiore di          | 5 > 3 è Vero   |
| <  | minore di            | 5 < 3 è Falso  |
| != | diverso da           | 5 != 3 è Vero  |
| >= | maggiore o uguale di | 5 >= 3 è Vero  |
| <= | minore o uguale di   | 5 <= 3 è Falso |

< > <= >= hanno la stessa priorità (associatività a sinistra), maggiore di == e !=

- Il C di base non fornisce gli identificatori Vero, Falso (true, false)
  - sono però definiti nella libreria stdbool.h
- Il C usa la seguente convenzione
  - Falso corrisponde a `x==0`
  - Vero corrisponde a `x!=0` (se `y` è il risultato di un'espressione condizionale con valore true, allora `y` vale 1)
- Gli operatori non vengono generalmente combinati nella stessa espressione. Vediamo perché con un esempio:

`0<=x<8`

se `x==9` vale true!

- Il C usa la seguente convenzione
  - Falso corrisponde a  $x==0$
  - Vero corrisponde a  $x!=0$  (se  $y$  è il risultato di un'espressione condizionale con valore true, allora  $y$  vale 1)
- Gli operatori non vengono generalmente combinati nella stessa espressione. Vediamo perché con un esempio:

$$0 \leq x < 8$$

se  $x==9$  vale true! perché si eseguono i confronti da sinistra a destra:

$$(0 \leq 9) < 8 \rightarrow (1) < 8 \rightarrow 1 \text{ cioè true}$$

Con  $(0 \leq x < 8)$  intendiamo verificare se  $x \geq 0$  E se  $x < 8$

Per comporre condizioni utilizziamo degli operatori ad hoc



| Operatore    | Significato  | Esempio                    |
|--------------|--|----------------------------|
| && (binario) | AND logico. Vero solo se entrambi gli operandi sono Veri | ((5==5) && (5<2)) è Falso. |
| (binario)    | OR logico. Vero se almeno uno degli argomenti è Vero     | ((5==5)    (5<2)) è Vero   |
| ! (unario)   | NOT logico. Vero se l'argomento è Falso                  | !(5==5) è Falso            |

- Notate che, come per le espressioni, si possono usare le parentesi per indicare l'ordine di valutazione esplicitamente

# Precedenza e Associatività Operatori



| Riga | Precedenza (più in alto, maggior priorità) | Associatività        |
|------|--|----------------------|
| 1    | ()   | da sinistra a destra |
| 2    | <b>sizeof</b> + - ! (vedere nota riga 2)   | da destra a sinistra |
| 3    | * / %                                      | da sinistra a destra |
| 4    | + - (somma e differenza)                   | da sinistra a destra |
| 5    | < > <= >=                                  | da sinistra a destra |
| 6    | == !=                                      | da sinistra a destra |
| 7    | &&   | da sinistra a destra |
| 8    |  | da sinistra a destra |

Riga 2: +,- sono operatori unari, indicano il segno di un numero, ! è il not logico

- Usare più parentesi del necessario per migliorare la leggibilità è una buona norma di stile

- Poiché i valori di verità di un'espressione booleana sono 2, si possono elencare tutti i possibili input/output tramite una tabella (detta tabella di verità):

| a     | ! a   |
|-------|-------|
| vero  | falso |
| falso | vero  |

| a     | b     | a && b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | falso  |
| falso | vero  | falso  |
| falso | falso | falso  |

| a     | b     | a    b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | vero   |
| falso | vero  | vero   |
| falso | falso | falso  |

- Con lo stesso metodo, è possibile calcolare funzioni boolean complesse:

- Con lo stesso metodo, è possibile calcolare funzioni booleane complesse:

| <b>a    b</b> | <b>a &amp;&amp; b</b> | <b>!(a &amp;&amp; b)</b> | <b>(a    b)    !(a &amp;&amp; b)</b> |
|---------------|-----------------------|--------------------------|--------------------------------------|
| vero          | vero                  | falso                    | vero                                 |
| vero          | falso                 | vero                     | vero                                 |
| vero          | falso                 | vero                     | vero                                 |
| falso         | falso                 | vero                     | vero                                 |

- Con lo stesso metodo, è possibile calcolare funzioni booleane complesse:

| $a \vee b$ | $a \wedge b$ | $\neg(a \wedge b)$ | $(a \vee b) \vee \neg(a \wedge b)$ |
|------------|--------------|--------------------|------------------------------------|
| vero       | vero         | falso              | vero                               |
| vero       | falso        | vero               | vero                               |
| vero       | falso        | vero               | vero                               |
| falso      | falso        | vero               | vero                               |

- Con lo stesso metodo, è possibile calcolare funzioni booleane complesse:

| $a \vee b$ | $a \wedge b$ | $\neg(a \wedge b)$ | $(a \vee b) \vee \neg(a \wedge b)$ |
|------------|--------------|--------------------|------------------------------------|
| vero       | vero         | falso              | vero                               |
| vero       | falso        | vero               | vero                               |
| vero       | falso        | vero               | vero                               |
| falso      | falso        | vero               | vero                               |

- Con lo stesso metodo, è possibile calcolare funzioni booleane complesse:

| $a \vee b$ | $a \wedge b$ | $\neg(a \wedge b)$ | $(a \vee b) \vee \neg(a \wedge b)$ |
|------------|--------------|--------------------|------------------------------------|
| vero       | vero         | falso              | vero                               |
| vero       | falso        | vero               | vero                               |
| vero       | falso        | vero               | vero                               |
| falso      | falso        | vero               | vero                               |

- Con lo stesso metodo, è possibile calcolare funzioni booleane complesse:

| $a \vee b$ | $a \wedge b$ | $\neg(a \wedge b)$ | $(a \vee b) \vee \neg(a \wedge b)$ |
|------------|--------------|--------------------|------------------------------------|
| vero       | vero         | falso              | vero                               |
| vero       | falso        | vero               | vero                               |
| vero       | falso        | vero               | vero                               |
| falso      | falso        | vero               | vero                               |

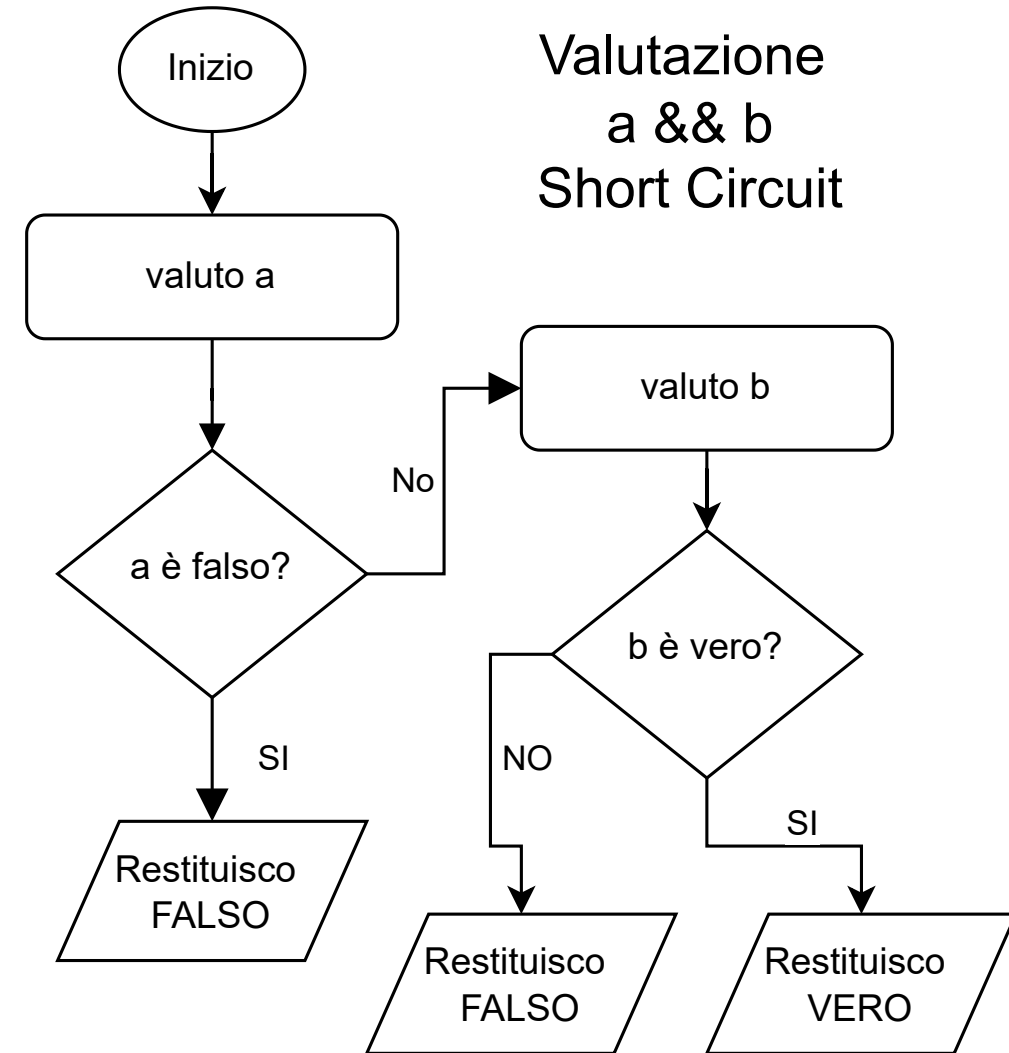


# Short Circuit Evaluation - AND



- Si valuta prima *a*; se *a* è falso si restituisce falso, cioè  $(a \ \&\& \ b) == 0$ , **senza valutare *b***
- Ci evitiamo di valutare *b*, che potrebbe essere un'espressione booleana lunghissima

| a     | b     | a && b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | falso  |
| falso | vero  | falso  |
| falso | falso | falso  |

