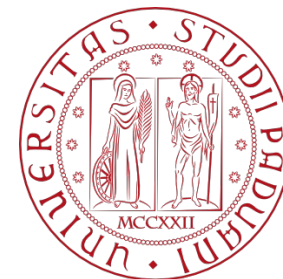


Comandi di Base



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Comandi di base del linguaggio C:

- Espressioni
- Variabili
- Comandi di scelta
- Iterazione
- Funzioni

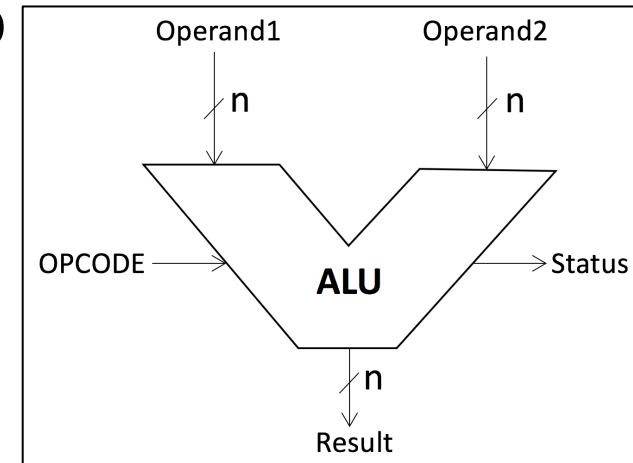
- Problema: vogliamo poter calcolare in un programma C espressioni aritmetiche, ad es. $7(2+5)-23$

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`

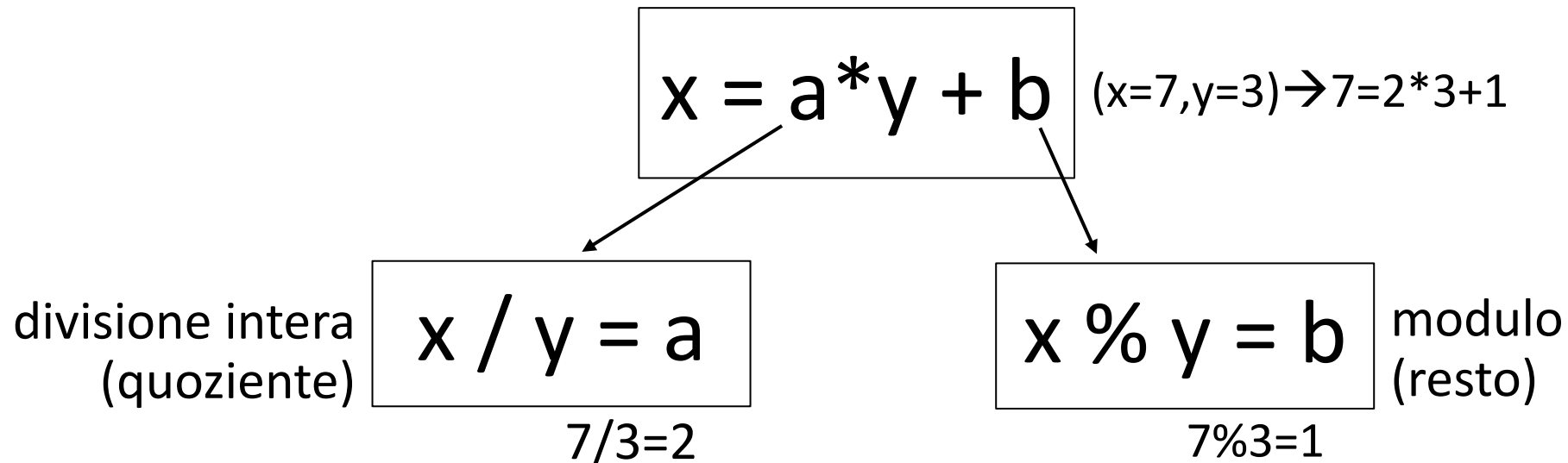
Vincoli dal linguaggio macchina

- 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 traduce automaticamente (in modo non ambiguo) espressioni complesse in sequenze di operazioni binarie: es. $(2+5)+3$
 - per farlo 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)
 - per gli operatori con la stessa priorità, l'ordine è da sinistra a destra
 - l'ordine risultante è non ambiguo



Vincoli dal linguaggio macchina

- circuiti diversi (operazioni diverse) per numeri interi e reali
- Allo stesso simbolo possono corrispondere operatori diversi, a seconda del 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 nel C non corrisponde sempre a come noi risolviamo le espressioni. 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 nel C non corrisponde sempre a come noi risolviamo le espressioni. Es.

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

- Come calcola l'espressione il C?

Ipotesi 1

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

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

$$(3/2)*2$$

- L'aritmetica nel C non corrisponde sempre a come noi risolviamo le espressioni. 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 nel C non corrisponde sempre a come noi risolviamo le espressioni. 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 nel C non corrisponde sempre a come noi risolviamo le espressioni. Es.

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

- Come calcola l'espressione il C?

Ipotesi 3

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



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

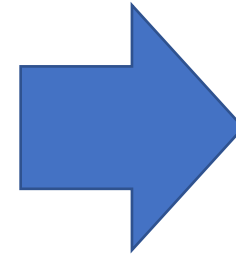
$$3.0 + 2 = ?$$

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

$$3.0 + 2.0 = 5.0$$

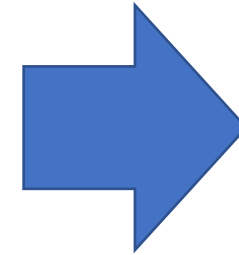
```
1  #include <stdio.h>
2
3  int main(void) {
4      |
5      printf("%d\n", 3/0);
6      |
7  }
```

file: main.c



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, maggiore di == e != (l'associatività è sempre a sinistra per tutti gli operatori)

- 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 per convenzione y vale 1)
 - `(5>3)+8==9; // 5>3==1`
 - `5<3==0`
- Esercizio: se `x==9` quanto vale

$0 \leq x < 8$?



$$0 \leq x < 8$$

- se $x=9$ vale true! perché si eseguono i confronti da sinistra a destra:
 $(0 \leq 9) < 8 \rightarrow (1) < 8 \rightarrow 1$ cioè true
- Con $(0 \leq x < 8)$ intendiamo verificare se $x \geq 0$ E se $x < 8$
- Gli operatori non vengono generalmente combinati nella stessa espressione.
- Per combinare condizioni utilizziamo degli operatori ad hoc

