



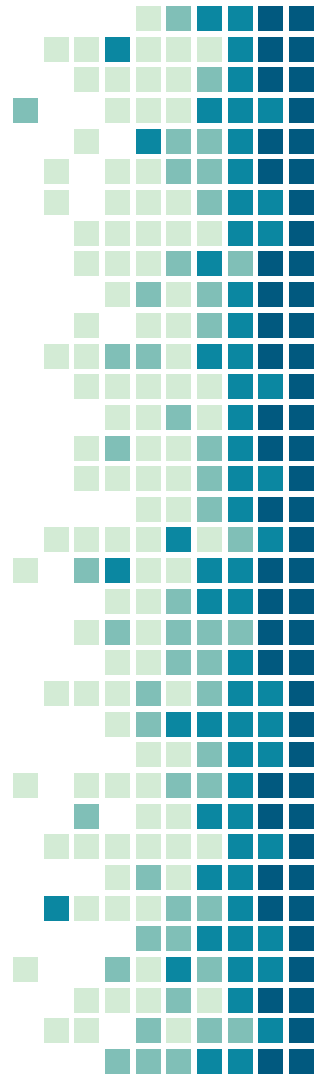
# CdL in Informatica – A.A. 2024 – 2025

## Programmazione 1 – Modulo 2

Lezione 5  
15/10/2024

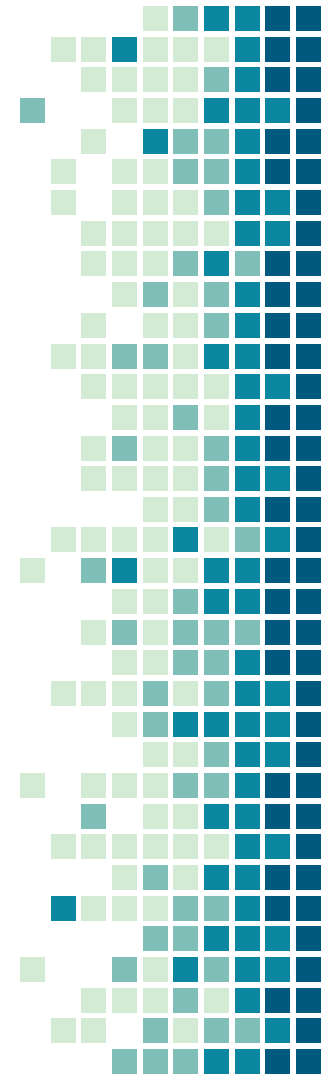
Andrea Loddo

Federico Meloni - Alessandra Perniciano - Fabio Pili



# Argomenti

- Operatori incremento e decremento
- Costrutti iterativi
- While
- Do while



# Operatori incremento e decremento



# Operatori incremento e decremento

Gli operatori di incremento e decremento unitario sono:

**++** : incrementa la variabile che lo precede (o segue) incrementando di 1;

**--** : decrementa la variabile che lo precede (o segue) decrementando di 1;

```
++x; > x++; > x += 1; > x = x + 1;  
--x; > x--; > x -= 1; > x = x - 1;
```

## Non necessita di riassegnare il valore esplicitamente

Modalità d'uso:

- **Prefisso**: ha la precedenza più alta nella valutazione di un'espressione QUINDI viene eseguito per **primo** nella valutazione dell'espressione
- **Postfisso**: ha la precedenza più bassa nella valutazione di un'espressione QUINDI viene eseguito per **ultimo** nella valutazione dell'espressione

Il risultato è **identico** ma **cambia la priorità nella valutazione dell'espressione**.

# Operatori incremento e decremento

```
i++; // operatore postfisso, rende i e poi incrementa.  
++i; // operatore prefisso, incrementa i e poi la rende.  
i--; // operatore postfisso, rende i e poi decrementa.  
--i; // operatore prefisso, decrementa i e poi la rende.
```

Rendono la scrittura del codice ancora più compatta, velocizzando incrementi/decrementi

**IMPORTANTE:** il loro utilizzo in espressioni aritmetiche causa un comportamento indefinito (undefined behaviour): lo standard del linguaggio C non specifica l'ordine in cui vengono valutati gli operandi di un particolare operatore

**Suggerimento:** evitare comportamenti indefiniti

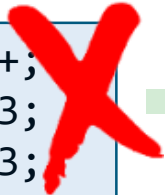
```
j = i * i;  
i++;
```

```
i = i * 3;  
i++;
```

```
i++;  
i = i * 3;
```

```
i = i + 2;  
j = i;
```

```
j = i * i++;  
i = i++ * 3;  
i = ++i * 3;  
j = ++i * i++;
```



# Un esempio

Quanto vale **b**?

