# Laurea In Ingegneria dell'Informazione

# Esercitazioni Guidate di Tecniche della Programmazione

**Note introduttive: Seguire le raccomandazioni date nelle precedenti EG …**

## 4. Esercitazione 4 (PRIMA e SECONDA PARTE)

### *4.1. ESECUZIONE PASSO-PASSO*

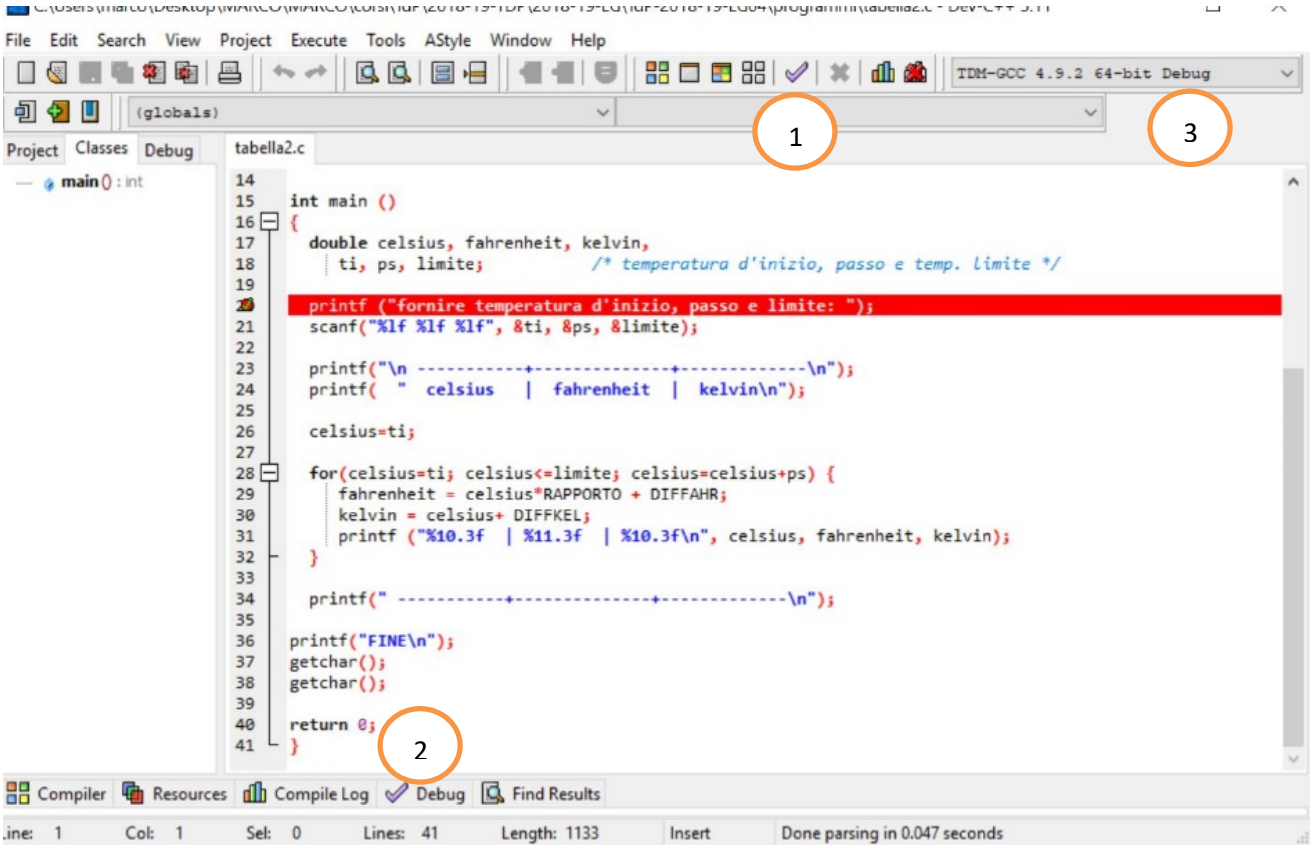
Riprendere il programma tabella2.c.
Adesso ci esercitiamo sull'uso del Debugger.
In DEV è possibile aggiungere al programma compilato informazioni utili per eseguire poi il programma passo-passo, vedendo come si comporta l'esecuzione. Questo e' utile quando si deve correggere un programma: si esegue il programma, tenndo d'occhio l'evoluzione delle variabili (del loro contenuto) e delle funzioni (delle loro chiamate) e si ha qualche probabilità in più di scoprire perché non funziona …