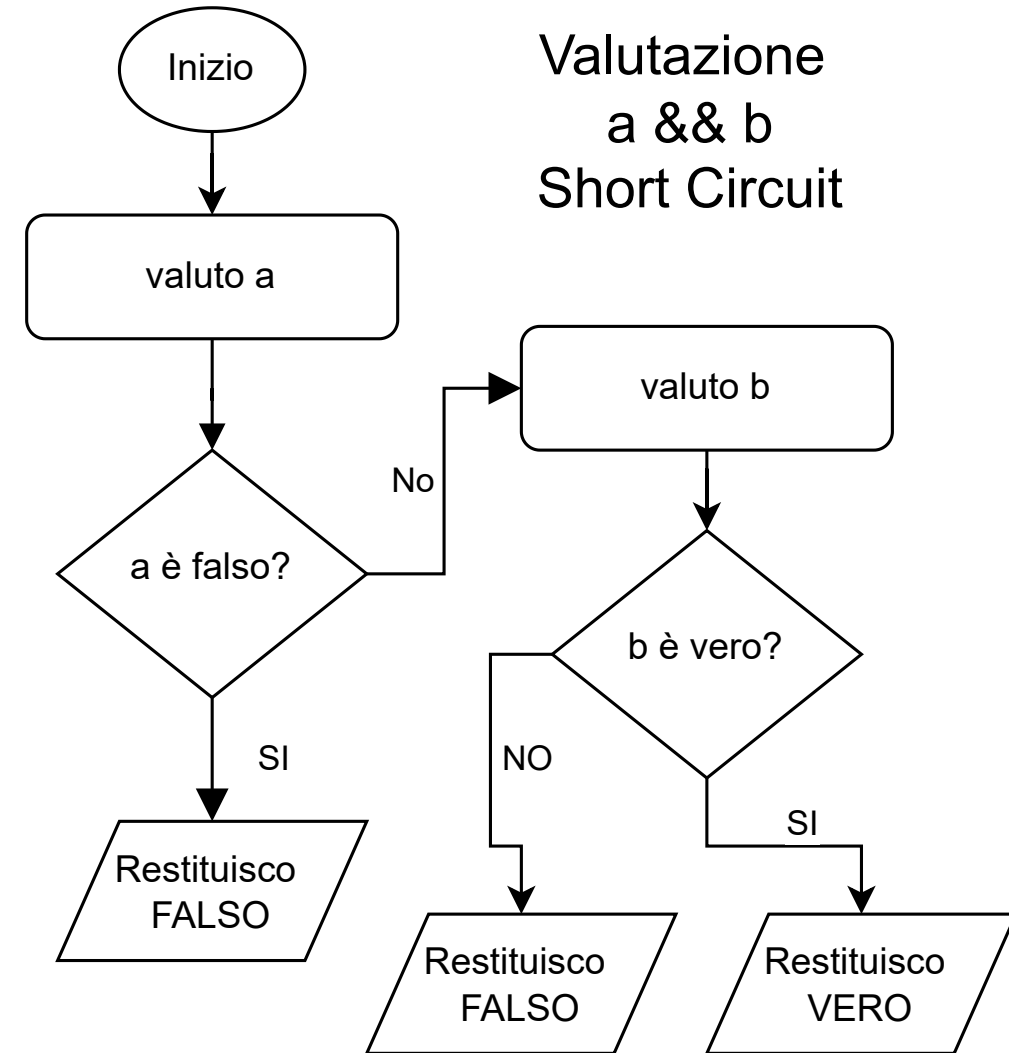


# Short Circuit Evaluation - AND



- si valuta prima a; se a è falso si restituisce falso, cioè  $(a \ \&\& \ b) == 0$ , **senza valutare b**
- se a è vero, valuto b: se è vero restituisco vero, se è falso restituisco falso

| a     | b     | a && b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | falso  |
| falso | vero  | falso  |
| falso | falso | falso  |

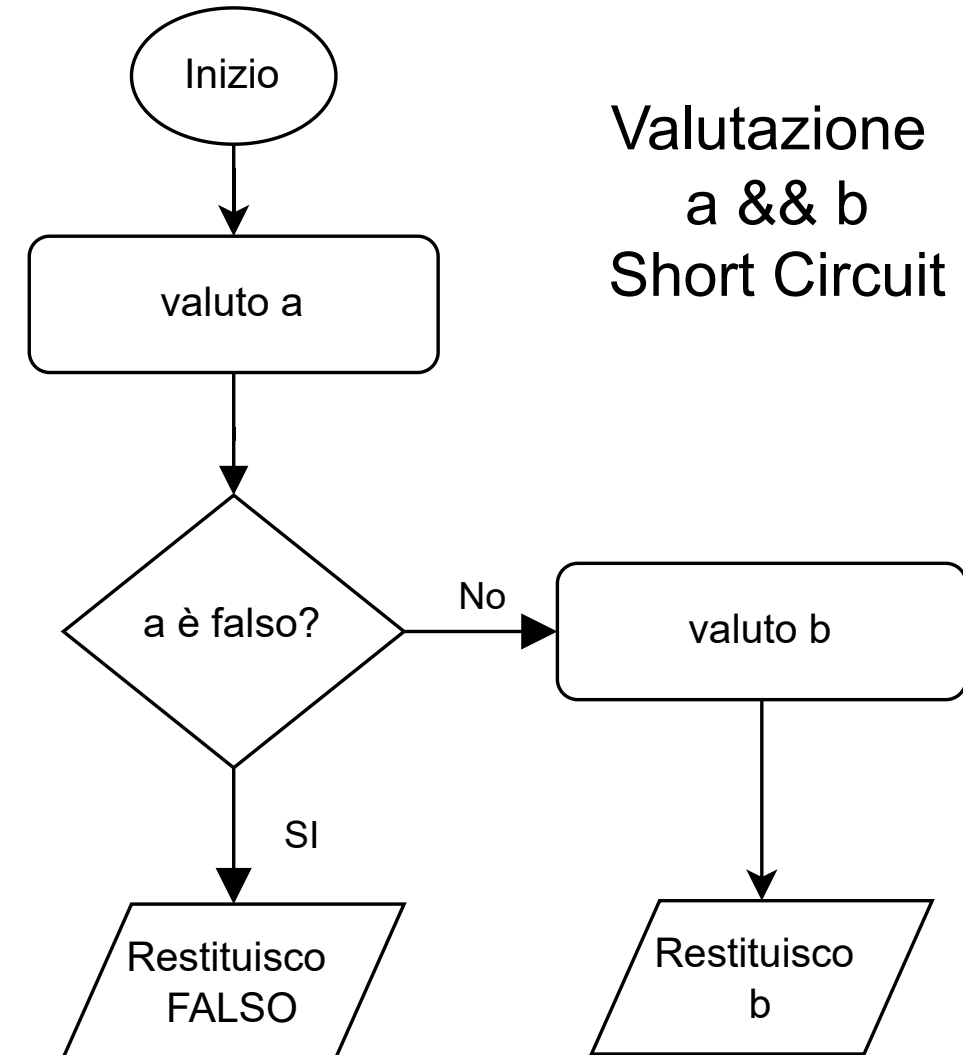


# Short Circuit Evaluation - AND



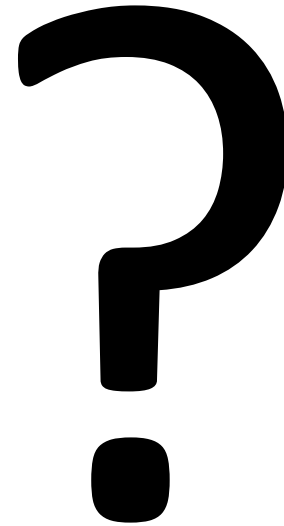
- si valuta prima a; se a è falso si restituisce falso, cioè  $(a \ \&\& \ b) == 0$ , **senza valutare b**
- se a è vero, restituisco il valore di verità di b

| a     | b     | a && b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | falso  |
| falso | vero  | falso  |
| falso | falso | falso  |



- Qual è l'algoritmo per calcolare la valutazione short-circuit per `||`?

| a     | b     | a    b |
|-------|-------|--------|
| vero  | vero  | vero   |
| vero  | falso | vero   |
| falso | vero  | vero   |
| falso | falso | falso  |

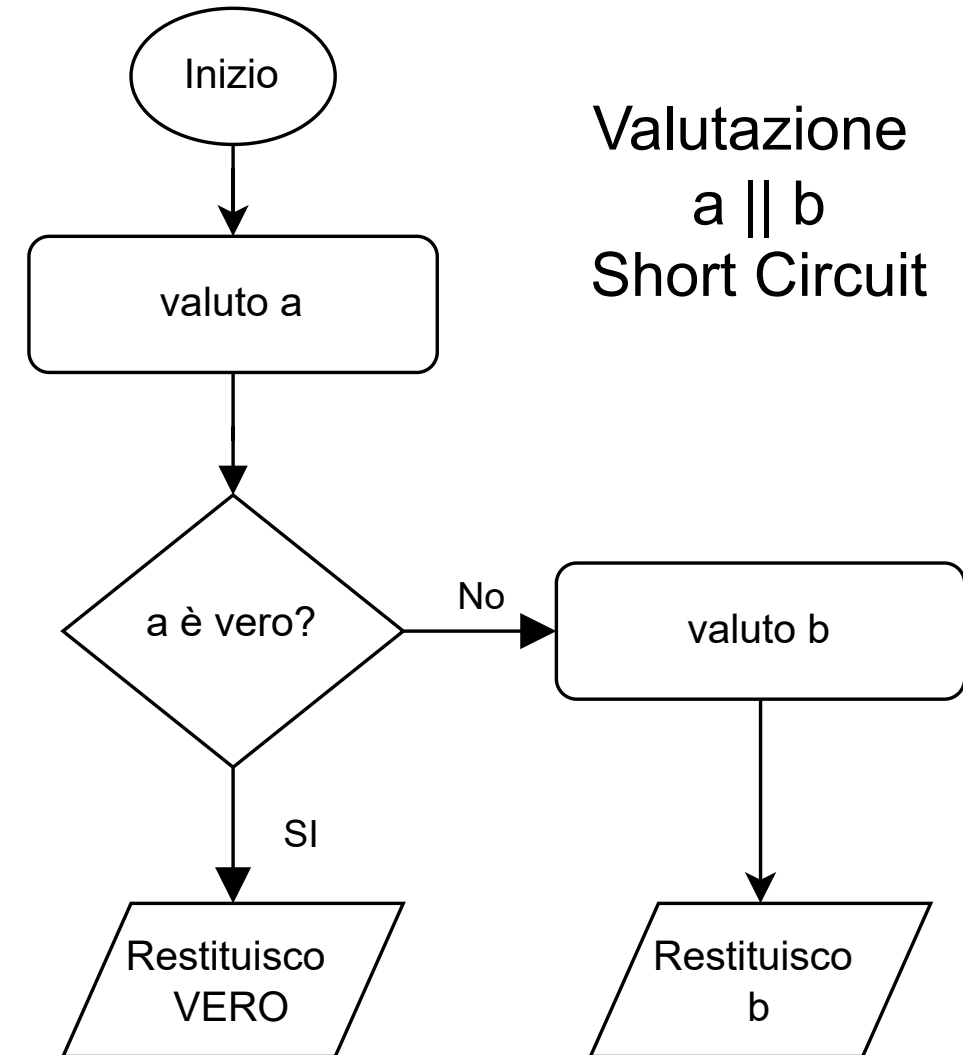


# Short Circuit Evaluation - OR



- si valuta prima a; se a è vero si restituisce vero, cioè  $(a \ || \ b) == 1$ , **senza valutare b**
- se a è falso, restituisco il valore di verità di b

| a     | b     | $a \    \ b$ |
|-------|-------|--------------|
| vero  | vero  | vero         |
| vero  | falso | vero         |
| falso | vero  | vero         |
| falso | falso | falso        |

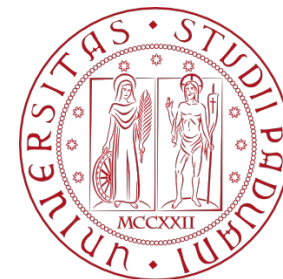


```
#include <stdio.h>

int main(void) {
    printf("%d\n", (3>0 || 3/0));
}
```

- A causa dello short-circuit questo codice compila ed esegue senza causare errori (3/0 viene ignorato dal compilatore), stampando 1

# Variabili parte 1



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Come in matematica, le espressioni e le condizioni possono essere generalizzate utilizzando simboli (variabili) al posto di alcuni valori, ad es.  $x*2$  generalizza  $2*2$ ,  $3*2$ ,...
- Una variabile ha i seguenti attributi:
  - il nome (che definiamo noi)
  - l'area di memoria in cui è mantenuto il suo valore (non assegnata dall'utente)
  - il tipo: le variabili vengono usate per rappresentare numeri interi, reali, caratteri (in C è definito dall'utente).
    - Il tipo è un modo conciso per dire quanta memoria occupa (dipende dall'architettura della macchina), come leggere o scrivere la sequenza di bit e quali operazioni posso fare con quella variabile.