Quanto vale **c**?

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b, c;

    b = ++a + 1;
    printf("Valore di b: %d.\n", b);

    c = a++ + 1;
    printf("Valore di c: %d.\n", c);

    return 0;
}
```

```
Valore di b: 7.
Valore di c: 7.
```

# Un esempio: uso corretto (ma meglio non farlo)

Eseguiamo il programma.  
Inizialmente, **a** vale 5, **b** e **c** non hanno valore.

Prima viene valutato **++a** (incrementa subito la variabile **a** che passa da 5 a 6), poi si somma 1 (6+1) e viene restituito. Viene quindi assegnato a **b** il valore 7.

Dopo l'istruzione, **a** contiene il valore 6.

Prima viene valutato **a + 1** (7), quindi incrementa la variabile **a** (che valeva 6), ma restituisce il valore di **a** non incrementato. Viene quindi assegnato a **c** il valore 7. Dopo l'istruzione, **a** contiene il valore 7.

```
#include <stdio.h>

int main()
{
    int a = 5;
    int b, c;

    b = ++a + 1;
    printf("Valore di b: %d.\n", b);

    c = a++ + 1;
    printf("Valore di c: %d.\n", c);

    return 0;
}
```

**Conclusioni:** prestate la **MASSIMA** attenzione quando usate gli operatori particolari. Il loro comportamento non è immediatamente leggibile, quindi è meglio usarli solo per i loro scopi prefissati.

# Un esempio: undefined behaviour

```
int main()
{
    int i = 0;

    i = ++i + i++;           // Quanto vale i?
    printf("Valore di i: %d.\n", i);

    return 0;
}
```

Usando **gcc** (CLion):

```
i: 3
Process finished with exit code 0
```

Usando **clang**  
(Xcode):

```
i: 2
Process finished with exit code 0
```

**Entrambi su Mac,  
ogni persona usa un  
compilatore  
diverso!**



# Operatori: priorità e associatività

Priorità	Operatore	Simbolo	Associatività
1 (max)	chiamate a funzione	()	da sinistra
	selezioni	[] -> .	
2	operatori unari: Negazione	! ~	da destra
	Aritmetici unari	+ -	
	Incremento/Decremento	++ --	
	Indirizzamento e dereferenziazione	& * *	
	sizeof	sizeof	
3	op. moltiplicativi	* / %	da sinistra
4	op. additivi	+ -	da sinistra
5	op. di shift	>> <<	da sinistra
6	op. relazionali	< <= > <=	da sinistra
7	op. uguaglianza	== !=	da sinistra
8	op. di AND bit a bit	&	da sinistra
9	op. di XOR bit a bit	^	da sinistra
10	op. di OR bit a bit		da sinistra
11	op. di AND logico	&&	da sinistra
12	op. di OR logico		da sinistra
13	op. condizionale (ternario)	?...:	da destra
14	op. assegnamento e sue varianti	=	da destra
		+= -= *= /=	
		%= &= ^=  =	
		<<= >>=	
15 (min)	op. concatenazione	,	da sinistra

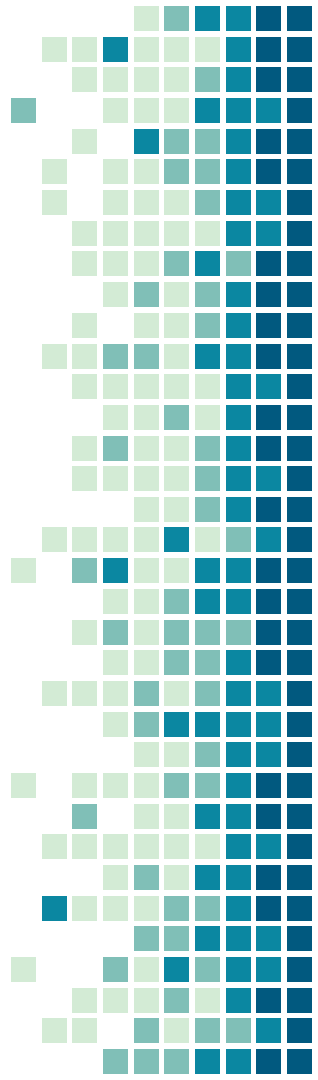
# Riepilogo

Quando vale b dopo le istruzioni:

```
int a = 5;  
int b = a++;  
printf("%d", b);
```

Quando vale b dopo le istruzioni:

```
int a = 5;  
int b = ++a;  
printf("%d", b);
```



# Domanda

Perché è meglio evitare istruzioni del tipo:

```
j = i++ * ++i;  
?
```

# Costrutti iterativi

- Esercizio: una classe di 5 studenti ha svolto un esame, valutato da 0 a 30. Determinare la media voti della classe.
- Come si potrebbe risolvere con gli strumenti visti finora?
- ...E se fossero 100?
- ...E se fossero un numero deciso dall'utente a runtime?
  - forse il "copia-incolla" non è la miglior soluzione...

```
int main()
{
    int voto = 0;
    int somma = 0;
    float media = 0.0;

    scanf("%d", &voto);
    somma += voto;

    scanf("%d", &voto);
    somma += voto;

    scanf("%d", &voto);
    somma += voto;

    scanf("%d", &voto);
    somma += voto;

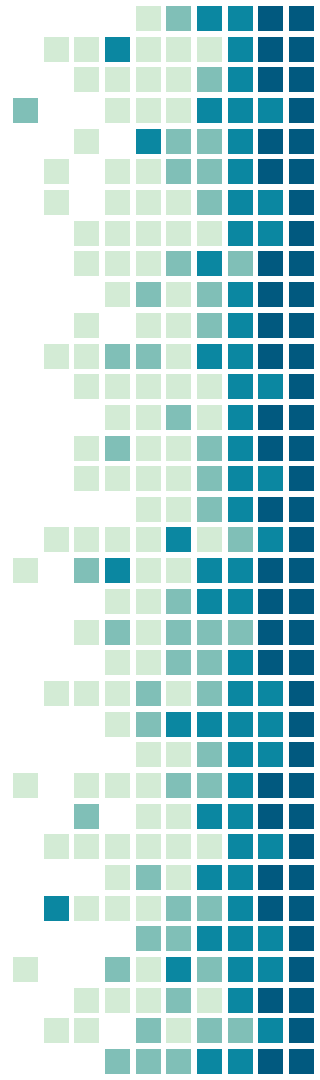
    scanf("%d", &voto);
    somma += voto;

    media = somma/5.0;

    return 0;
}
```

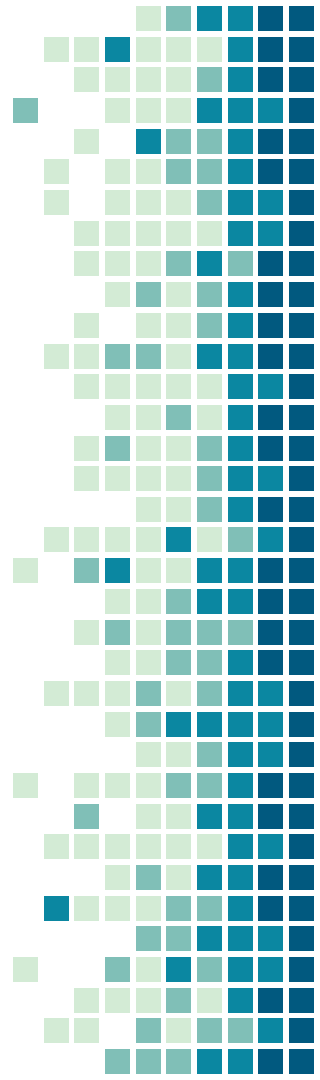
# Costrutti iterativi

- Il paradigma sequenziale prevede la possibilità di tornare indietro nel flusso d'esecuzione per ripetere una o più istruzioni (**salto**):
  - Salto **condizionato**: si ha la ripetizione di una o più istruzioni in seguito alla valutazione di un'espressione booleana.
  - Salto **incondizionato**: le istruzioni vengono ripetute senza effettuare alcuna valutazione (meglio evitare).



# Costrutti iterativi

- In generale:
  - la possibilità di eseguire più volte una o più istruzioni aumenta le potenzialità di un linguaggio di programmazione;
  - consente di risolvere una più ampia gamma di problemi.
- Il C distingue **semanticamente** tra la ripetizione per un:
  - numero di iterazioni indefinite: costrutti **while** e **do while**;
  - numero di iterazioni definite: costrutto **for**.
- **NB**: la scelta fra questi due tipi di ciclo è legata a considerazioni di leggibilità, chiarezza del codice e "teoria" del linguaggio.
- In linea di principio, infatti, qualunque algoritmo che impieghi un **for** può essere trascritto in una forma che usi il **while**, e viceversa.



# Costrutto while

- Il comando **while** viene solitamente usato per la ripetizione di un insieme di istruzioni per un numero indefinito di volte.