Per eseguire un programma in modalità debug bisogna

- Inserire un *breakpoint* (un breakpoint è un punto del programma in cui vogliamo che l'esecuzione si interrompa momentaneamente, in attesa del nostro ordine di proseguire).
- Usare il tasto Debug, che permette di compilare il programma in modo che poi sia possibile usare le funzionalità di *debugging* (esecuzione passo-passo del programma, ispezione del contenuto delle variabili …)

Nella figura successiva si vede che abbiamo inserito un breakpoint in corrispondenza di un'istruzione (è la non tanto sottile linea rossa)

Per inserire un breakpoint si può semplicemente cliccare sulla parte sinistra della linea, in corrispondenza del numero di linea. In prossimità del numero di linea si vede un simbolo che indica il brakpoint, e la linea diventa rossa. (Per togliere il breakpoint basta fare click ancora sul numero di linea). Quando il programma è in *"esecuzione in modalità debug"*, l'esecuzione si blocca ogni volta che sta per essere eseguita una istruzione corrispondente ad un breakpoint.



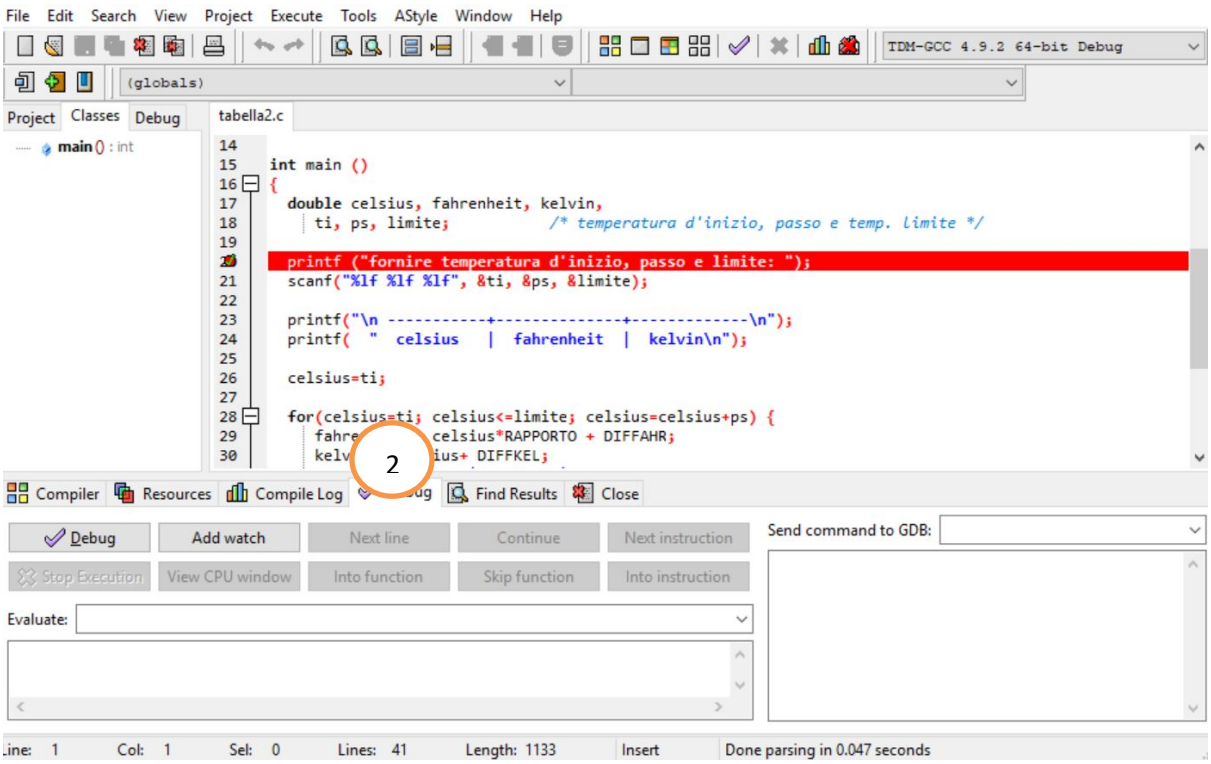

… ma ora il programma non è ancora in esecuzione.
Se lo compiliamo e mandiamo in esecuzione come facciamo di solito, lo vedremo funzionare, ma non vedremo la sua esecuzione in *modalità Debug*.