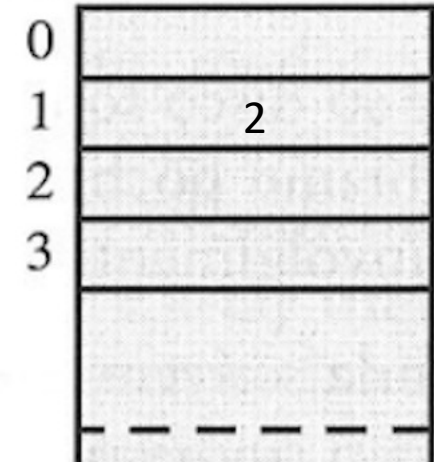


# Variabili ed Assegnamento



| Variabile   |   |  |
|-------------|---|--|
| nome e tipo | L-valore<br>Identificativo dell' area di memoria riservata alla variabile | R-valore<br>il contenuto corrente della cella di memoria |

- L'operazione di assegnamento = permette di modificare il contenuto (valore) di una variabile:
- $y = E$ ; // la parte a sinistra di  $=$  deve restituire un L-valore, la parte a destra un R-valore:
  - $y = E \rightarrow$  vai alla cella di memoria indicata dall' L-valore di  $y$  e scrivici dentro il risultato della valutazione dell'espressione  $E$
  - i tipi di  $y$  ed  $E$  devono essere compatibili (eventualmente il compilatore effettua una conversione automatica, es. se  $y$  è float e  $E$  risulta in un int)
- $y = 2$ ; // vai alla cella di memoria indicata dall' L-valore di  $y$  e scrivici dentro il risultato dell'espressione alla destra dell'uguale, ovvero 2



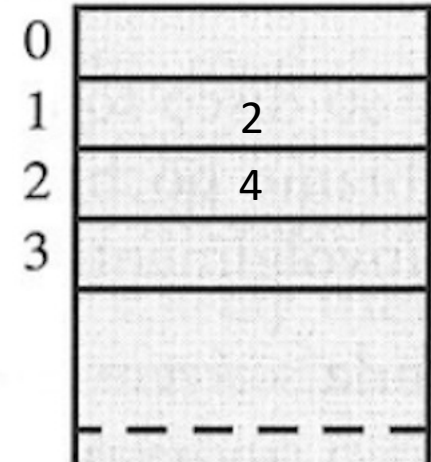
$y$ : L-valore=1  
R-valore=2

# Variabili ed Assegnamento



| Variabile   |   |  |
|-------------|---|--|
| nome e tipo | L-valore<br>Identificativo dell' area di memoria riservata alla variabile | R-valore<br>il contenuto corrente della cella di memoria |

- $y = 2;$
- Notate che l'attributo selezionato della variabile (L o R valore) dipende da dove essa compare nell'istruzione di assegnamento:
- $x = y + 2;$  // vai alla cella di memoria indicata dall' L-valore di x e scrivici dentro il risultato dell'espressione alla destra dell'uguale, ovvero il risultato della somma tra 2 e l'R-valore della variabile y:  $x=2+2=4$
- $x = x + 1$  // ?



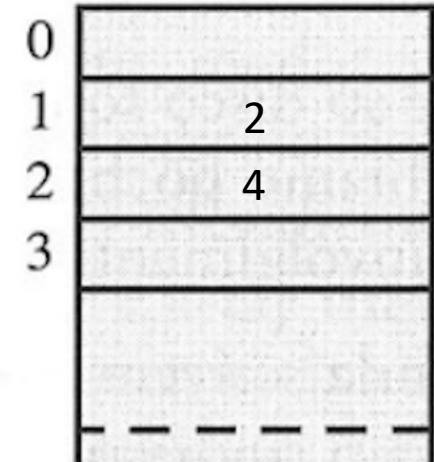
x: L-valore=2  
R-valore=4

# Variabili ed Assegnamento



| Variabile   |   |  |
|-------------|---|--|
| nome e tipo | L-valore<br>Identificativo dell' area di memoria riservata alla variabile | R-valore<br>il contenuto corrente della cella di memoria |

- $y = 2;$
- Notate che l'attributo selezionato della variabile (L o R valore) dipende da dove essa compare nell'istruzione di assegnamento:
- $x = y + 2;$  // vai alla cella di memoria indicata dall' L-valore di x e scrivici dentro il risultato dell'espressione alla destra dell'uguale, ovvero il risultato della somma tra 2 e l'R-valore della variabile y:  $x=2+2=4$
- $x = x + 1$  //  $x=4+1=5$



x: L-valore=2  
R-valore=4

Se non pretendiamo di essere troppo rigorosi, possiamo immaginarci che l'assegnamento

```
float x=2+5*3;
```

sia tradotto come segue dal compilatore

```
//valuta E=2+5*3
```

```
MoltInteri(5,3,R1);
```

```
SommaInteri(R1, 2, R2);
```

```
//valutazione di E terminata
```

```
//E ed x hanno tipi diversi: intero e reale (la conversione viene fatta
```

```
// quando l'operazione corrente lo richiede, non prima)
```

```
ConvertiAfloat(R2,R3);
```

```
Store(R3, Lvalore(x)); // Lvalore(x) è l'indirizzo in memoria della variabile x
```

- In C è necessario dichiarare le variabili prima di usarle
  - `int x; // dichiara una variabile di tipo intero`  
    `// riserva 4-8 byte di memoria per una variabile di nome x`
  - `int x = 2; // dichiara una variabile di tipo intero ed inizializza il suo valore a 2`
- Non si deve mai utilizzare una variabile prima di averle assegnato un valore:

```
int x; int y;
```

```
y = x+2; //non sappiamo che valore abbia x!
```

- Un legame tra una variabile ed un suo attributo si dice statico se è stabilito prima dell'esecuzione e non può essere cambiato in seguito, dinamico altrimenti:
  - il valore è un legame dinamico
  - In C il tipo è un legame statico (questo implica che il compilatore può identificare i seguenti tipi di errore: `int x; x = "Ciao Mondo!"`;

- In C è possibile definire “variabili il cui valore è un legame statico”, quelle che comunemente chiamiamo costanti (es. pi greco)
  - `const int x = 3; //` poiché non possiamo cambiare `x`, dobbiamo definirne il valore quando dichiariamo la variabile
- L’assegnamento ha un effetto collaterale:
  - `x=8`, oltre ad assegnare 8 alla variabile `x`, restituisce anche 8
- l’assegnamento ha bassa priorità come operatore (ma è pur sempre utilizzabile in un’espressione)
- `4+(x=8)` restituisce 12

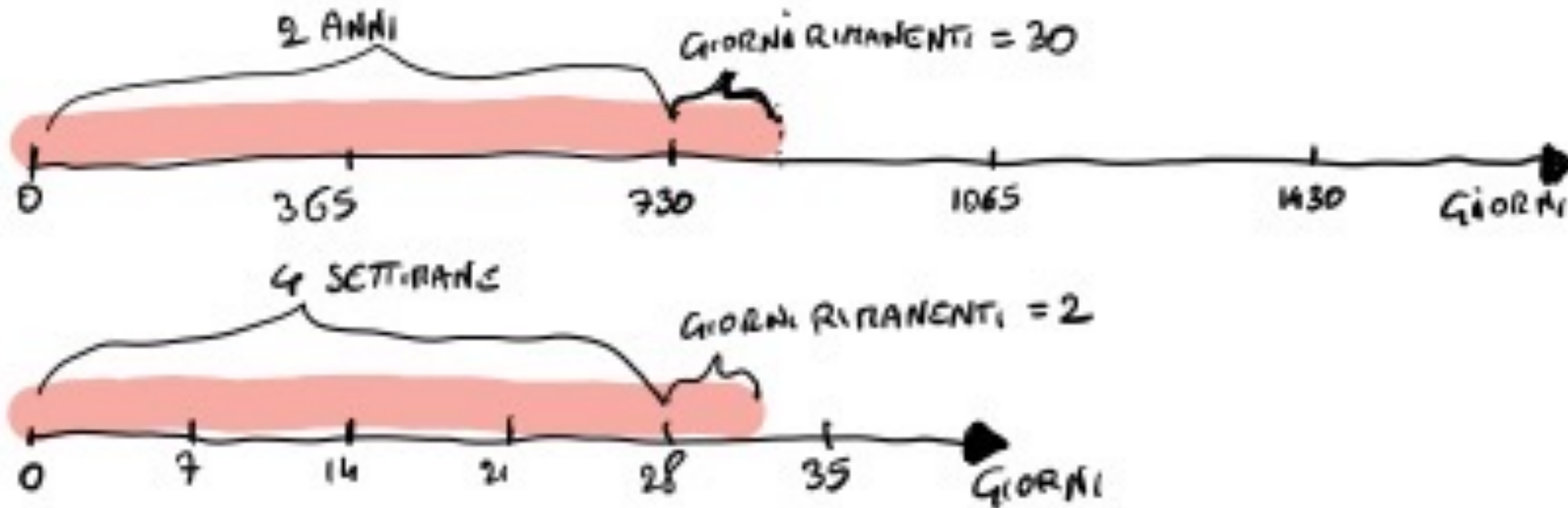


- Nomi di variabili:
  - usiamo caratteri alfanumerici (a-zA-Z0-9 e \_)
  - ma il nome non deve iniziare con 0-9 e \_,
  - il C è case sensitive (ma evitiamo di avere due variabili di nome VAR e var)
  - evitiamo anche di avere variabili che assomigliano ad un comando o ad un elemento del linguaggio: IF, INT
- i nomi delle variabili devono essere il più possibile indicativi della loro funzione
  - ma evitate nomi troppo lunghi

- Scrivere un programma per convertire un numero di giorni  $x$  in anni, settimane, giorni. Stampare " $x$  giorni corrispondono ad anni  $y$ , settimane  $w$ , giorni  $z$ ", dove  $x, y, w, z$  sono i giorni in input e gli anni, settimane e giorni calcolati.
- Per esempio se  $x=760$  stamperemmo "760 giorni corrispondono ad anni 2, settimane 4, giorni 2".
- Assumere che un anno sia formato da 365 giorni.