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

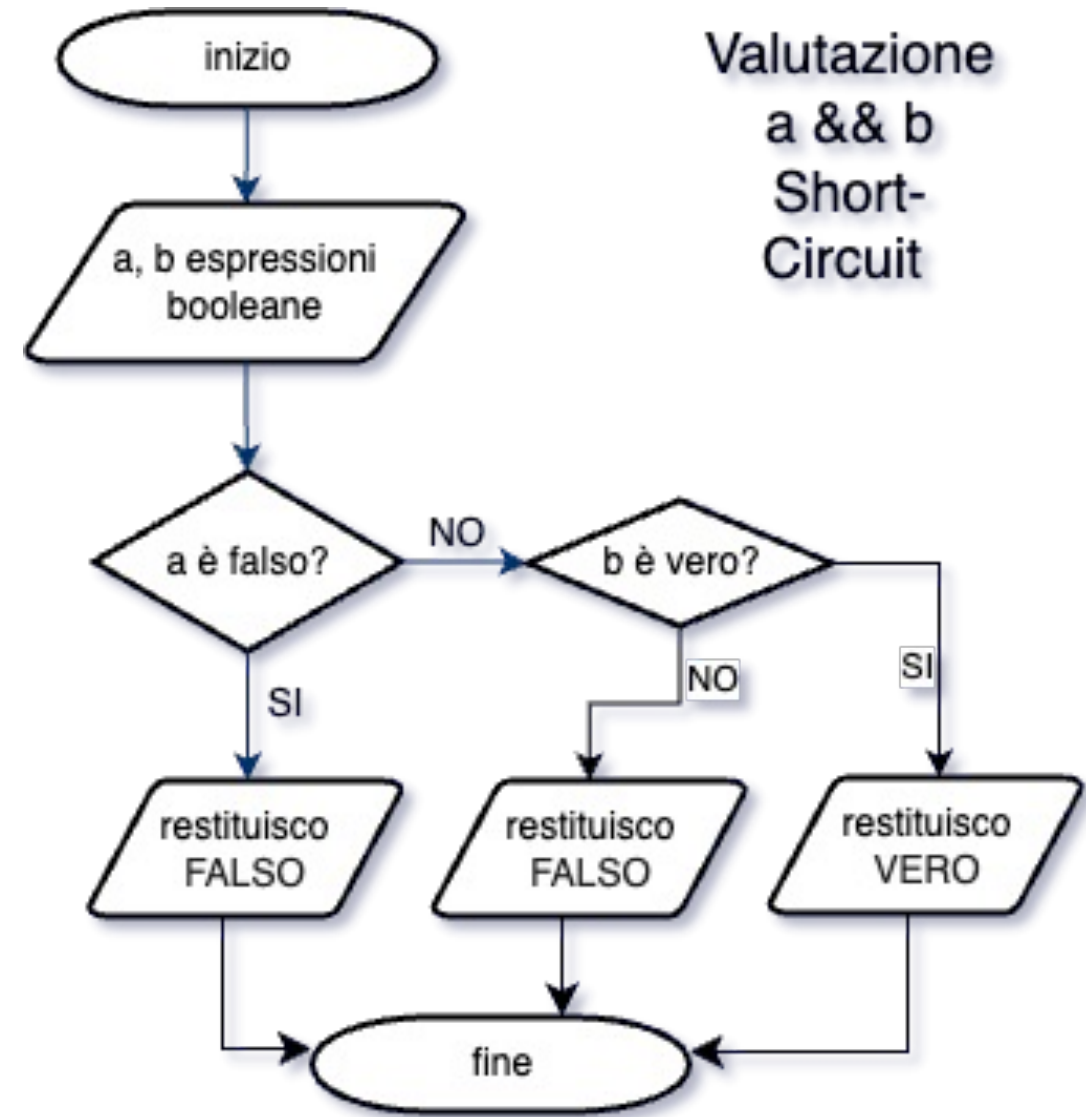
- Utilizzare più parentesi del necessario se migliorano la leggibilità