Per vedere il programma eseguito in modalità debug, bisogna averlo compilato attraverso l'opzione di debug; l'opzione si attiva in uno tra due modi:

1) Cliccando sul tasto segnalato come (1) in figura

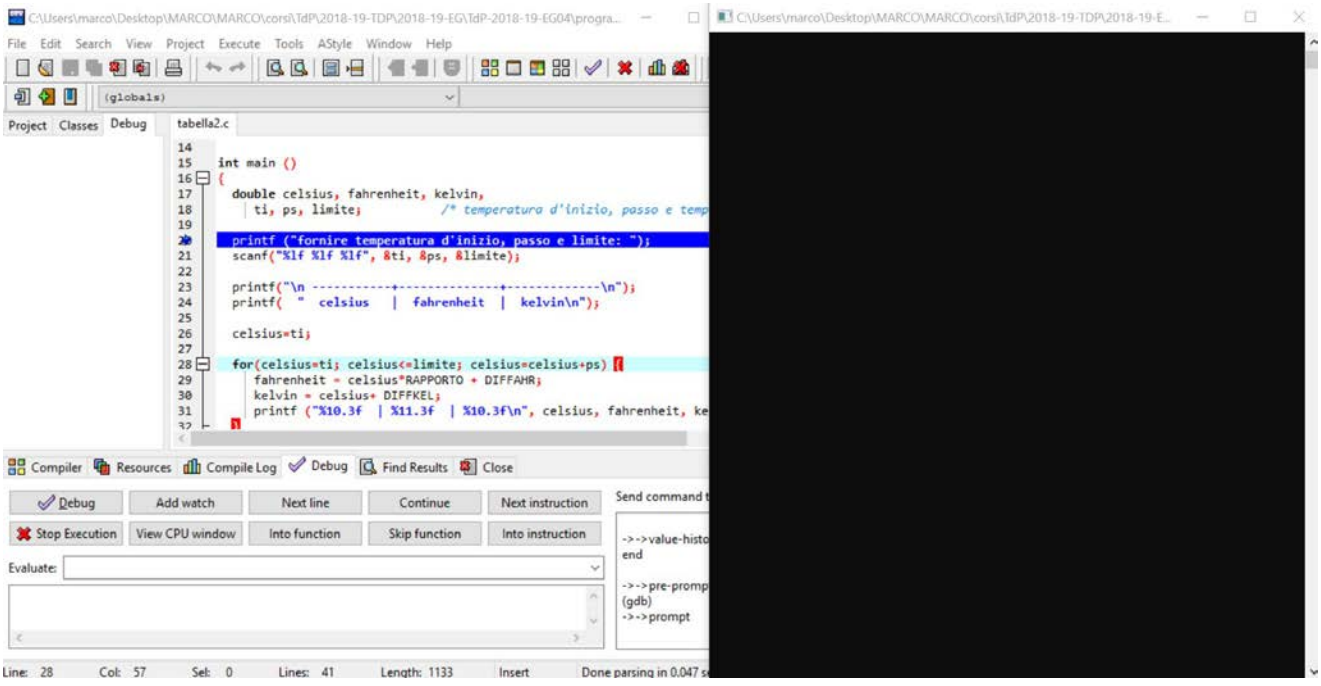
2) oppure aprendo il pannello di debug mediante clic sull'opzione corrispondente nella barra in basso (quella che si apre per farci vedere i risultati della compilazione). Se qui si clicca sul bottone segnalato con (2), si apre la porzione di finestra dedicata al *debugging* (vedi figura seguente).

**Però,** dobbiamo stare attenti ad aver selezionato il compilatore con potere di debug, in alto a destra (3). Si tratta di una casella di selezione. Di solito abbiamo in funzione il primo compilatore in lista. Ora dovremmo fare in modo di avere selezionato il secondo (quello con indicato (Debug) in fondo al nome.




Quando il programma è in esecuzione in modalità debug, l'esecuzione si vede attraverso la solita finestra, vedi figura successiva, dove abbiamo ridimensionato le due finestre, in modo da poterle vedere contemporaneamente.

> [Ridimensionare e disporre le finestre come si vede sotto è molto utile durante le operazioni di test/debugging dei programmi … in particolare aiuta ad evitare di confondersi su quale sia la finestra attiva mentre spingiamo qualche tasto …]