```
while ( condizione )
{
    /* blocco di istruzioni da svolgere
       finché la condizione è vera */
}
```

- Funzionamento del while:
  - Si valuta l'espressione booleana **condizione** all'interno delle parentesi tonde;
  - Se la condizione è **vera** si eseguono le istruzioni del **blocco**;
  - Al termine del blocco si ripete la valutazione di **condizione**;
  - Se la valutazione di **condizione** risulta **falsa**, il ciclo **termina** ed il flusso di esecuzione passa alla prima istruzione dopo il while.
- **NB:**
  - il blocco viene eseguito anche se **condizione** cambia in qualsiasi punto di esso;
  - al più non avverrà la ripetizione del blocco a partire dall'iterazione successiva.

# While, caso studio 1: iterazione controllata con contatore

```
int main()
{
    int voto = 0, somma = 0, conta = 0;
    float media = 0.0;

    while( conta < 5 )
    {
        printf("Inserire voto %d: \n", conta+1);
        scanf("%d", &voto);

        somma += voto;
        conta++;
    }

    media = somma/5.0;
    printf("La media voti dei 5 studenti e'
           %.2f", media);

    return 0;
}
```

- Se la condizione è vera, esegue il blocco; altrimenti salta alla prima istruzione dopo il blocco
- Dopo l'esecuzione, si torna alla valutazione della condizione
- Se la condizione è **falsa** in partenza, le istruzioni del blocco non vengono **mai** eseguite



## While, caso studio 2: iterazione controllata da sentinella

Come potremmo **generalizzare** l'esempio precedente per un numero arbitrario di studenti?

```
int main(){
    int voto = 0, somma = 0, conta = 0;
    float media = 0.0;

    while( voto != -1 ){
        printf("Inserire voto %d (-1 per terminare): \n", conta+1);
        scanf("%d", &voto);

        if (voto != -1){
            somma += voto;
            conta++;
        }
    }
    media = (float)somma/conta;
    printf("La media voti dei %d studenti e' %.2f", conta, media);

    return 0;
}
```

# Costrutto do while

- Il comando **do while** viene usato per la ripetizione di un insieme di istruzioni per un numero indefinito di volte, quando si necessita che il blocco d'istruzioni venga eseguito almeno una volta.

```
do
{
    /* blocco di istruzioni da svolgere almeno
    una volta e finché la condizione è vera */
} while ( condizione );
```

**Attenzione:** ; dopo  
il while SEMPRE  
necessario

- Il funzionamento del do while consiste in:
  - Si eseguono le istruzioni del blocco;
  - Si valuta l'espressione booleana **condizione** all'interno delle parentesi tonde;
  - Se la condizione è **vera** si eseguono nuovamente le istruzioni del **blocco**.
  - Al termine del blocco si ripete la valutazione di **condizione**.
  - Se la valutazione di **condizione** risulta **falsa**, il ciclo **termina** ed il flusso di esecuzione passa alla prima istruzione dopo il do while.
- NB:**
  - il blocco viene eseguito anche se **condizione** cambia in qualsiasi punto di esso;
  - al più non avverrà la ripetizione del blocco a partire dall'iterazione successiva.

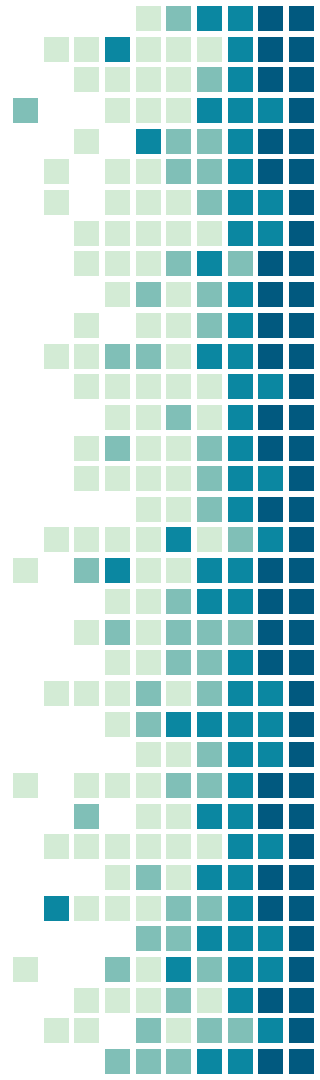
# Do while: esempio