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

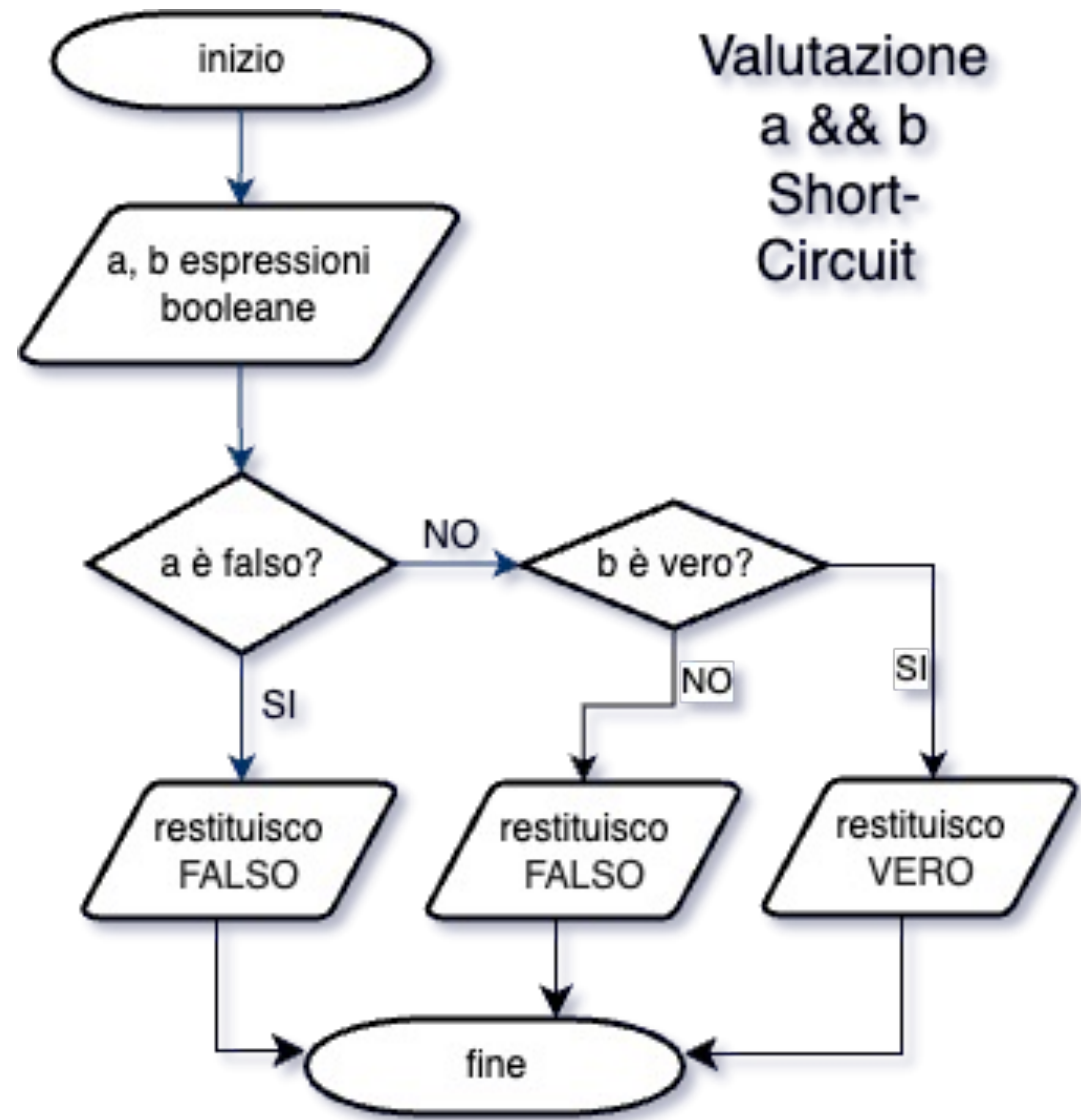


Short Circuit Evaluation - AND



- 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

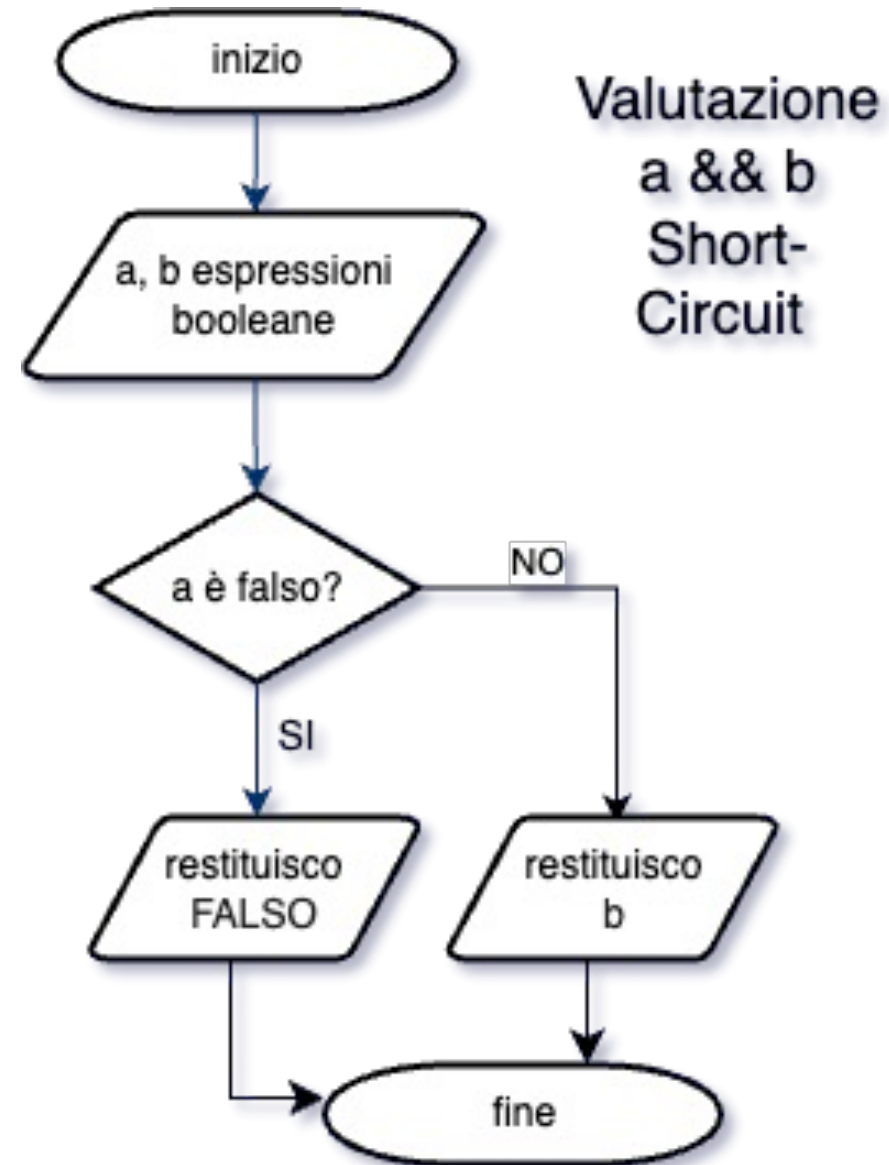


Short Circuit Evaluation - AND



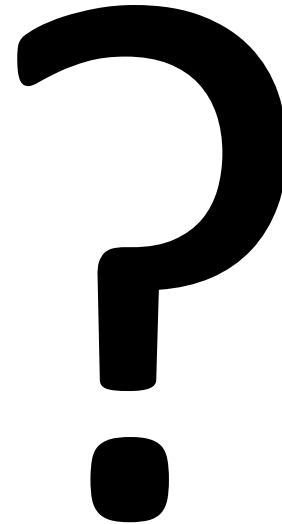
- si valuta prima a; se a è falso si restituisce falso, cioè $(a \ \&\& \ b) == 0$, **senza valutare b**
- se a è vero, restituisco il valore di verità di b