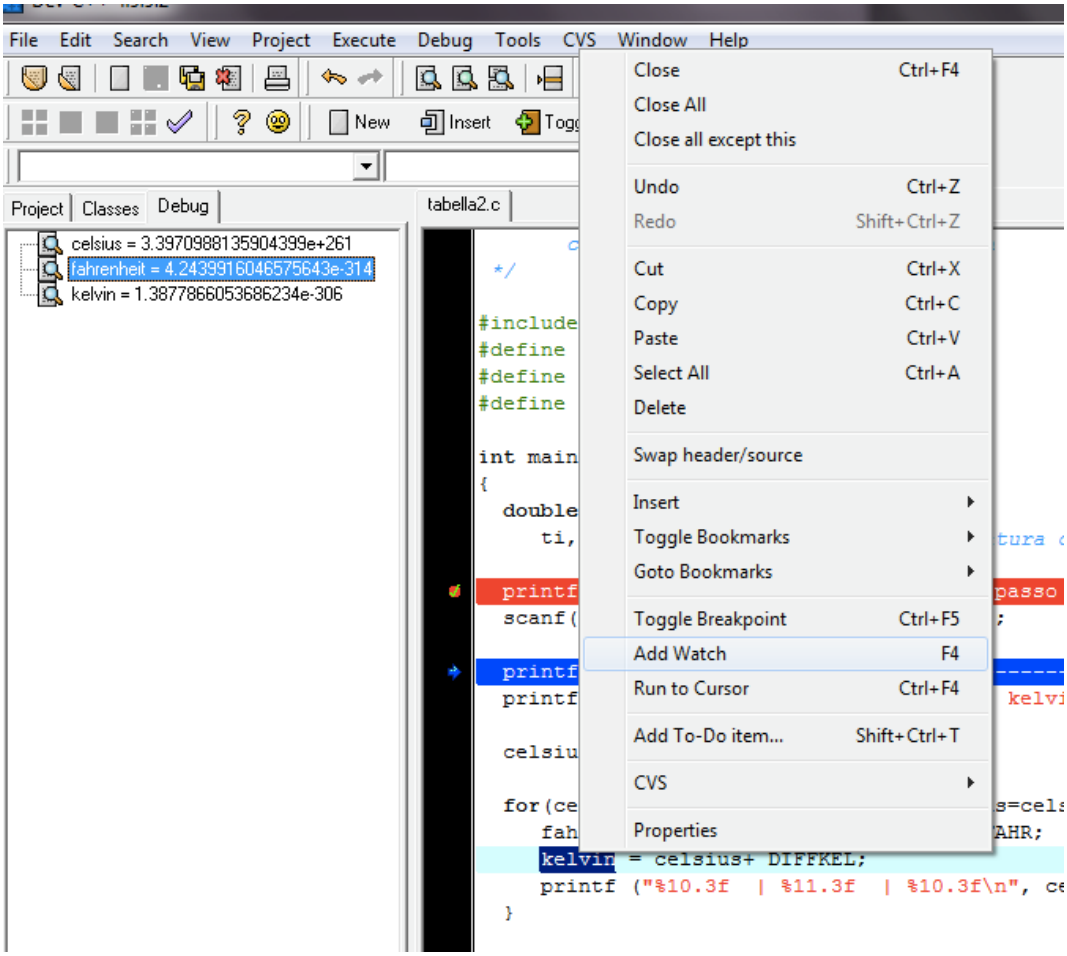
Il programma ha iniziato l'esecuzione e si è fermato sulla linea breakpoint.
Questa ora è blu perché è la prossima linea che verrà eseguita, quando lo chiediamo.

Premendo Next_Step (o *NextLine*, o F7) si avanza eseguendo l'istruzione e posizionandosi sulla successiva.
Ora la istruzione successiva (scanf … diventa blu. È lei che al prossimo Step verrà eseguita.
Nell'eseguirla, il programma si aspetta dati di input e poi prosegue.
Ogni volta che eseguiamo il Next_Step un'istruzione viene eseguita.
Se sulla linea ci sono più istruzioni, tutte verranno eseguite e si passerà alla prossima linea. In effetti è consigliabile mettere una sola istruzione per linea, così vedremo l'esecuzione del programma "una istruzione alla volta" (*passo-passo*).

Nella figura si vede anche il tasto "Add watch" ("Aggiungi Osservazione" in italiano), attraverso il quale si possono ispezionare le variabili durante l'esecuzione del programma (e, vedendo come e se cambiano, capire dove sono gli errori).
Ma questo non e' l'unico modo, inoltre il tutto e' nella prossima sezione…

### *4.2. CONTROLLO DEL VALORE DELLE VARIABILI (WATCH)*

Il valore di espressioni (e quindi anche di variabili del programma) può essere tenuto costantemente sotto controllo mentre si esegue passo-passo il programma. Ciò è utilissimo durante la ricerca di errori di codifica o logici, non rilevabili automaticamente.




Inserire delle *watch* … ad esempio per le variabili `fahrenheit`, `celsius` e `kelvin`. Un modo è visualizzato in figura (si evidenzia la variabile nel codice e si usa il tasto destro per vedere il menù contestuale e selezionare **Add Watch**.

Queste variabili ora sono “sotto osservazione” (*watch* vuol dire “guardare”, e anche “*tenere sotto controllo visivo”*)

Eseguendo il programma passo-passo si vede come il valore delle variabili “*sotto watch”* cambia.