```
int i = 1, valoreUtente;  
  
printf("Inserisci un numero: ");  
scanf("%d", &valoreUtente);  
  
do  
{  
    printf("Numero: %d\n", i);  
    i++;  
} while( i < valoreUtente );
```

- Prima di controllare la condizione si eseguono le istruzioni nel blocco
- Se poi la condizione risulta vera, si ripete il blocco, altrimenti prosegue con l'istruzione successiva
- Se la condizione è falsa in partenza, si sarà eseguito almeno una volta il blocco d'istruzioni

# Utilizzo di while e do while

- Usare ogni volta che si deve ripetere un insieme di operazioni sulla base della validità o meno di una condizione
- Il valore di verità della condizione varia sulla base delle operazioni eseguite **all'interno del ciclo**
- Nota bene: la condizione è valutata o **PRIMA** o **DOPO** le istruzioni del blocco, **mai DURANTE**
- Se la condizione diventa falsa a metà ciclo non ci sarà un'immediata uscita dal blocco, ma **si attende** comunque la valutazione seguente



# Infinite loop (ciclo infinito)

- Se in un ciclo la condizione è sempre vera si entra in un **ciclo infinito**.
- Non è un errore sintattico, ma a runtime il programma va in "**stallo**".

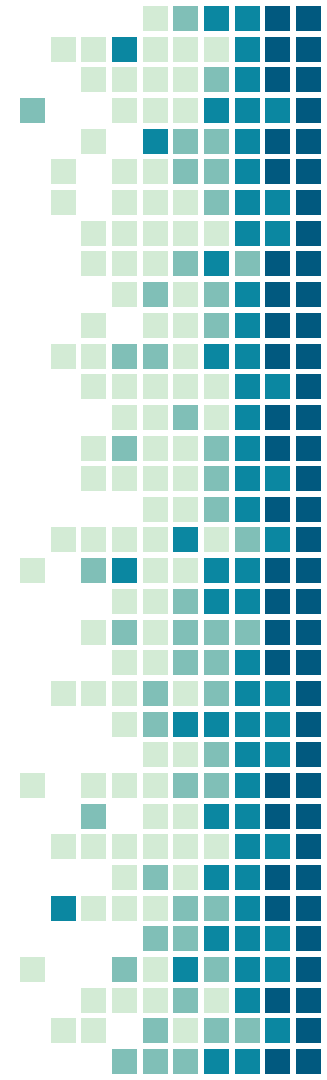
```
int x = 50;

srand(time(NULL));

while ( (x >= 0) && ( x <= 100) )
{
    x = 1 + rand() % 50;