INPUT DAYS = 760



760 GIORNI CORRISPONDONO A 2 ANNI, 4 SETTIMANE, 2 GIORNI

1. Contare quanti gruppi da 365 posso creare con i giorni in input
  - quanti anni stanno nei giorni in input
2. Calcolare i giorni che avanzano
3. Contare quanti gruppi da 7 posso creare con i giorni che avanzano
4. Calcolare i giorni che avanzano da quest'ultimo raggruppamento
5. Stampare il risultato dei calcoli

```
#include <stdio.h>

int main() {
    int input_days = 760;
    int giorniRimanenti; /* giorni rimanenti dopo aver suddiviso input_days in anni */
    int anni, settimane, giorni;
    /* PRE: input_days >= 0
       POST: anni, settimane, giorni sono gli anni corrispondenti a input_days */
    anni = input_days / 365;
    giorniRimanenti = input_days % 365;
    settimane = giorniRimanenti / 7;
    giorni = giorniRimanenti % 7;

    printf("%d giorni corrispondono ad anni %d, settimane %d, giorni %d\n", input_days, anni, settimane, giorni);
}
```

Trasformare il valore in gradi fahrenheit della variabile fahrenheit (X) nel corrispondente valore celsius (Y) arrotondato all'intero inferiore e stampare "X gradi fahrenheit corrispondono a Y gradi celsius"

Ad esempio se fahrenheit=78 stampa

78 gradi fahrenheit corrispondono a 25 gradi celsius

Si ricorda che  $\text{celsius} = (5/9)(\text{fahrenheit} - 32)$

Trasformare il valore in gradi fahrenheit della variabile fahrenheit (X) nel corrispondente valore celsius (Y) ~~arrotondato all'intero inferiore~~ e stampare "X gradi fahrenheit corrispondono a Y gradi celsius"

Ad esempio se fahrenheit =78 stampa

78 gradi fahrenheit corrispondono a 25.5556 gradi celsius

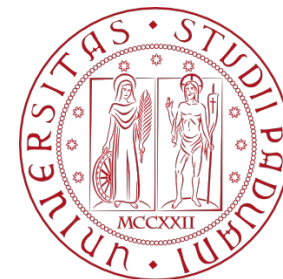
Si ricorda che  $\text{celsius} = (5/9)(\text{fahrenheit} - 32)$

- Per gli interi abbiamo già visto come dichiarare diversi tipi di interi

| Nome tipo in  | Descrizione                       | Byte | Valore Min | Valore Max | formato in printf |
|---------------|-----------------------------------|------|------------|------------|-------------------|
| int           | intero                            | 4    | INT_MIN    | INT_MAX    | printf("%d", x)   |
| long          | intero che usa il doppio dei byte | 8    | LONG_MIN   | LONG_MAX   | printf("%ld", x)  |
| short         | intero che usa la metà dei byte   | 2    | SHRT_MIN   | SHRT_MAX   | printf("%hd", x)  |
| unsigned int  | un intero positivo                | 4    | 0          | UINT_MAX   | printf("%u", x)   |
| unsigned long | un long positivo                  | 8    | 0          | ULONG_MAX  | printf("%lu", x)  |

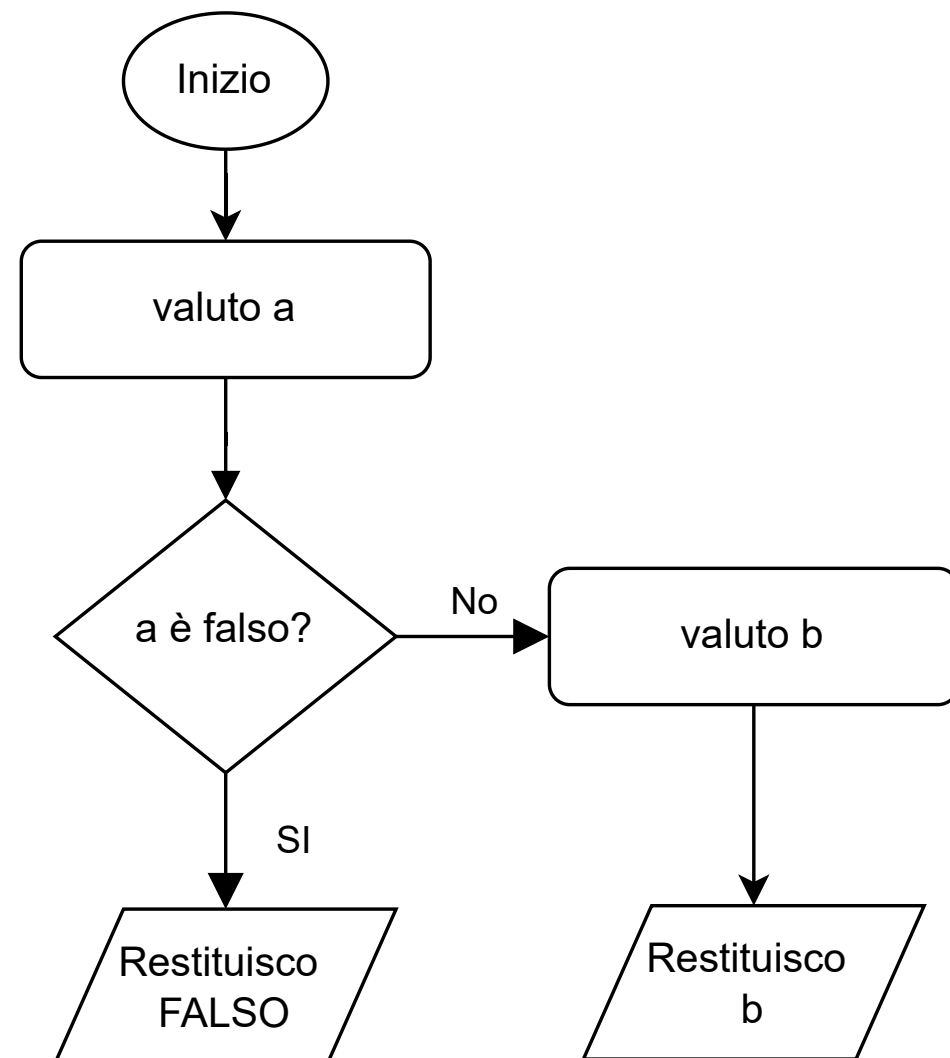
- Per i reali abbiamo 2 opzioni
  - float o double (il secondo utilizza il doppio della memoria del primo)

# Comandi di Scelta



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Nell'esempio a destra, una volta arrivati al rombo, prendiamo due strade diverse a seconda che `a==true` o `a==false`
- in altre parole il C ha un equivalente del rombo, implementato con le istruzioni `Salto()` e `SaltaSeUguale()` del nostro linguaggio macchina?





# IF: Definizione

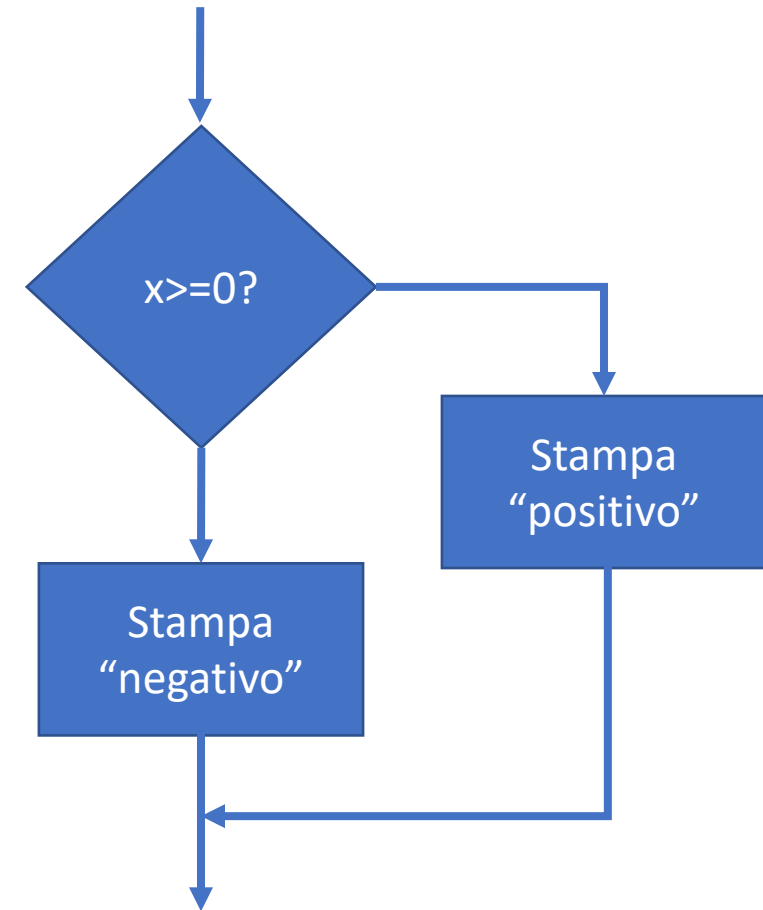


```
if (condizione) {  
    //comandi da eseguire se la condizione è vera  
} else {  
    //comandi da eseguire se la condizione è falsa  
}  
  
// questa parte di codice viene eseguita indipendentemente  
dal valore di condizione
```

Esempio:

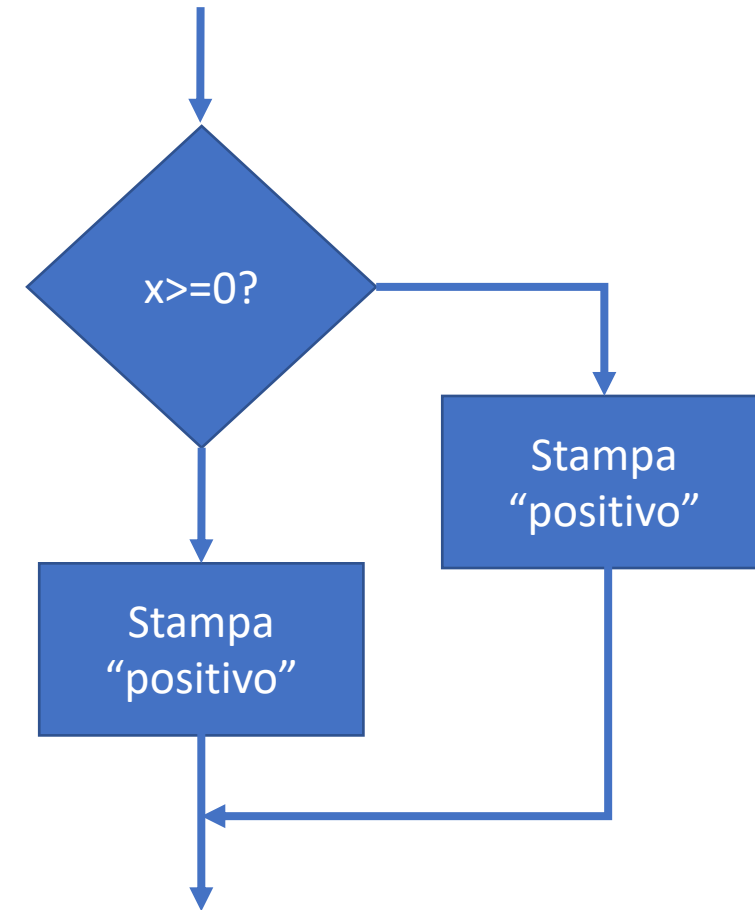
```
if (x >= 0) {  
    printf("positivo");  
} else {  
    printf("negativo");  
}
```

}  
mutualmente  
esclusive



```
if (condizione) {  
    //comandi da eseguire se la condizione è vera  
} else {  
    //comandi da eseguire se la condizione è falsa  
}
```

- *condizione* può essere un'espressione logica complicata a piacere, basta che restituisca un valore di verità
- *condizione* deve essere racchiusa tra parentesi tonde
- I simboli {} definiscono una sequenza di comandi (blocco). Notate che non terminano con ;

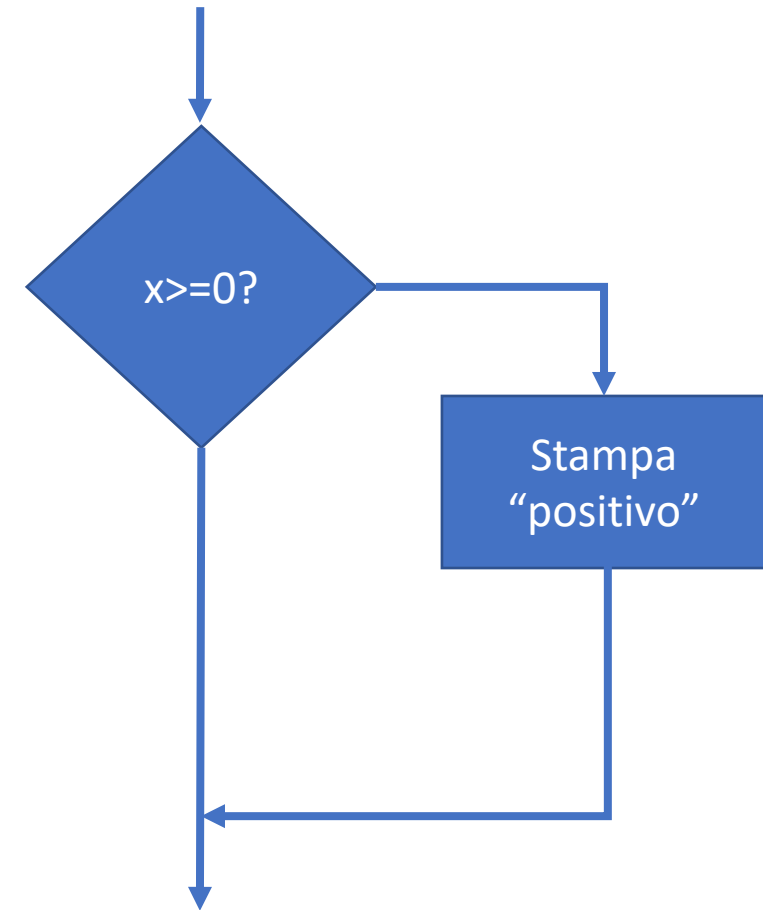


```
if (condizione) {  
    //comandi da eseguire se la condizione è vera  
}  
//comando2
```

Esempio:

```
if (x >= 0) {  
    printf("positivo");  
}  
//comando2
```

l' else non deve necessariamente esserci



- I simboli {} definiscono una sequenza di comandi (blocco).
- Se ho un solo comando da eseguire, posso omettere {}

```
if (!x)
    printf("x è 0");
else {
    printf("il numero non è ");
    printf("zero");
}
```

MA

```
if (condizione1)  
    if (condizione2)  
        comando1;  
else  
    comando2;
```

Senza {} l'else fa riferimento all'if più vicino (*condizione2*), quindi per leggibilità può essere utile a volte utilizzare {} anche quando c'è un solo comando nel blocco

```
if (condizione1) {  
    //blocco1: da eseguire se condizione1 è vera  
} else if (condizione2) {  
    //blocco2: da eseguire se condizione2 è vera  
} else if (condizione3) {  
    //blocco3: da eseguire se condizione3 è vera  
} else {  
    //blocco4:comandi da eseguire se condizion1-3 sono false  
}  
//comando2
```

È possibile comporre comandi if, notate che solo uno tra blocco1,...,blocco4 viene eseguito.

```
if (condizione1) {  
    //blocco1: da eseguire se condizione1 è vera  
} else { if (condizione2) {  
    //blocco2: da eseguire se condizione2 è vera  
} else { if (condizione3) {  
    //blocco3: da eseguire se condizione3 è vera  
} else {  
    //blocco4:comandi da eseguire se condizion1-3 sono false  
}  
}  
}  
}  
//comando2
```

Stesso comando if con {} aggiunte

```
if (condizione1) {  
    if (condizione2) {  
        //da eseguire se condizione1 e condizione2 sono vere  
    } else {  
        //da eseguire se condizione1 è vera e condizione2 è falsa  
    }  
} else {  
    //blocco 4:comandi da eseguire se condizion1 è falsa  
}  
//comando2
```

È possibile utilizzare if uno dentro all'altro



- Esecuzione condizionale all'interno di un'espressione:  
condizione? valore\_se\_vero: valore\_se\_falso (all'interno di un'espressione)

int x = -2, y; //si possono dichiarare più variabili separandole da virgole

y = 3+(x>0?x:-x); // y=5

// se x>0 calcola 3+x, altrimenti 3-x

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

```
int main () {
```

```
    int a=3;
```

```
    if (a==5); {
```

```
        printf("il valore di a è 5\n");
```

```
    }
```

```
}
```

cosa stampa?

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

```
int main () {
```

```
    int a=3;
```

```
    if (a==5);
```

```
{
```

```
    printf("il valore di a è 5\n");
```

```
}
```

```
}
```

stampa "il valore di a è 5". L'if ha come comando ; (ovvero il comando vuoto) se la condizione è vera. L'istruzione tra {} viene eseguita perciò in ogni caso (l'if è completamente terminato a quel punto)

```
#include <stdio.h>

int main () {

    int a=3;
    if (a=5) {
        printf("il valore di a è 5\n");
    }

}
```



poiché = può essere usato in un'espressione `a=5` è sintatticamente corretto, ma il corpo dell'`if` viene eseguito indipendentemente dal valore che `a` aveva prima di valutare la condizione dell'`if` (ed `a` assume il valore 5).

- Dato il programma a fianco
  - Riempite la tabella di verità sotto (esco==vero se il codice stampa “esco”, falso altrimenti)
  - Implementate un programma equivalente che eviti di ripetere due volte l’istruzione printf(“esco\n”);

| Piove | ho L’ombrello | stampa esco |
|-------|---------------|-------------|
| falso | falso         |             |
| falso | vero          |             |
| vero  | falso         |             |
| vero  | vero          |             |

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

```
int main () {
```

```
    int piove = 0;
```

```
    int ho_ombrello = 1;
```

```
    if(!piove) {
```

```
        printf("esco\n");
```

```
    } else if (ho_ombrello) {
```

```
        printf("esco\n");
```

```
    } else {
```

```
        printf("sto a casa\n");
```

```
    }
```

```
}
```

# Esercizio - Soluzione



```
#include <stdio.h>

int main () {
    int piove = 0;
    int ho_ombrello = 1;

    if(piove && !ho_ombrello) {
        printf("sto a casa\n");
    } else {
        printf("esco\n");
    }
}
```

| Piove | ho L'ombrello | Esco  |
|-------|---------------|-------|
| falso | falso         | vero  |
| falso | vero          | vero  |
| vero  | falso         | falso |
| vero  | vero          | vero  |

```
#include <stdio.h>

int main () {

    int piove = 0;
    int ho_ombrello = 1;

    if(!piove) {
        printf("esco\n");
    } else if (ho_ombrello) {
        printf("esco\n");
    } else {
        printf("sto a casa\n");
    }

}
```

```
/*  
Date 3 variabili intere: x,y,z, stampare il valore minore tra le 3.  
Es. se x=5,y=2,z=7 stampa  
"Il minore dei tre valori è 2  
"  
*/  
  
#include <stdio.h>  
  
int main() {  
  
    int x=5, y=2, z=7;  
    printf("Il minore dei tre valori è ");  
    if (...  
  
}
```

# Esercizio - Soluzione



```
#include <stdio.h>
int main() {
    int x=5, y=2, z=7;
    printf("Il minore dei tre valori è ");
    if (x < y) {
        if (x < z) {
            printf("%d\n", x);
        } else {
            printf("%d\n", z);
        }
    } else {
        if (y < z) {
            printf("%d\n", y);
        } else {
            printf("%d\n", z);
        }
    }
}
```



- In molti casi si presenta la necessità di ripetere più volte una serie di istruzioni
  - es. Stampare "Ciao Mondo!" molte volte
- Vogliamo evitare di riscrivere le stesse istruzioni
  - Se troviamo un errore nelle istruzioni ripetute, dobbiamo correggerle più volte
  - Molte volte non sappiamo a priori quante volte dobbiamo ripetere una sequenza di istruzioni!
- Vorremmo uno schema come quello a fianco: ripetiamo una sequenza di istruzioni finché una condizione è verificata



```
while (condizione) {  
    //comandi da eseguire se la condizione è vera  
}  
comando2
```