Notate che, tipicamente, prima che qualche istruzione abbia assegnato un valore ad una certa variabile, questa contiene valori non significativi (e imprevedibili).

In questa fase, il tasto F8 (Step Into) ha le stesse funzionalità del tasto F7. (Diventerà significativo e differente più tardi durante questa esercitazione).

Dopo che avrete sperimentato l’esecuzione del programma passo-passo

… seguendo i punti precedenti,

… e ripetendo quel che c’è scritto

*(no … non a voce! Mettendolo in pratica sul computer …)*

… e aggiungendo qualche esperimento che vi viene di fare

*(fateveli venire!*
*Se avete dubbi o volete mettere a punto un esempio che vi piace, chiedete al docente …*
*),*

… dicevo …

“dopo” …

possiamo fare qualche esercizio.

Come quelli di seguito.

### *4.3. I maggiori (funmax.c)*

Scrivere un programma C che legge due numeri interi e stampa il più grande dei due. Però il programma deve far uso di una funzione `maggiore()`, definita dal programmatore, che riceva due parametri interi e restituisca il valore maggiore tra i due parametri.

### *4.4. I minori (funmin.c)*

Scrivere un programma C che legge due numeri interi e stampa il più piccolo dei due. Però il programma deve far uso di una funzione `minore()`, definita dal programmatore, che riceva due parametri interi e restituisca il valore maggiore tra i due parametri.

Nel programma proposto come soluzione vengono messe in pratica due azioni significative:

1) la funzione minore restituisce il risultato usando una tra due distinte `return`:

```
if (primo < secondo)
   return (primo);
else return(secondo);
```

2) la `printf` che stampa, nella `main`, il minore tra i due input, stampa direttamente il risultato della chiamata della funzione minore() (senza dover usare variabili di appoggio come la `m` di `FUNMAX.C`).

```
printf("il...tra i due e' %d\n", minore(num1,num2) );
```

### *4.5. Unità chiamantI e unita chiamatE (funParz.c)*

Chiamare una funzione vuol dire scrivere il nome della funzione, inserendo tra parentesi i parametri attuali che la funzione dovrà usare per fare i suoi calcoli.

Ad esempio **`printf("il...tra i due e' %d\n", minore(num1,num2) );`**

oppure **`maggiore(n,m);`**

oppure **`sqrt(x);`**

Un'istruzione siffatta è una *"chiamata di funzione"* …

Ogni funzione può essere chiamante e/o chiamata:

- ***chiamata*** (nel momento in cui viene chiamata da qualche altra funzione, ad esempio la `main()`).

- ***chiamante*** (nel momento in cui una delle sue istruzioni è la chiamata di una funzione)

Scrivere un programma che legge una **sequenza** di **6 numeri interi** e stampa il più grande.

Il programma però deve essere costruito come segue:

- deve essere definita una funzione `maggiore()` che, ricevendo due parametri interi, restituisce il maggiore;

- deve essere poi definita una funzione `maxSequenza()` che esegue la lettura di 6 numeri e restituisce il massimo tra essi riscontrato; si tratta di una funzione senza parametri, che chiama `maggiore()` per confrontare ogni numero letto con un massimo parziale, secondo l'algoritmo noto;

- infine deve essere definita la main() che chiama maxSequenza() e ne usa il risultato per stampare qual è stato il numero massimo incontrato.

In altre parole, ecco uno schema del programma complessivo:

```
#include<stdio.h>
#define QUANTI_NUMERI 6
... definizione di int maggiore(int primo, int secondo)...
... definizione di int maxSequenza()...
int main ()
{
...
  massimo=maxSequenza();

  printf("---- il massimo numero dato e' %d\n", massimo );

...
return 0;
}
```

*E adesso debugghiamo il programma*

### *4.6. Esecuzione passo passo con F8*

Finora siamo stati capaci di eseguire programmi "istruzione per istruzione" usando F7.

La funzione espletata da F7 è chiamata *trace*, in quanto permette di "tracciare" l'esecuzione del programma, normalmente tenendo sotto controllo il contenuto di tutte o alcune variabili.

Se, quando siamo posizionati su una linea del programma, premiamo F7, l'istruzione corrispondente viene eseguita e ne possiamo vedere gli effetti nella finestra di *watches*.

Se l'istruzione era una chiamata di funzione, Next_Step considera l'esecuzione della chiamata come un unico passo, per cui

- Viene eseguita la funzione, senza mai interrompersi;
- Al termine della chiamata nelle watches possiamo vedere i risultati dell'esecuzione della funzione (ma non abbiamo visto cosa e' successo durante l'esecuzione)
- e la prossima istruzione eseguita sarà quella corrispondente al punto di ritorno dalla chiamata.