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

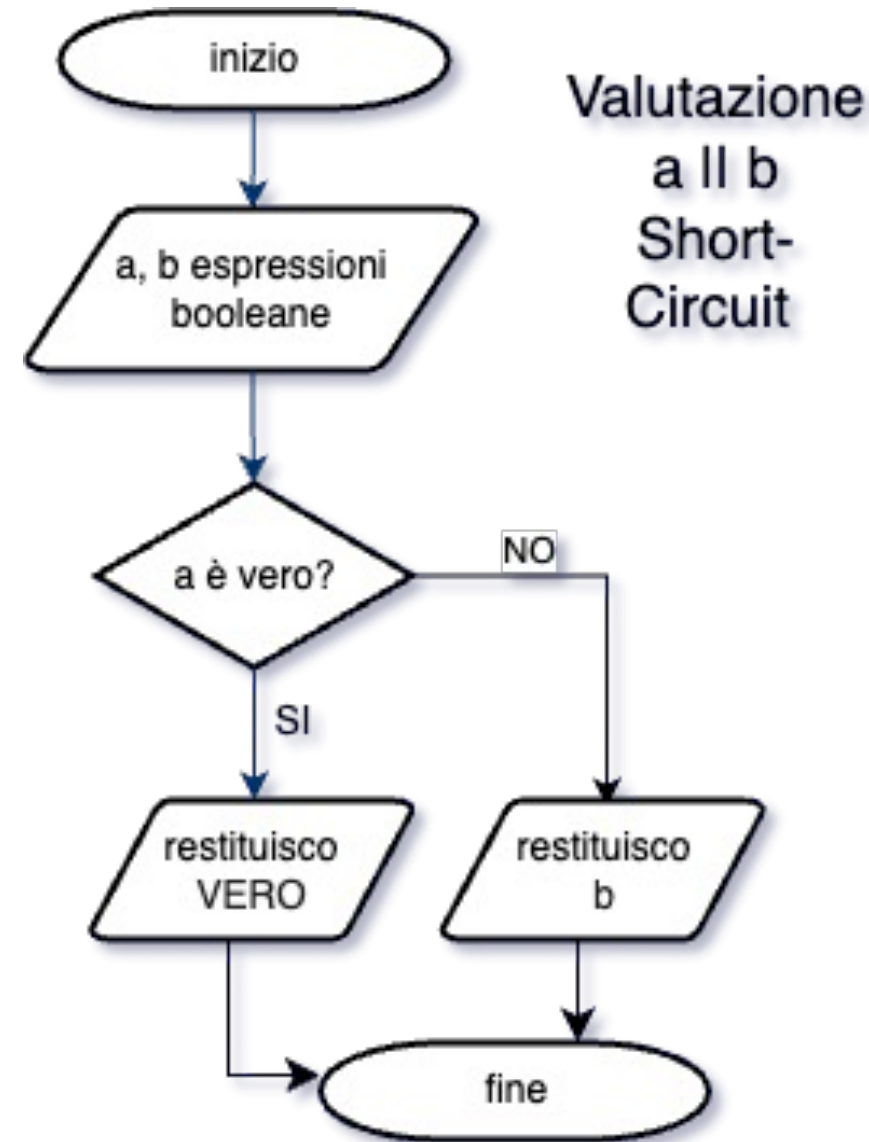


Short Circuit Evaluation - OR



- 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

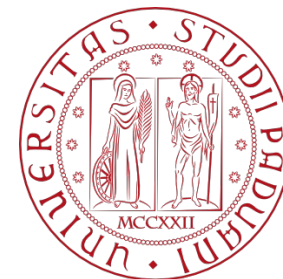


```
#include <stdio.h>

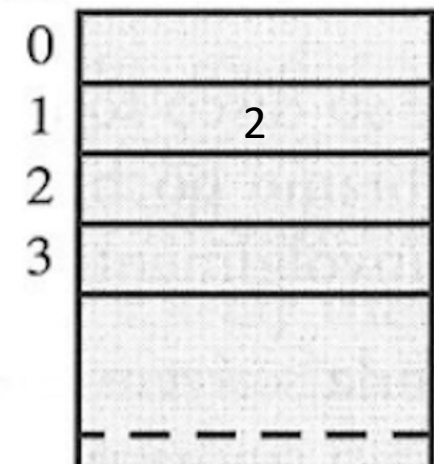
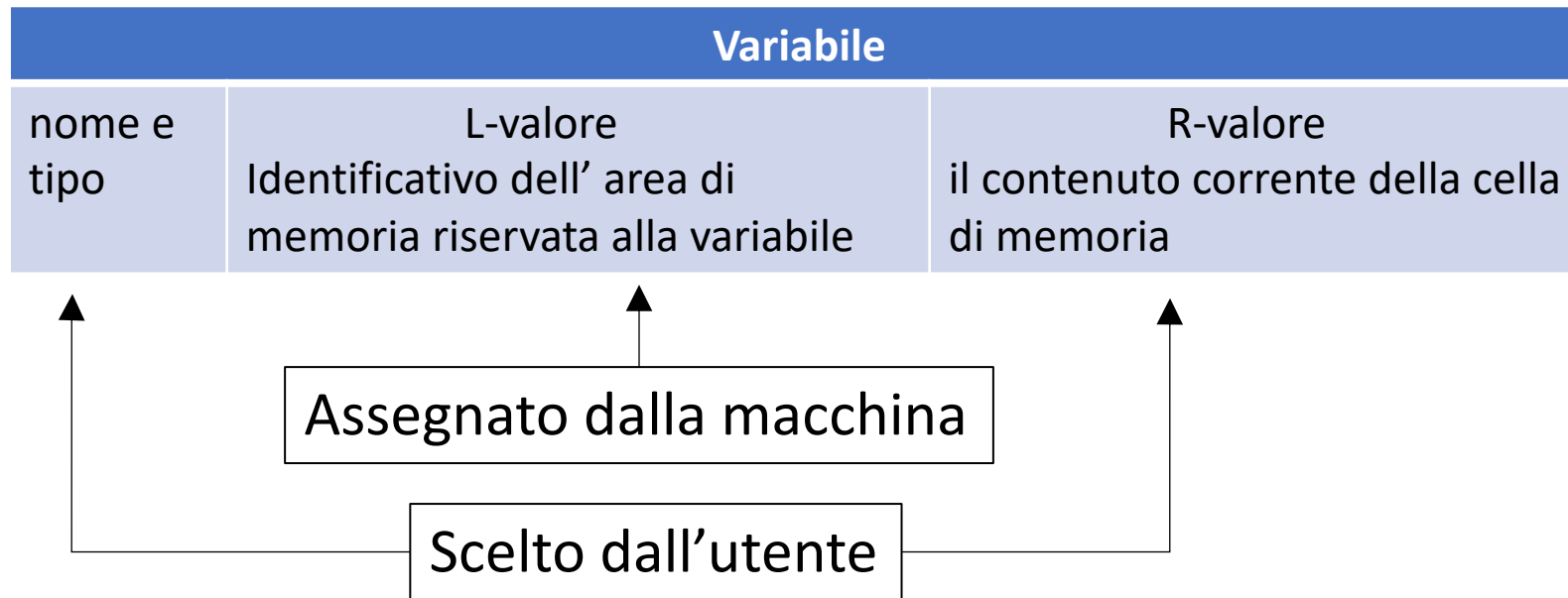
int main(void) {
    printf("%d\n", (3>0 || 3/0));
}
```

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

Variabili parte 1



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



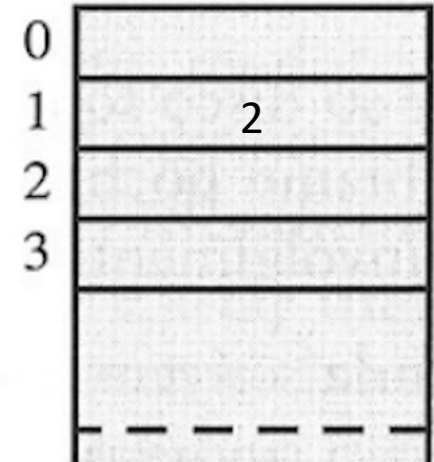
y: L-valore=1
R-valore=2

- Il tipo è un modo conciso per dire
 - quanta memoria occupa la variabile (dipende dall'architettura della macchina)
 - come leggere o scrivere la sequenza di bit
 - quali operazioni posso fare con quella variabile.

Assegnamento



- L'operazione di assegnamento = permette di modificare il contenuto (R-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



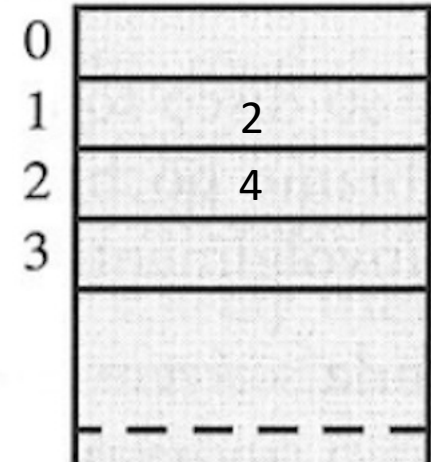
y : L-valore=1
R-valore=2

Variabili ed Assegnamento



Variabile		
nome e tipo	L-valore Identificativo dell' area di memoria riservata alla variabile	R-valore 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: $x=2+2=4$
- $x = x + 1$ // ?



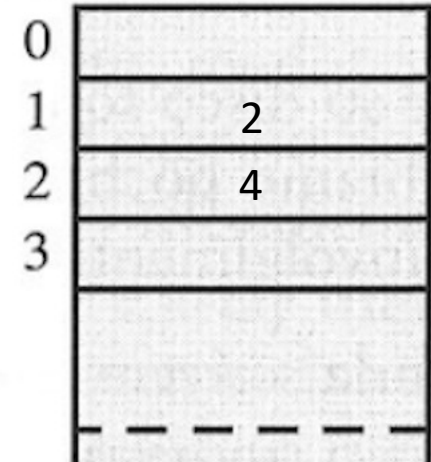
x: L-valore=2
R-valore=4

Variabili ed Assegnamento



Variabile		
nome e tipo	L-valore Identificativo dell' area di memoria riservata alla variabile	R-valore 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

- 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`
- Mai utilizzare una variabile prima di averle assegnato un valore:
`int x; int y;`
`y = x+2; // non sappiamo che valore abbia x!`
`// il valore di x è indefinito`

- 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,
 - il valore è un legame dinamico
 - dinamico altrimenti:
 - 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, quindi l'assegnamento è utilizzabile all'interno di un'espressione
- L' assegnamento ha bassa priorità come operatore
- $4+(x=8)$ 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 (rendono più lento leggere il codice)

- 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)

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"