Il comando while:

1. se *condizione* è falsa, non esegue i comandi all'interno del blocco e passa a comando2
2. se *condizione* è vera, esegue i comandi all'interno de blocco
3. Una volta eseguiti i comandi del blocco, ritorna al punto 1

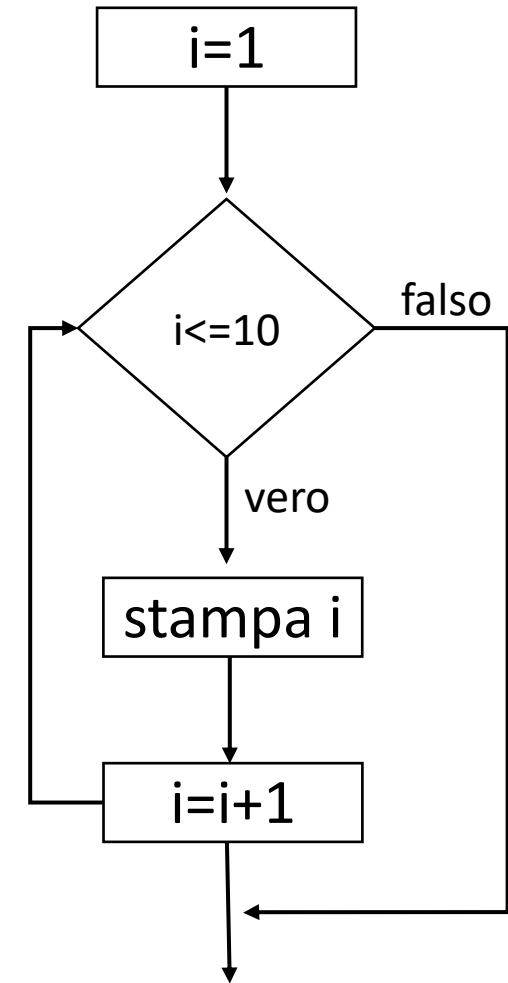


- I costrutti iterativi consistono dei seguenti elementi
- un comando di inizializzazione: indica le condizioni iniziali delle variabili coinvolte nel ciclo
- una condizione di continuazione che indica se l'esecuzione del ciclo debba continuare
- il corpo del ciclo: la sequenza da istruzioni da ripetere potenzialmente più volte
- istruzione iterativa: fa sì che il ciclo prosegua verso la terminazione

# Iterazione: Esempio



- Stampare i numeri da 1 a 10
1. inizializzare una variabile, es.  $i$ , ad 1.
  2. finché  $i$  è minore o uguale a 10
  3. stampa  $i$
  4. incrementa  $i$  di 1
  5. ritorna al punto 2



- Stampare i numeri da 1 a 10
1. inizializzare una variabile, es. i, ad 1.
  2. finché i è minore o uguale a 10
  3. stampa i
  4. incrementa i di 1
  5. ritorna al punto 2

```
int i=1;  
while(i<=10) {  
    printf("%d\n", i);  
    i=i+1;  
}
```

- Se si rimuove l'istruzione `i=i+1`; cosa succede?

?

```
int i=1;  
while(i<=10) {  
    printf("%d\n", i);  
    i=i+1;  
}
```

- Se si rimuove l'istruzione  $i=i+1$ ;
- l'esecuzione del ciclo non termina mai perché  $i$  è sempre uguale a 1 e perciò la condizione  $i \leq 10$  è sempre vera
- il codice è sintatticamente corretto per cui possiamo accorgerci dell'errore solamente durante l'esecuzione
- Utilizzare Ctrl-c (control-c) per forzare la terminazione del programma

```
int i=1;
while(i<=10) {
    printf("%d\n", i);
    i=i+1;
}
```

/\* . Scrivere un programma che stampi x volte "Ciao Mondo!"

Es. se  $x = 3$

Ciao Mondo!

Ciao Mondo!

Ciao Mondo!

\*/



/\* . Scrivere un programma che stampi x volte "Ciao Mondo!"

Es. se x = 3

Ciao Mondo!

Ciao Mondo!

Ciao Mondo!

\*/

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

Esiste un comando equivalente al while ma con una sintassi differente: il for

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

```
#include <stdio.h>

int main() {

    int i;
    for(i=1; i<=5; i=i+1) {
        printf("Ciao Mondo!\n");
    }

}
```

Esiste un comando equivalente al while ma con una sintassi differente: il for

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

```
#include <stdio.h>

int main() {

    int i;
    for(i=1; i<=5; i=i+1) {
        printf("Ciao Mondo!\n");
    }

}
```

Esiste un comando equivalente al while ma con una sintassi differente: il for

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

```
#include <stdio.h>

int main() {

    int i;
    for(i=1; i<=5; i=i+1) {
        printf("Ciao Mondo!\n");
    }

}
```

Esiste un comando equivalente al while ma con una sintassi differente: il for

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

```
#include <stdio.h>

int main() {

    int i;
    for(i=1; i<=5; i=i+1) {
        printf("Ciao Mondo!\n");
    }

}
```

Esiste un comando equivalente al while ma con una sintassi differente: il for

```
#include <stdio.h>

int main() {

    int i=1;
    while(i<=5) {
        printf("Ciao Mondo!\n");
        i=i+1;
    }

}
```

```
#include <stdio.h>

int main() {

    int i;
    for(i=1; i<=5; i=i+1) {
        printf("Ciao Mondo!\n");
    }

}
```

```
// inizializzazione: es. i = 0  
while (condizione: es. i < 10) {  
    //sequenza di comandi  
    //assegnamento: es. i = i + 1;  
}
```



```
for(inizializzazione; condizione; assegnamento) {  
    //sequenza di comandi;  
}
```

- for e while sono equivalenti, in alcuni contesti è più naturale usare uno o l'altro, ma potete usare sempre uno solamente.

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

```
int main () {
```

```
    int a=1;
```

```
    while (a<5); {
```

```
        printf("il valore di a è %d\n", a);
```

```
        i = i+1;
```

```
    }
```

```
}
```

come per l'if, il ; dopo la condizione fa sì che il while non esegua il comando vuoto (;) all'infinito



- È utile chiedersi:
  - Quante volte viene eseguito il corpo di un ciclo?
  - Quale valore ha la variabile di iterazione (a) la prima volta che viene eseguito il corpo del ciclo?
  - Quale valore ha l'ultima volta?
  - Quale valore assume dopo l'uscita del ciclo?

```
int a=0;
while (a<5); {
    printf("il valore di a è %d\n", a);
    a = a+2;
}
```

- nel corpo di un ciclo è possibile avere qualsiasi comando, tra cui un altro ciclo.  
Es. stampare le tabelline
- Tabellina di x=2
- Il codice a fianco stampa:  
2 4 6 8 10 12 14 16 18 20
- Se vogliamo stampare le tabelline per più valori? Es. 2-5

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50

```
int x=2,i;  
  
for(i=1; i<=10; i=i+1) {  
    printf(" %d", x*i);  
}  
printf("\n");
```

- Se vogliamo stampare le tabelline per più valori? Es. 2-5

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50

```
int x,i;

for(x=2; x<=5; x=x+1) {
    for(i=1; i<=10; i=i+1) {
        printf(" %d", x*i);
    }
    printf("\n");
}
```

# Esercizio: Somma 1..n



/\* Calcolare la somma dei primi n numeri \*/

- $n=5$ :  $1+2+3+4+5=15$
- possiamo utilizzare un ciclo per iterare sui valori tra 1 ed n
- come facciamo per calcolare la somma?
- possiamo creare una variabile somma la quale, all' $i$ -esima iterazione del ciclo, contenga la somma dei primi  $i$  numeri
- ovvero che sommi tutti i valori  $i$  a mano che li incontra nel ciclo

| i     | 1 | 2 | 3 | 4  | 5  |
|-------|---|---|---|----|----|
| somma | 1 | 3 | 6 | 10 | 15 |

se  $i=3$

$$\text{somma}_{i=3} = 6 = 3 + (2+1)$$

$$\text{somma}_{i=3} = 3 + \text{somma}_{i=2}$$

$$\text{somma}_i = i + \text{somma}_{i-1}$$

# Esercizio: Somma 1..n



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

```
int main() {
```

```
    int n=5, i, somma=0;
```

```
    for(i=1; i<=n; i+=1) {
```

```
        somma = somma + i;
```

```
    }
```

```
    printf("la somma tra 1 ed %d è %d\n", n, somma);
```

```
}
```

# Esercizio: Somma numeri pari tra 1 e n



- Calcolare la somma dei numeri pari tra 1 ed n e stamparla a video. Riporto il codice per la somma tra 1..n, come possiamo modificarlo per il nuovo problema?

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

```
int main() {
```

```
    int n=8, i, somma=0;
```

```
    for(i=1; i<=n; i+=1) {
```

```
        somma = somma + i;
```

```
    }
```

```
    printf("Somma pari tra 1 ed %d è %d\n", n, somma);
```

```
}
```

# Esercizio: Somma numeri pari tra 1 e n



- Calcolare la somma dei numeri pari tra 1 ed n e stamparla a video.

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

```
int main() {
```

```
    int n=8, i, somma=0;
```

```
    for(i=2; i<=n; i+=2) {
```

```
        somma = somma + i;
```

```
    }
```

```
    printf("Somma pari tra 1 ed %d è %d\n", n, somma);
```

```
}
```

# Esercizio: Prodotto 1..n



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

```
int main() {
```

```
    int n=5, i, prod;
```

```
    printf("Prodotti tra 1 ed %d = %d\n", n, prod);  
}
```



# Esercizio: Prodotto 1..n



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

```
int main() {
```

```
    int n=5, i, prod=1;
```

```
    for(i=1; i<=n; i+=1) {
```

```
        prod = prod * i;
```

```
    }
```

```
    printf("Prodotti tra 1 ed %d = %d\n", n, prod);  
}
```

```
/*
```

Stampare le prime  $n$  potenze di 2, ovvero  $2^0, 2^1, \dots, 2^{n-1}$ .

Ad esempio se  $n=5$  stampa:

1 2 4 8 16

```
*/
```

```
/*
```

Stampare le prime n potenze  
di 2, ovvero  $2^0, 2^1, \dots, 2^{n-1}$ .