Se invece, in corrispondenza di un'istruzione che comporta una chiamata di funzione, usiamo `Step_Into` (F8) il sistema comincia a far vedere l'esecuzione della funzione, istruzione per istruzione.

IN altre parole, usando F8 su una chiamata di funzione, entriamo nel blocco della funzione e lo vediamo mentre viene eseguito, istruzione per istruzione.

**Sperimentare** questa funzionalità sui programmi precedenti: eseguirli passo passo, usando F8 anziché F7. A seconda di cosa usiamo (F8 o F7, in corrispondenza di una chiamata di funzione) vedremo o meno, ad esempio, l'esecuzione passo passo del codice di minore.
Bisogna riuscire a vedere passo-passo l'esecuzione delle funzioni …

### *4.7. ANCORA UNITÀ CHIAMANTI e UNITÀ CHIAMATE (funMinPZ.c)*

Scrivere un programma che legge una sequenza di `n` valori interi, con `n` dato in input, e stampa il valore minimo riscontrato. Il programma però deve essere costruito come segue:

- deve essere definita una funzione `minore()` come sopra;
- deve essere poi definita una funzione `minSeq(int numeroInput)` che esegue la lettura di `numeroInput` numeri interi e ne restituisce il minimo;
- infine deve essere definita la `main()` che legge `n` e chiama `minSeq()`.

### *4.8. Tabella Celsius/Fahreneit/Kelvin (tabella3.c)*

Rieseguire l'esercizio 3.16, in modo che i valori Fahrenheit e Kelvin, relativi ad un dato Celsius, vengano calcolati da una opportuna funzione (ad es. `Fahr()` e `Kelv()`).

### *4.9. Funzione per il massimo comun divisore (funMCD.c)*

Scrivere un programma che legge varie coppie di numeri interi e stampa per ognuna il relativo massimo comun divisore. La funzione `main()` deve usare una funzione `mcd()` che, ricevuti due numeri interi, restituisca il relativo mcd. Il programma termina quando almeno uno dei numeri in una coppia è 0 (zero).

### *4.10. Esperimento sulla visibilità degli identificatori*

Nell'esercizio precedente sul massimo comun divisore la soluzione proposta in `funMCD.c` ha la seguente struttura:

```
int main () {
  int primo,secondo;     /* i numeri di cui trovare il MCD */
...
  while ( (primo!=0) && (secondo!=0) ) {
    printf("mcd tra %d e %d e' %d\n", primo,secondo, mcd(primo,secondo));
...
printf("FINE\n");
return 0;
}
...
int mcd(int n, int m) {

  while (n!=m)
...
  return n;       /* o m ... */
}
```

Provare a sostituire la printf evidenziata sopra con la seguente:

```
printf("mcd tra %d e %d e' %d\n", n,m, mcd(n,m));
```

Cosa succede?

Succede che i simboli `n` ed `m` **non** sono noti nel blocco di istruzioni della `main()` e quindi risultano *undefined*. È ovvio che sia così! L'unico blocco in cui `n` ed `m` sono noti è quello della funzione `mcd()`, in cui questi simboli identificano dei parametri.

### *4.11. Esperimento sul "passaggio di parametri" ad una funzione*

Sempre facendo riferimento alla soluzione proposta in funmcd.c, provare a far girare il programma proposto, mediante un'esecuzione passo passo (con F7 e F8, in modo da tracciare anche il comportamento delle chiamate a mcd()).

Durante queste esecuzioni, tenere sotto controllo gli identificatori **primo**, **secondo, n** ed **m**.

A meno di ritardi nel refresh delle variabili under watch, dovremmo vedere che `n` ed `m` risultano esistere solo mentre stiamo eseguendo la chiamata ad `mcd()`

(cioe' mentre esiste il RDA)

e invece `n` ed `m` sono indefiniti (*not found in current context*) mentre stiamo eseguendo le istruzioni di altre funzioni, come la `main()`.

Analogamente per `primo` e `secondo`

… se non si vede, provare a eliminare e rimettere la watch; ora lo stato della variabile dovrebbe essere aggiornato.

Comunque, se tutte queste quattro locazioni sono elencate tra le *Watches*, possiamo renderci conto che durante l'esecuzione di una chiamata di `mcd(primo, secondo)`

- `n` ed `m` inizialmente hanno i medesimi valori di `primo` e `secondo`,
- poi `n` ed `m` cambiano progressivamente fino a diventare uguali

- ma, quando la chiamata è terminata, `primo` e `secondo` non sono cambiati, cioè hanno conservato il valore che avevano prima della chiamata. Cioè i cambiamenti di `n` ed `m` non si sono riflessi su `primo` e `secondo`.