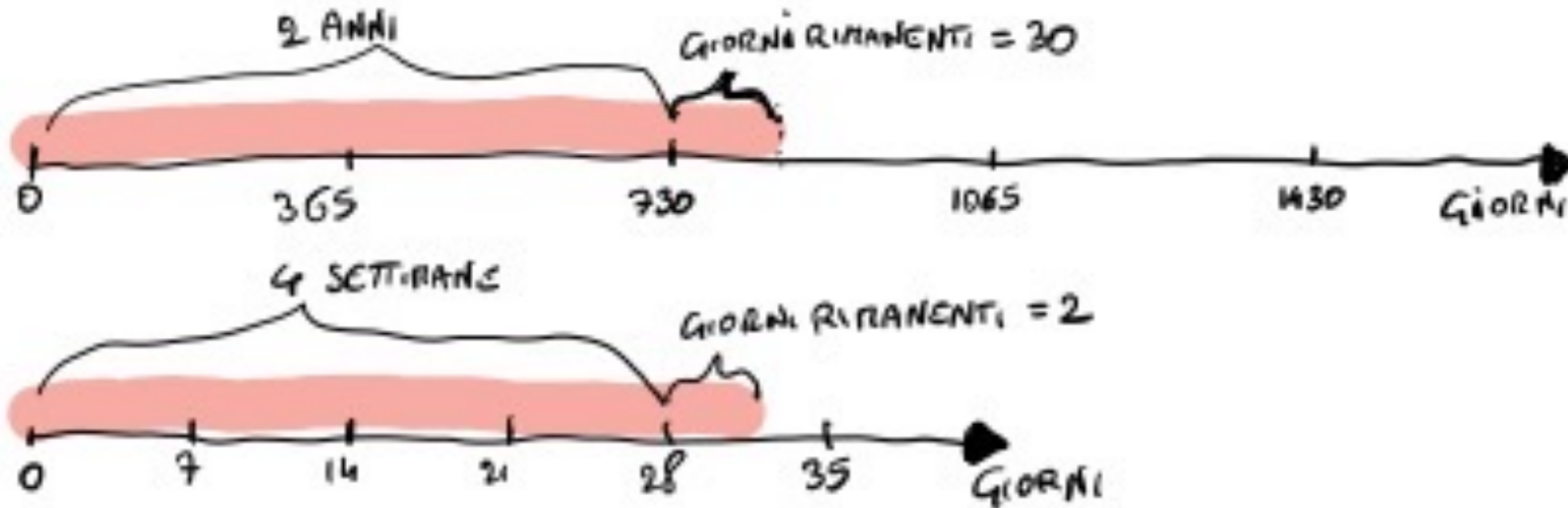
Ad esempio se fahrenheit=78 stampa

78 gradi fahrenheit corrispondono a 25.5556 gradi celsius

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

- 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

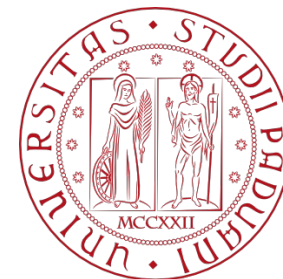
1. Contare quanti gruppi da 365 posso creare con i giorni in input
 - quanti anni stanno nei giorni in input
2. Calcolare i giorni che avanzano
3. Contare quanti gruppi da 7 posso creare con i giorni che avanzano
4. Calcolare i giorni che avanzano da quest'ultimo raggruppamento
5. Stampare il risultato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


Comandi di Selezione



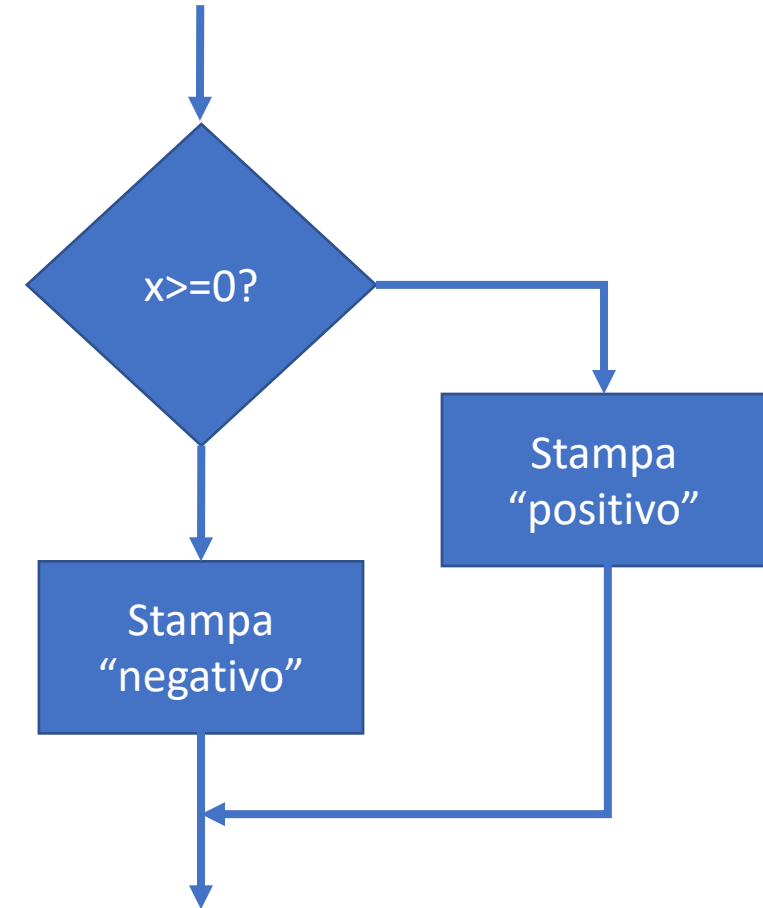
UNIVERSITÀ
DEGLI STUDI
DI PADOVA

```
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:

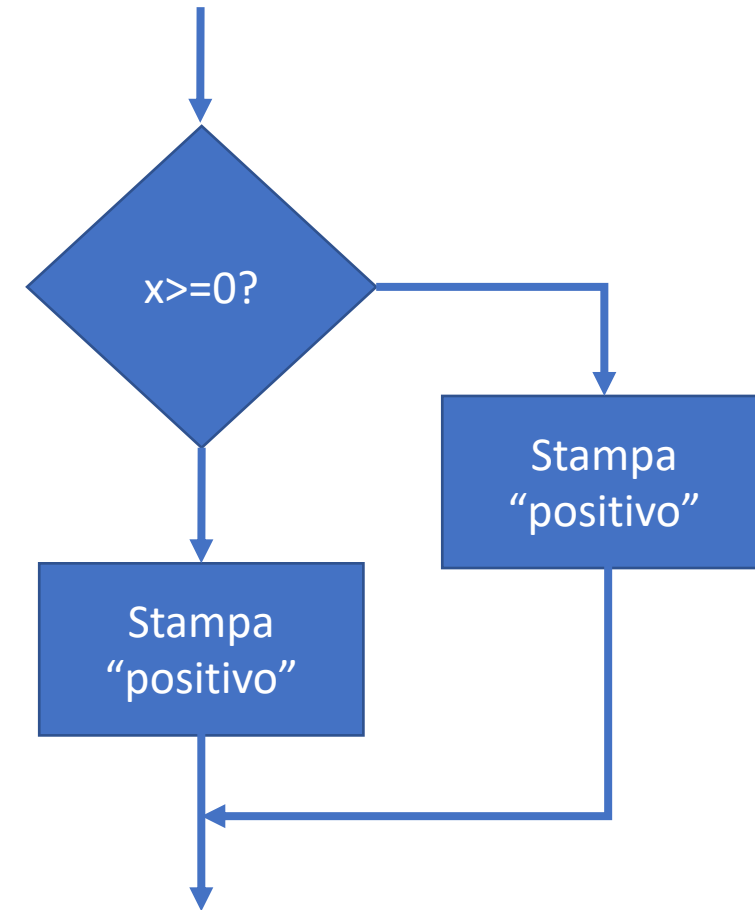
```
if (x >= 0) {  
    printf("positivo");  
} else {  
    printf("negativo");  
}
```

}
mutuamente
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 sono seguiti da ;

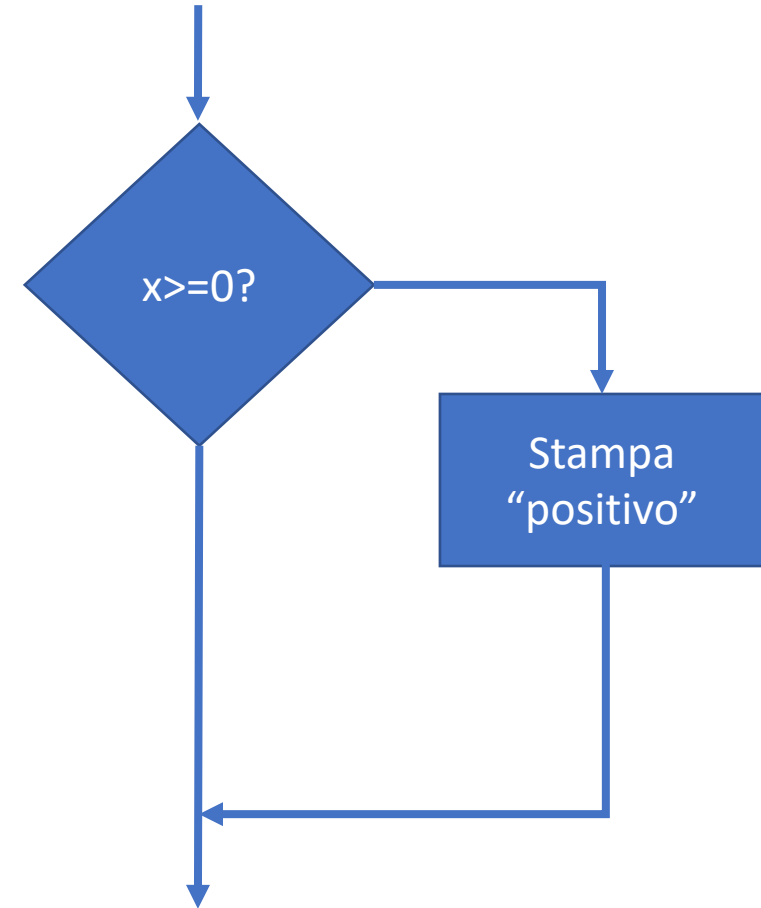