Ad esempio se  $n=5$  stampa:

1 2 4 8 16

```
*/
```

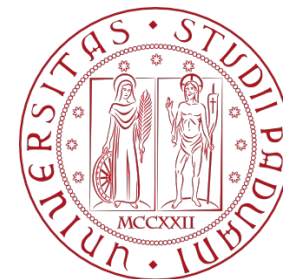
```
#include <stdio.h>
```

```
int main() {
```

```
    int n=5, i, power=1;
    for(i=0; i<n; i+=1) {
        printf(" %d", power);
        power = power*2;
    }
    printf("\n");
}
```

# Variabili parte 2

## Visibilità



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Non vogliamo riferire celle di memoria RAM tramite il loro indirizzo numerico nel nostro codice,
- perché il nostro codice deve essere più possibile indipendente dalla macchina
  - non vogliamo riscriverlo per due macchine diverse.
- Idealmente possiamo riferirci ad una variabile con un nome se teniamo corrispondenza tra nome-indirizzo (ed il tipo)

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $l_1$         | int   |
| y    | $l_2$         | float |

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $l_1$         | int   |
| y    | $l_2$         | float |

- Concettualmente\* possiamo pensare che Il compilatore faccia questo per noi in modo automatico, ovvero
- quando dichiariamo una variabile, viene creata una riga della tabella sopra
- Questo ci permette di utilizzare nomi nel nostro codice invece di indirizzi RAM
  - i nomi vengono risolti in indirizzi andandoli a cercare nella tabella sopra

\*in realtà il compilatore fa una cosa molto più efficiente ed ottimizzata (al termine della compilazione fa una passata del codice sostituendo ogni nome con l'indirizzo corrispondente), ma concettualmente è corretto pensare alla tabella sopra.

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $I_1$         | int   |
| y    | $I_2$         | float |

- Esistono svantaggi ad utilizzare nomi invece di indirizzi?

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $l_1$         | int   |
| y    | $l_2$         | float |

- Esistono svantaggi ad utilizzare nomi invece di indirizzi?
- i nomi devono essere unici per evitare ambiguità, quindi
- per programmi molto grandi diventa difficile pensarne di diversi ed esplicativi
- Non lo abbiamo ancora visto, ma possiamo scrivere un programma su più file, ciascuno possibilmente affidato ad un programmatore diverso [1]
  - In questo caso coordinarsi per non ripetere nomi diventa molto complesso

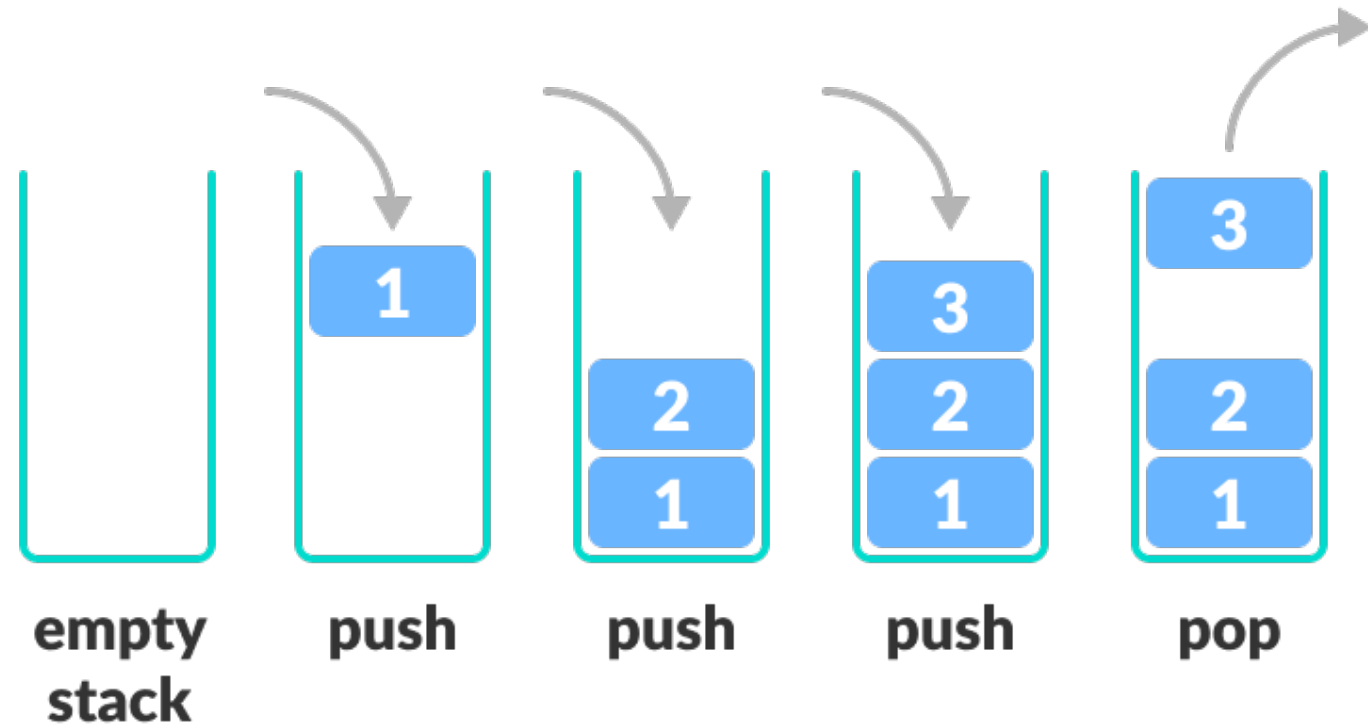
[1] <https://it.wikipedia.org/wiki/Linux>



| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $I_1$         | int   |
| y    | $I_2$         | float |

- Se vogliamo avere due variabili diverse con lo stesso nome, è sufficiente utilizzarle in punti diversi del programma
  - in questo frammento di codice x significa  $I_1$ , in un altro x significa  $I_3$
- ovvero avere un meccanismo per rimuovere (temporaneamente) e reinserire righe dalla tabella e
- garantire che non appaiano nella tabella allo stesso istante due variabili (righe) con lo stesso nome nello stesso istante

- La tabella di memoria è implementata come una pila (stack in inglese)
- gli elementi vengono
  - aggiunti in alto
  - rimossi dall'alto



- I simboli {} definiscono una sequenza (blocco) di comandi. Sono tipicamente utilizzati in combinazione con altri comandi (if e while), ma possono anche apparire da soli.
- le variabili dichiarate all'interno di un blocco sono dette locali
- vengono aggiunte allo stack quando vengono dichiarate
- vengono rimosse al termine del blocco, quando si incontra }

```
{
```

```
float y;
```

```
{
```

```
int x;
```

```
printf("%d",x);
```

```
}
```

```
}
```

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
|      |               |       |
| y    | $l_2$         | float |

- I simboli {} definiscono una sequenza (blocco) di comandi. Sono tipicamente utilizzati in combinazione con altri comandi (if e while), ma possono anche apparire da soli.
- le variabili dichiarate all'interno di un blocco sono dette locali
- vengono aggiunte allo stack quando vengono dichiarate
- vengono rimosse al termine del blocco, quando si incontra }

```
{  
float y;  
{  
    int x; ←  
    printf("%d",x);  
}  
}
```

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $l_1$         | int   |
| y    | $l_2$         | float |

- I simboli {} definiscono una sequenza (blocco) di comandi. Sono tipicamente utilizzati in combinazione con altri comandi (if e while), ma possono anche apparire da soli.
- le variabili dichiarate all'interno di un blocco sono dette locali
- vengono aggiunte allo stack quando vengono dichiarate
- vengono rimosse al termine del blocco, quando si incontra }


```
{  
    float y;  
    {  
        int x;  
        printf("%d",x);  
    }  
}
```

| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
| x    | $l_1$         | int   |
| y    | $l_2$         | float |



- le variabili dichiarate all'interno di un blocco sono dette locali
- vengono aggiunte allo stack quando vengono dichiarate
- vengono rimosse al termine del blocco, quando si incontra }
  - Esistono solamente all'interno del blocco in cui sono definite
  - in questo modo non occupano memoria anche quando non verranno più usate

```
{  
    float y;  
    {  
        int x;  
        printf("%d",x);  
    }  
}
```



| nome | Indirizzo RAM | tipo  |
|------|---------------|-------|
|      |               |       |
| y    | $l_2$         | float |

- La ricerca di un nome di variabile in una tabella avviene dall'alto verso il basso
- questo permette di avere due variabili con lo stesso nome (la risoluzione del nome è non ambigua)
- Una variabile locale è visibile (utilizzabile) ovunque all'interno del blocco in cui è definita a meno che non venga ridefinita in un blocco più interno

```
{  
    int x=2; //chiamiamo x1 questa istanza di x  
    {  
        int x=3; // da questo momento x1 non è più visibile  
    } // x1 è visibile nuovamente  
}
```

```
{ // blocco 1  
  int x; //x1  
}
```

```
{ //blocco 2  
  int x; //x2  
  int y;  
}
```

Posso definire la stessa variabile x in due blocchi diversi ed è come aver definito due variabili diverse (notate che dentro il blocco 2 non posso accedere a x1 e dentro il blocco 1 non posso accedere a x2)



- È possibile dichiarare nell'inizializzazione di un ciclo for
- la variabile `x` è locale al corpo del for

```
for (int x=1; x <= 3; x=x+1) {  
    printf("Ciao Mondo!\n");  
}  
printf("%d", x); // errore di compilazione!
```

- Una variabile è detta globale se è visibile in ogni parte del programma
- Le variabili globali sono dichiarate fuori da ogni funzione
- Ogni variabile locale nasconde la variabile globale con lo stesso nome

```
#include <stdio.h>
int x=2;
int main() {
    printf("%d", x);
}
```

stampa 2

```
#include <stdio.h>
int x=2;
int main() {
    {
        int x=3;
        printf("%d", x);
    }
}
```