Il fatto che `primo` e `secondo` (i **parametri attuali** della chiamata di `mcd()`) mantengano il loro valore (e non cambino come fanno `n` ed m) è una conseguenza delle modalità del passaggio dei parametri nelle funzioni C (*passaggio per valore*).

Quando `mcd(primo, secondo)` viene attivata, nel record di attivazione vengono allocate due locazioni per n ed m (i parametri formali). In queste locazioni vengono copiati i valori dei parametri attuali primo e secondo. Ogni uso che facciamo di n ed m nella funzione `mcd()` si riflette sulle locazioni allocate per loro nel record di attivazione e non su quelle dei parametri attuali.

*NB [[Le locazioni di memoria dedicate ai parametri formali sono INTERNE al RDA; le locazioni di memoria deicate ai parametri attuali sono evidentemente accessibili solo alla funzione chiamante e quindi ESTERNE al RDA.]]*

### *4.12. Funzione per la media*

Scrivere un programma che esegua varie volte le seguenti operazioni:

1) lettura di un numero `n`
2) lettura di n numeri interi e calcolo della relativa media
3) stampa della media calcolata al punto 2)

L'operazione di cui al punto 2) deve essere eseguita da una funzione opportuna. Il programma termina quando viene letto un valore 0 per n.

### *4.13. Funzione per la media (2)*

Come sopra, ma facendo in modo che la funzione `main()` stampi, alla fine, il valore massimo riscontrato tra tutte le medie.

### *4.14. Array e medie (arrmed.c, arrMed2.c)*

Scrivere un programma C il cui Input sia costituito da una sequenza di N numeri frazionari. Ad esempio, N potrebbe valere 6. È bene che N sia un simbolo di costante.

L'Output del programma deve consistere nella stampa del valore medio dei numeri dati in input e nella stampa dei medesimi numeri, incrementati della media appena menzionata.

Per risolvere il problema è indispensabile mantenere in memoria i dati letti da input. Per far ciò si suggerisce di usare un array di N double. Per il momento è anche consigliabile fare a meno della definizione di funzioni (a parte la main()!).

```
 fornire i 6 numeri frazionari, grazie
2.5
3.64
47.42
42.47
61.61
2005
 la media dei numeri proposti e': 360.44
 ecco i numeri proposti, incrementati di 360.44
362.940000
364.080000
407.860000
402.910000
422.050000
2365.440000
FINE
```

Il secondo programma soluzione per questo esercizio fornisce una soluzione più furba, rispetto al primo, per il medesimo problema.

### *4.15. Commercio al minuto (NEGOZI.C)*

Realizzare un programma che risolva il problema della società commerciale:

Una società amministra 20 negozi (che si immaginano distinti solo in base ad un numero: negozio 7, negozio 12 ...).

Considerando come dati di Input i guadagni mensili dei negozi, il programma deve

- calcolare e stampare la media dei guadagni;
- stampare quali negozi (e con quali guadagni) guadagnano meno di un terzo della media;
- calcolare e stampare la media dei guadagni escludendo il massimo e minimo guadagno (chiamiamola *media2*);
- stampare quali negozi (e con quali guadagni) guadagnano meno di un terzo della media2, cioè si configurano come le pecore nere del gruppo (e i cui gestori stanno per passare un brutto momento ...).

L'output che dovrebbe essere prodotto dal programma è mostrato di seguito

```
fornire il guadagno mensile del negozio 0: 321
fornire il guadagno mensile del negozio 1: 198
fornire il guadagno mensile del negozio 2: 12
fornire il guadagno mensile del negozio 3: 1261
fornire il guadagno mensile del negozio 4: 99
fornire il guadagno mensile del negozio 5: 312
(media=367.167) ecco i negozi che guadagnano poco:
- negozio 2 con Euro 12
- negozio 4 con Euro 99
(media2=232.5) ecco le pecore nere:
*** negozio 2 con Euro 12
FINE
```

Anche in questo problema è indispensabile mantenere in memoria i dati letti da input.

Per far ciò si suggerisce di usare un array di N double.

E anche qui, per il momento è consigliabile fare a meno della definizione di funzioni ulteriori alla main().

### *4.16. Gli errori dei commercianti (NEGOZERR.C)*

Per risolvere il problema di cui al punto precedente il docente, in un momento di sconforto, ha realizzato un programma in NEGOZERR.C.

Il programma contiene degli errori (del programmatore, non dei commercianti).
Quali?

Aiutate il docente a trovarli … Fare un po' di attività di debugging, se non si riconoscono gli errori a prima vista ...