```
if (condizione) {  
    //comandi da eseguire se la condizione è vera  
}  
//comando2
```

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

- 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 con virgole  
y = 3+(x>0?x:-x); // y=5  
                // se x>0 calcola 3+x, altrimenti 3-x ma y==5 in ogni  
                // caso
```

```
#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 (“stampa 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");
    }

}
```

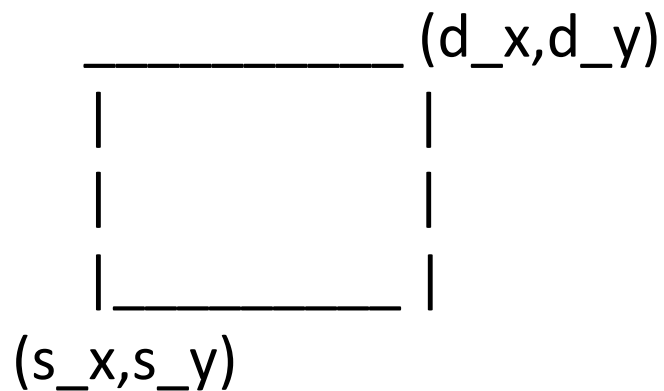
```
/*  
Date 3 variabili intere: x,y,z, stampare il valore minore tra le 3.  
Es. se x=5,y=2,z=7 stampa  
"Il minore dei tre valori è 2  
"  
*/  
  
#include <stdio.h>  
  
int main() {  
  
    int x=5, y=2, z=7;  
    printf("Il minore dei tre valori è ");  
    if (...  
  
}
```

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);
        }
    }
}
```

Dato un piano cartesiano nel quale è disegnato un rettangolo, identificato dalle coordinate del punto più in basso a sinistra $s=(s_x,s_y)$ e dal punto in alto a destra $d=(d_x,d_y)$, calcolare se un punto di coordinate (p_x,p_y) è all'interno del rettangolo (i punti sul bordo non fanno parte dell'interno del rettangolo).



Esempi: se $s=(1,1)$, $d=(4,2)$ e $p=(3,1.5)$, stampa (3, 1.5) interno al rettangolo

Se $s=(1,1)$, $d=(4,2)$ e $p=(3,2)$, stampa (3, 2) esterno al rettangolo

```
#include <stdio.h>

int main() {

float s_x=1, s_y=1, d_x=4, d_y=2; //rettangolo
float p_x=3, p_y=1.5;

printf("(%.1f, %.1f) %s al rettangolo\n", p_x, p_y,
      (p_x>s_x && p_x<d_x && p_y>s_y && p_y<d_y)? "interno": "esterno");

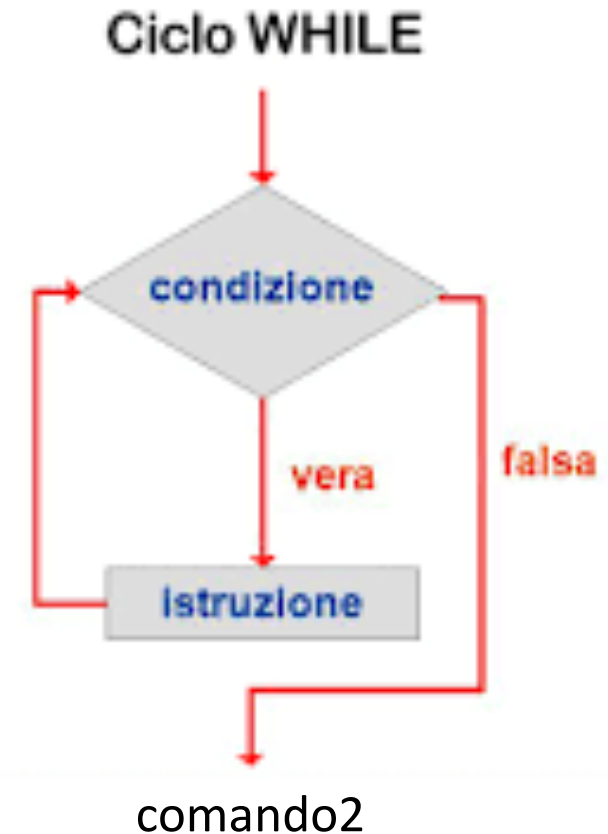
}
```



```
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



- 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 $i=i+1$;
- 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!

*/

```
#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 solamente uno dei due.

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

```
int main () {
```

```
    int a=1;
```

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

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

```
        a = a+1;
```

```
    }
```

```
}
```

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

- nel corpo di un ciclo è possibile avere qualsiasi comando, tra cui un altro ciclo.
Es. stampare le tabelline

- Stampa un quadrato di asterischi $n \times n$

- Il codice a fianco stampa ($n=4$):

- Se vogliamo stampare un quadrato 4×4 ?

```
int n=4, i;  
  
for(i=1; i<=n; i=i+1) {  
    printf("*");  
}  
printf("\n");
```

- Se vogliamo stampare un quadrato $n \times n$?