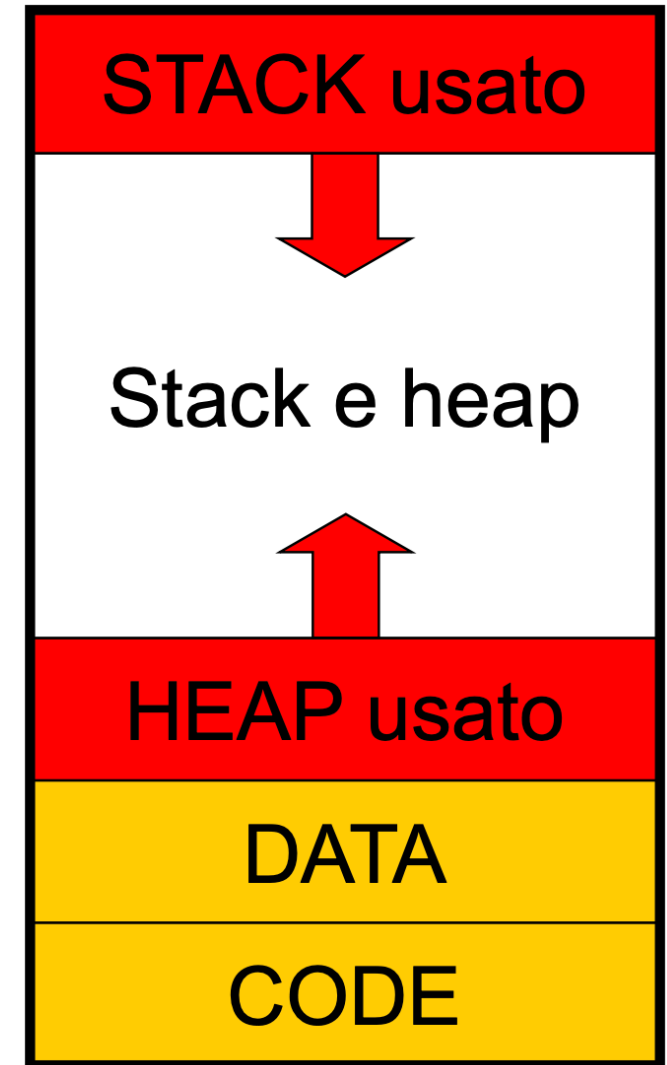
stampa 3

```
#include <stdio.h>
int x=2;
int main() {
    {
        int x=3;
    }
    printf("%d", x);
}
```

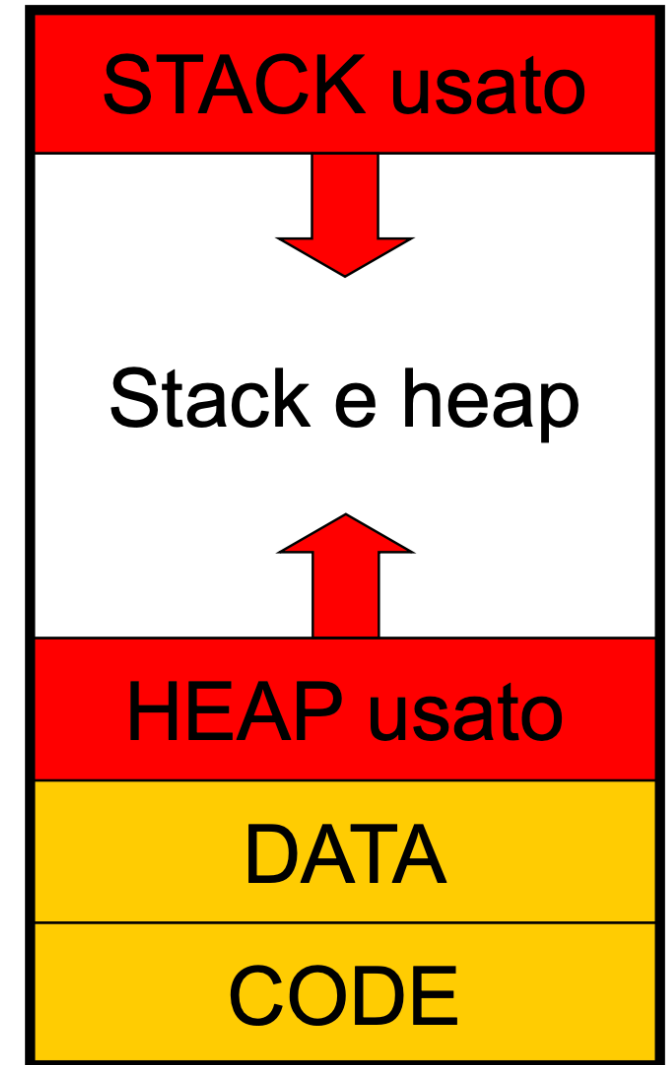
stampa 2

- Il programma compilato è costituito da due parti distinte:
- segment del codice: codice eseguibile
- segmento dei dati: costanti e variabili
- Quando il programma viene eseguito, il Sistema Operativo alloca spazio di memoria per:
- il segmento del codice (CS)
- il segmento dei dati (DS)
- lo stack e lo heap\* (condivisi)

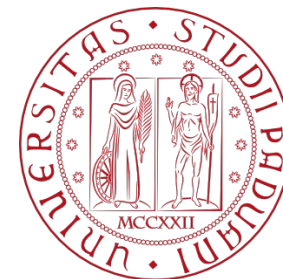
\*tratteremo lo heap in seguito



- Poiché le variabili globali
  - sono sempre visibili in ogni parte del programma
  - vivono finché il programma è in esecuzione
- Per semplificare l'implementazione vengono mantenute in una tabella diversa da quella che mantiene le variabili locali
- fanno parte del segmento DATA nella figura a fianco



# Funzioni



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Per semplificare la struttura di un programma complesso è possibile suddividerlo in moduli, sottoprogrammi o, nel gergo del C, funzioni
- Una funzione è una serie di istruzioni che assolvono a un compito preciso (ad es. calcolare se un numero è primo) e a cui è stato dato un nome
- Sintassi della definizione di una funzione:

```
tipo_restituito nomeFunzione (parametri) {  
    //definizioni variabili locali  
    comandi della funzione  
    return  
}
```

La scelta del nome della funzione segue le regole per le variabili

- Ogni funzione può essere considerata un piccolo programma isolato dalle altre funzioni
- Una funzione viene invocata scrivendo il suo nome seguito dalle parentesi ()
  - `nome_funzione()`
- Definire una funzione è come definire un nuovo comando: dopo che il corpo della funzione è stato eseguito, si torna ad eseguire il comando successivo a `nome_funzione()`

```
void stampa_ciao_mondo() {  
    int i; // la visibilità di questa funzione è locale al blocco dove è definita  
    for(i=0;i<10; i=i+1) {  
        printf("ciao mondo!\n");  
    }  
}  
  
int main () {  
    stampa_ciao_mondo();  
    printf("finito\n");  
}
```



- Una funzione può restituire un valore (per esempio di tipo int), che può essere utilizzato all'interno del codice come se fosse una variabile di quel tipo.
- Per restituire un valore si usa il comando return. Es.

```
int numero_gatti() {  
    return 44; //comando per restituire un valore alla funzione chiamante  
}
```

```
int main () {  
    int x = numero_gatti();  
    printf("Ci sono %d gatti", numero_gatti()+3);  
}
```

- Una funzione può restituire un valore (per esempio di tipo int), che può essere utilizzato all'interno del codice come se fosse una variabile di quel tipo.
- Se una funzione non restituisce niente, si usa il tipo void. Es.

```
void stampa_numero_gatti() {  
    printf("%d gatti\n", 44);  
}
```

```
int main () {  
    printf("Ci sono i");  
    numero_gatti();  
}
```

Vantaggi della programmazione modulare:

- il programma complessivo ha un maggior livello di astrazione perché i moduli “nascondono” al loro interno i dettagli implementativi delle funzionalità realizzate
- il codice per ottenere una certa funzionalità viene scritto una volta sola e viene richiamato ogni volta che è necessario
- il codice complessivo è più corto
- essendo più piccoli, i moduli sono più semplici da implementare e da verificare
- il codice di un modulo correttamente funzionante può essere riutilizzato in altri programmi

- Per rendere le funzioni più flessibili ed interessanti, si ha la possibilità di passare, all'interno delle parentesi tonde, dei parametri sui quali la funzione possa operare.
- Nella definizione della funzione, per ogni parametro bisogna indicare il tipo

```
int successivo(int n) { //parametro formale della funzione
    return n+1;
```

```
} // qua abbiamo solamente definito la funzione, non abbiamo eseguito alcun comando
```

```
int main () {
    int x=2;
    printf("x+1=%d\n", successivo(x)); //x=parametro attuale della funzione
}
```

- Gli argomenti che la funzione riceve dal chiamante devono essere memorizzati in opportune variabili locali alla funzione stesse dette parametri
- I parametri sono automaticamente inizializzati con i valori degli argomenti
- Nell'esempio precedente:

```
int successivo(int n) { return n+1; }
```

quando invoco `successivo(x)`, potete immaginare quello che succede come:

```
int successivo() { int n=x; return n+1; }
```

- in questo caso diciamo che il parametro è passato per valore

- se ci sono più parametri, i valori dei parametri attuali vengono assegnati ai parametri formali in ordine
- i parametri della funzione si comportano come variabili locali.

```
int somma(int x, int y) {  
    return x+y; //x=4, y=5  
}  
  
int main(void) {  
    int a=4,b=5, somma;  
    somma = somma(a,b);  
}
```

- Argomenti e parametri devono corrispondere in base alla posizione e al numero (almeno per le funzioni che definiremo noi)
- I nomi dei parametri sono indipendenti dai nomi delle variabili del chiamante
- Se la funzione non richiede parametri è preferibile indicare void tra le parentesi
- In memoria i parametri sono del tutto distinti e indipendenti dagli argomenti, quindi cambiare il valore di un parametro non modifica l'argomento corrispondente.

- La visibilità di una funzione indica dove essa può essere richiamata:
- si estende dal punto in cui viene definita fino a fine file (quindi può essere utilizzata solo dalle funzioni che nello stesso file seguono la sua definizione)
- Per ovviare a questa limitazione, basta aggiungere il prototipo di una funzione (la prima riga con l'aggiunta del ;)
  - `int somma(int b, int e);`
- Adesso è possibile invocare la funzione dalla riga successiva del prototipo (quindi conviene aggiungere il prototipo subito dopo gli include)
- Notate che il prototipo fornisce tutte le informazioni necessarie a chi voglia utilizzare la funzione



`int somma(int x, int y);` //corrisponde ad alla dichiarazione di una  
// variabile, `int n;` ,per una funzione.

```
int main(void) {  
    int a=4,b=5, somma;  
    somma = somma(a,b);  
}
```

```
int somma(int x, int y) {  
    return x+y; //x=4, y=5  
}
```

# Invocazione di una Funzione



```
// printf("%d", s);  
// s = somma(a,b);  
store(L-valore(a), R1)  
push*(stack, "a", R1);  
store(L-valore(b), R1)  
push(stack, "b", R1);  
push(stack, "retadd", PC);  
salta(l1) // eseguire codice della funzione
```

\*push=store() dove si mette il valore di R1 nella cella di memoria corrispondente alla cima della pila

```
// funzione somma  
load(R1, R-valore("a"))  
load(R2, R-valore("b"))  
sommaInt(R1, R2, FVAL);  
pop(stack); // rimuove b da stack  
pop(stack); //rimuove b da stack  
load(R1, R-valore("retadd"))  
pop(stack)  
Salta(R1)  
*FVAL: registro con risultato funz.
```