    printf("Numero generato: %d\n", x);
}

...
```

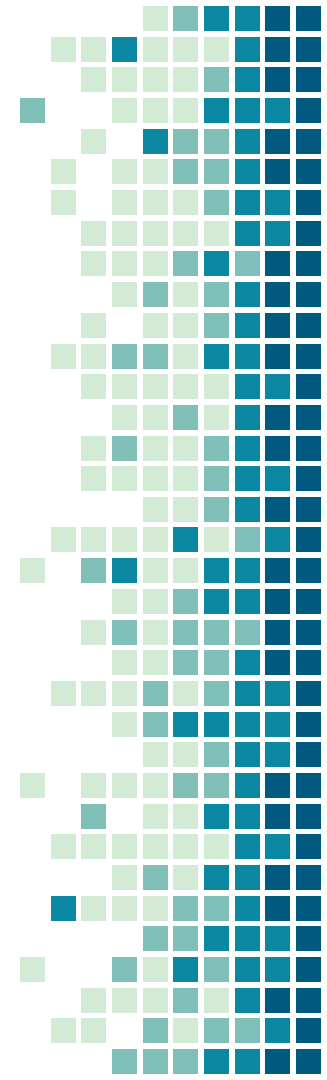


# FINE!

Domande?

# Autovalutazione: operatori

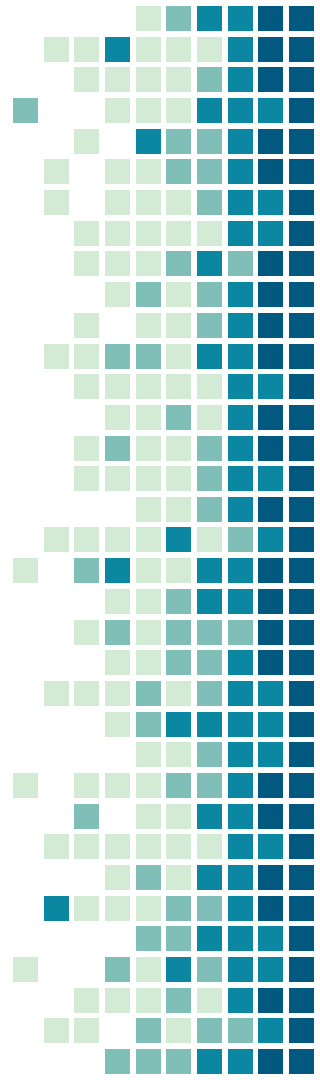
1. Che differenze ci sono tra le seguenti espressioni:  $x += 1$ ,  $x = x + 1$  e  $x++$  ?
2. Che differenze ci sono tra le seguenti espressioni:  $x += 5 * 3$ ,  $x = x + 5 * 3$  e  $x = (x + 5) * 3$  ?
3. Che differenza c'è tra incremento prefisso e postfisso?
4. Che differenza c'è tra queste due espressioni:  $y = x++ * 5$  e  $y = ++x * 5$  ?





# Autovalutazione: costrutti iterativi

1. Che differenza c'è tra salto condizionato e incondizionato?
2. Quale costrutto utilizziamo se dobbiamo eseguire un numero di iterazioni indefinite?
3. ... E se il numero è definito?
4. ... E con un numero di iterazioni indefinite maggiore di zero?
5. Che differenze ci sono tra `do while` e `while`? In quali situazioni si usa uno e in quale l'altro?
6. La condizione all'interno del `while` è facoltativa?
7. Se a metà del blocco di codice cambia il valore presente nella condizione del `while` cosa succede?
8. Il blocco di codice nel `while` è obbligatorio? Ovvero, l'assenza di almeno un'istruzione permette la compilazione?
9. Nel `do while`, le parentesi graffe sono obbligatorie? E il `;` dopo il `while`?
10. È possibile inserire un ciclo `do while` all'interno di un ciclo `while`?



# Esercizi

1. Scrivere 3 espressioni aritmetiche che utilizzino un operatore prefisso o postfisso; stampare, quindi, il valore di tutte le variabili implicate nell'operazione prima e dopo il calcolo dell'espressione. Commentare opportunamente tutto il codice.
2. Scrivere un programma che esegua la divisione tra due numeri, chiedendo di inserire nuovamente il divisore finché questo non è diverso (non è diverso == uguale) da 0.
3. Calcolare il numero di cifre che compongono un numero intero inserito dall'utente.
4. Scrivere un programma che chieda all'utente un numero `num` all'utente, e generi continuamente numeri casuali compresi tra 0 e MAX finché il numero generato non è maggiore di `num`. Stampare infine il numero di valori generati casualmente e il valore maggiore di `num`.  
Opzionalmente, verificare che l'utente inserisca un valore consona.
5. Scrivere un programma che, dati due numeri `a` e `b` generati casualmente, calcoli  $a^b$  senza utilizzare funzioni di libreria. Verificare poi la correttezza del risultato utilizzando la funzione `pow()` definita nella libreria `math.h`.

# Esercizi

6. Scrivere un programma che acquisisca numeri float in successione. Per ogni numero inserito dovrà essere calcolata la media e stampato il minore ed il maggiore inseriti fino a quel momento. Il programma deve terminare nel momento in cui si inserisce 0 o un numero qualsiasi negativo.
7. Scrivere un programma che calcoli la media di tutti i valori introdotti dalla tastiera finché non ne viene introdotto uno non compreso tra 18 e 30. La visualizzazione della media deve avvenire solo alla fine, e non ogni volta che un valore viene introdotto.
8. Scrivere un programma che richieda N numeri da tastiera e ne calcoli il valore massimo.
9. Scrivere un programma in cui viene generato casualmente un numero intero compreso tra 0 e 99 e chiedere all'utente di indovinare quel numero. Ad ogni input dell'utente il programma risponde con "troppo alto" o "troppo basso", finché non viene trovato il valore corretto.