```
int n=4,i,j;

for(i=1; i<=n; i=i+1) {
    for(j=1; j<=n; j=j+1) {
        printf("*");
    }
    printf("\n");
}
```

Esercizio: Prodotto 1..n



Dato $n > 0$, Calcolare il prodotto dei numeri da 1 a n . Ad es. se $n=5$ restituisce $1*2*3*4*5=120$

```
#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

```
*/
```

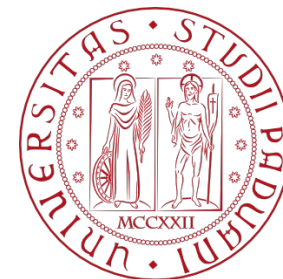
```
#include <stdio.h>
```

```
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

- Le variabili dello pseudocodice possono essere implementate nel linguaggio macchina sostituendo i nomi con indirizzi di memoria
- Però non vogliamo riferire celle di memoria RAM tramite il loro indirizzo numerico nel nostro codice,
- perché renderebbe il nostro codice molto più difficile da comprendere
- 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 i nomi può diventare ingestibile.

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

- il problema è che una volta dato un nome ad una variabile, l'associazione rimane per tutta la durata dell'esecuzione, anche quando non uso più quella variabile.
- Idea: creiamo associazioni tra un nome ed un indirizzo di memoria che abbiano una durata limitata



- I simboli {} definiscono una sequenza (blocco) di comandi. Sono di solito 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 (a quel blocco)
- vengono aggiunte alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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 di solito 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 alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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 di solito 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 alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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 di solito 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 alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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 alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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

- le variabili dichiarate all'interno di un blocco sono dette locali
- vengono aggiunte alla tabella quando vengono dichiarate (se non c'è una variabile con lo stesso nome nello stesso blocco, altrimenti si genera un errore)
- 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);
```

```
}
```



```
int x; → OK!
```

```
}
```

nome	Indirizzo RAM	tipo
y	l_2	float

```
{ // 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)

- La ricerca di un nome di variabile in una tabella avviene dall'alto verso il basso
- questa regola permette di avere due variabili con lo stesso nome in blocchi diversi (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 (in una riga superiore della tabella)

{

int x=2; // nella cella l_1

{

int x=3; // da questo momento x- l_1 non è più visibile

} // x- l_1 è visibile nuovamente

}

nome	Indirizzo RAM	tipo
x	l_6	int
x	l_1	int

- È possibile dichiarare una variabile nell'inizializzazione di un ciclo for
- tale variabile (x nell'esempio) è locale al corpo del for

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


- Le variabili globali sono dichiarate fuori da ogni funzione
- Sono visibili in ogni funzione definita dopo la loro dichiarazione (se subito dopo la `#include` ovunque nel programma)
- Ogni variabile locale nasconde l'eventuale 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);
    }
}
```

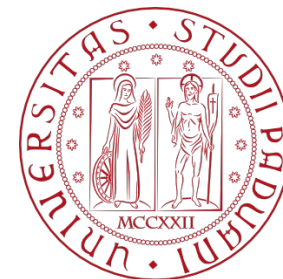
stampa 3

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

stampa 2



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 un (solo) compito (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;  
}
```

```
int domanda_ultima_universo () {  
    return 42;  
}
```

- 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 (si può usare) 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()`

- 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 che
 - interrompe l'esecuzione della funzione e torna ad eseguire la funzione chiamante



```
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 (ad esempio di tipo int), che può essere usato all'interno del codice (in un'espressione) come un qualsiasi valore di quel tipo.
- Se una funzione non restituisce niente, si usa il tipo **void**. Es.
- in C una funzione non può restituire più di un valore.

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

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

- Per il C tutte le funzioni sono definite allo stesso livello; non si possono definire funzioni all'interno di altre funzioni.

```
int precedente(int n) {  
    ...  
}  
int successivo(int n) {  
    ...  
}  
int main() {  
    ...  
}
```

OK!

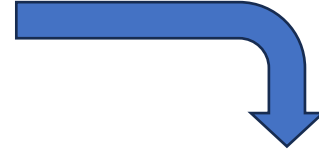
```
int precedente(int n) {  
    ...  
}  
int main() {  
    int successivo(int n) {  
        ...  
    }  
}
```

NO

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

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

funzioni-prototipo.c



gcc funzioni-prototipo.c

funzioni-prototipo.c:3:8: **error**: implicit declaration of function 'somma' is invalid in C99 [-Werror,-Wimplicit-function-declaration]

sum = somma(a,b);
 ^

1 error generated.

- Al momento dell'uso di `somma()`, la funzione non è stata ancora definita, quindi non possiamo invocarla!

- La visibilità di una funzione indica dove essa può essere invocata (usata):
- 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); // “dichiariamo” la funzione come dichiariamo variabili
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
}
```



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 breve
- 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) {           // 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
}
```

Passaggio di Parametri per Valore



- Gli argomenti che la funzione riceve dal chiamante sono memorizzati in opportune variabili locali alla funzione stessa dette parametri
- I parametri della funzione sono automaticamente inizializzati con una copia dei valori dei parametri attuali → passaggio per valore

```
int successivo(int n) {  
    return n+1;  
}  
  
int main() {  
    int x=2, n=8;  
    printf("%d\n", successivo(3));  
    printf("%d\n", successivo(x));  
}
```

```
int successivo() {  
    int n = 3;  
    return n+1;  
}
```

istruzione
“aggiunta dal
compilatore”

- se ci sono più parametri, i valori dei parametri attuali vengono assegnati ai parametri formali in ordine (il numero ed il tipo devono corrispondere)
- i parametri della funzione si comportano come variabili locali.

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

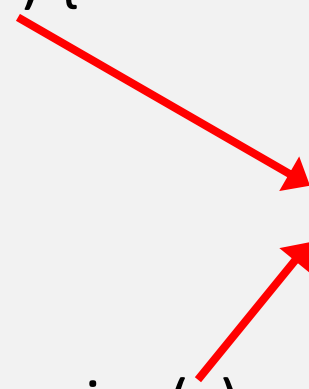
cosa stampa?

- se ci sono più parametri, i valori dei parametri attuali vengono assegnati ai parametri formali in ordine (il numero ed il tipo devono corrispondere)
- i parametri della funzione si comportano come variabili locali.

```
int somma(int x, int y) {  
    return x+y;  
}  
  
int main(void) {  
    int a=4, b=5, somma;  
    somma=somma(a,b);  
    printf("%d", x); // ERRORE di compilazione: x non è definita  
}
```

- Argomenti e parametri devono corrispondere in base alla posizione, al numero (almeno per le funzioni che definiremo noi), e al tipo.
- Se la funzione non richiede parametri si può usare void nella definizione tra le parentesi (ma non si deve)
- I nomi dei parametri sono indipendenti dai nomi delle variabili del chiamante

```
int successivo(int x) {  
    return x+1;  
}  
int main() {  
    int x=3;  
    int y = successivo(x);  
}
```



OK!

- In memoria i parametri attuali sono del tutto distinti e indipendenti dai parametri formali, quindi cambiare il valore di un parametro formale non modifica l'argomento corrispondente.

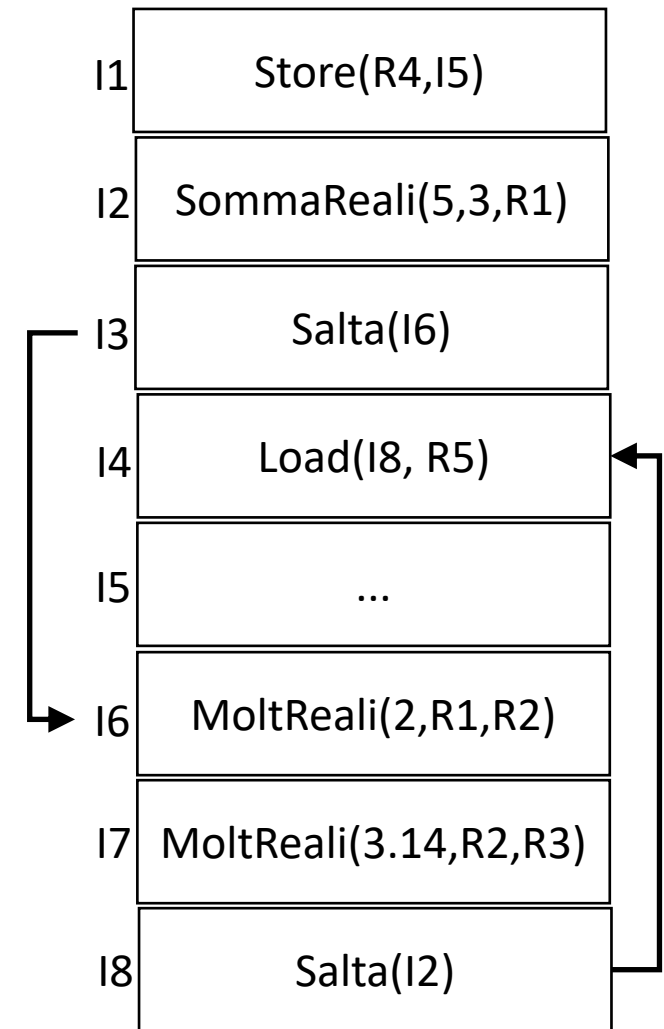
```
int successivo(int x) {  
    x=x+1;  
    return x;  
}  
  
int main(void) {  
    int x=2;  
    int y=successivo(x);  
    printf("%d", x);  
}
```

cosa stampa?

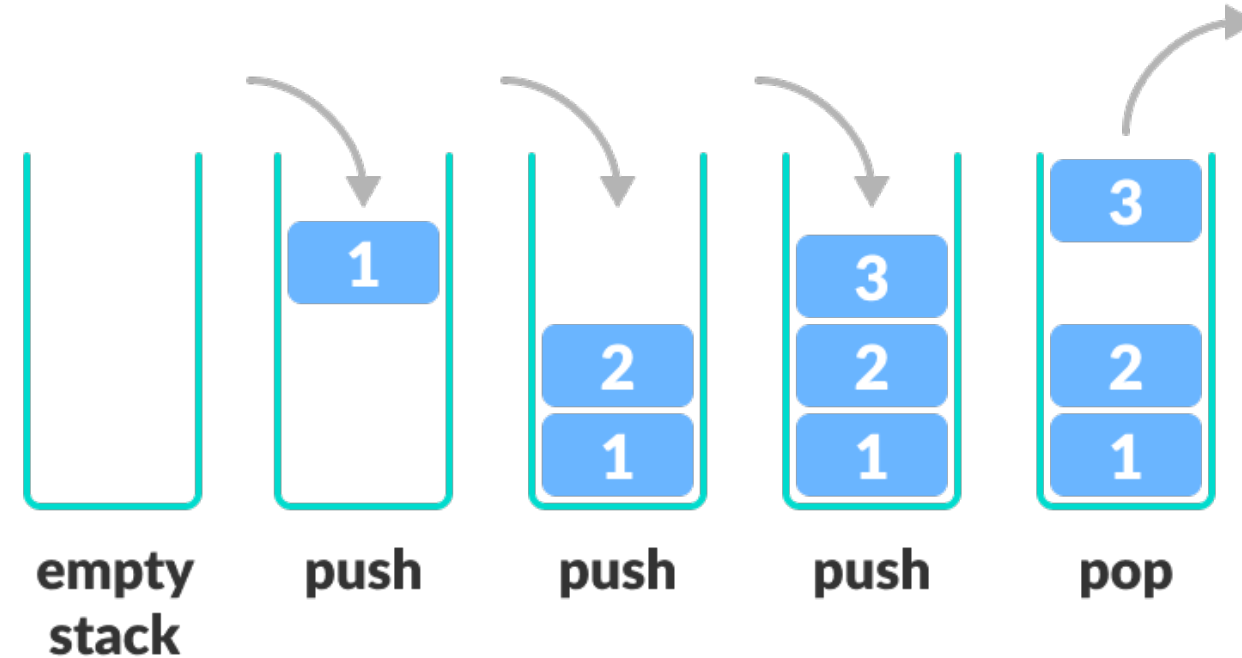
- In memoria i parametri attuali sono del tutto distinti e indipendenti dai parametri formali, quindi cambiare il valore di un parametro formale non modifica l'argomento corrispondente.

```
int successivo(int x) {  
    x=x+1;  
    return x;  
}  
  
int main(void) {  
    int x=2;  
    int y=successivo(x);  
    printf("%d", x); //stampa 2  
}
```

- Quando eseguiamo una funzione, saltiamo alla prima istruzione del codice della funzione
- dobbiamo però tenere traccia dell'istruzione che stavamo eseguendo prima del salto, perché al termine della funzione vogliamo tornare indietro e continuare ad eseguire il codice nella funzione chiamante
- Possiamo usare un registro di memoria per memorizzare l'istruzione di ritorno, es. R10?
- No, perché all'interno della funzione chiamata potremmo invocare una seconda funzione, andando a sovrascrivere R10
- Dobbiamo costruire una pila nella RAM



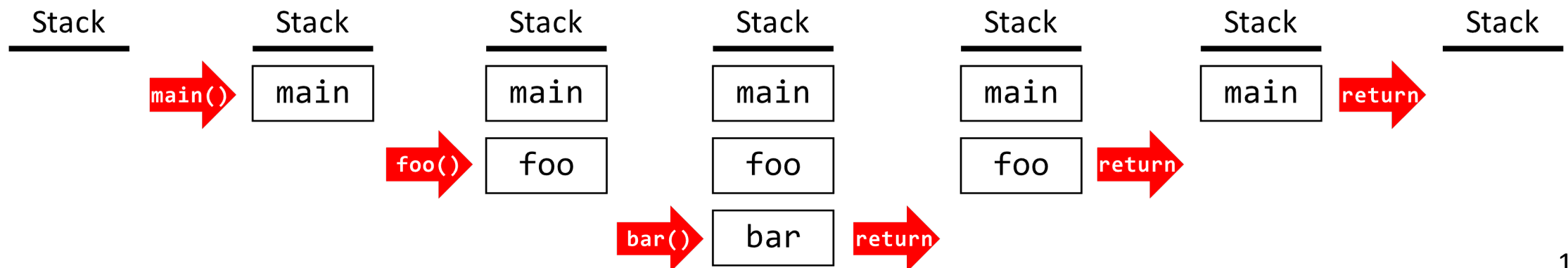
- Pila: struttura dati in cui gli elementi vengono
 - aggiunti in alto
 - rimossi dall'alto
- Un elemento della nostra pila si chiama record di attivazione e contiene:
 - l'indirizzo di ritorno della funzione (l'istruzione da eseguire quando è terminata l'esecuzione della funzione)
 - l'eventuale valore restituito dalla funzione
 - parametri della funzione
 - variabili locali della funzione



```
void bar() {}
```

```
void foo() {  
    bar();  
}
```

```
int main() {  
    foo();  
}
```



- Il codice di una funzione è in code
- Data contiene le costanti e le variabili globali del nostro programma
- i parametri e le variabili locali di una funzione vengono allocati in un record di attivazione nello stack (pila)
- Quando la funzione termina, il record di attivazione viene rimosso dallo stack; quindi in cima allo stack adesso avremo il record di attivazione della funzione chiamante