### *4.17. Funzioni con parametri array*

Qui ci esercitiamo con funzioni che ricevono array attraverso parametri …
Usiamo array di dimensione N, dove N è una costante simbolica.
Poi applicheremo le capacità allenate al problema dei negozi; per ora invece miglioriamo programmi già fatti.

Scrivere una funzione C `accumulazione()` che,

- Ricevendo tramite parametro un array (`g[]`) di N double (dove N è una costante simbolica),
- calcoli e restituisca la somma degli elementi di `g[]`.

Fare una copia del file ARRMED.C in un nuovo file ARRMEDNUOVO.C, e incastonare nel programma la funzione `accumulazione()` usandola per il calcolo della media dei valori letti da input e per la successiva ristampa dei medesimi valori incrementati di media.

Magari facciamo che N non sia troppo grande … 5, 6, …

Testare il programma su diversi input, cercando di variare gli input … per esempio usare anche numeri negativi e nulli …

### *4.18. Funzioni con parametri array – seconda parte (negFun1.c)*

Scrivere le seguenti funzioni C:

- `massimo()` che, ricevuto un array di N double, calcola e restituisce il valore del massimo dei suoi elementi;
- `minimo()` che, ricevuto un array di N double, calcola e restituisce il valore del minimo dei suoi elementi;

notare che per scrivere queste due funzioni, sulla carta, prima scrivendo l'algoritmo, poi l'intestazione e poi la definizione completa, non c'è realmente bisogno di avere in mente un problema particolare in cui usarle … si tratta di sottoproblemi abbastanza generali …

Per testare le funzioni, usarle nella risoluzione del problema del commercio al minuto: modificate il programma con cui avete risolto quel problema, in modo che usi queste funzioni.

### *4.19. Funzioni con parametri array – terza parte (negFun2.c)*

Scrivere le seguenti funzioni C:

- `accumulazione(), massimo(), minimo()` (ne abbiamo già vista la specifica ...);

- `stampaNegoziScarsi()` che, ricevuti
  - un array di N double, `g[]`, che rappresenta i guadagni dei negozi,
  - e un numero double, `soglia`

  stampi il numero d'ordine e il guadagno dei negozi che guadagnano meno di `soglia`

Usare le suddette funzioni nella risoluzione del problema dei guadagni dei negozi.

In particolare `stampaNegoziScarsi` sarà una funzione `void` (dato che deve solo produrre delle stampe e non deve "restituire" valori particolari).

Essa verrà chiamata due volte:

- una volta per stampare i negozi che guadagnano meno di un terzo di `media`
- e un'altra per stampare i negozi che guadagnano meno di un terzo di `media2`.

### *4.20. Funzioni con parametri array – quarta parte (negFun3.c)*

Scrivere una funzione C `lettura()` che,

- ricevendo un array `g[]` di N double (dove N è una costante simbolica)
- legga N valori da tastiera e li memorizzi nell'array

Usare questa funzione per scrivere un'altra versione del programma sul problema dei guadagni dei negozi. In quest'ultima versione, si useranno anche tutte le altre funzioni definite nei due esercizi precedenti.

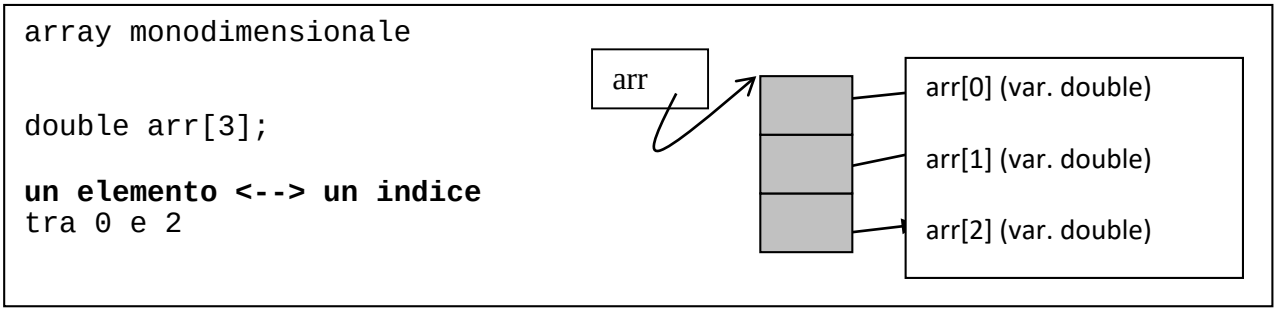
### *4.21. Array bidimensionali*

Quel che si è visto finora, è che un array è una collezione di un certo numero N di variabili di un certo tipo.

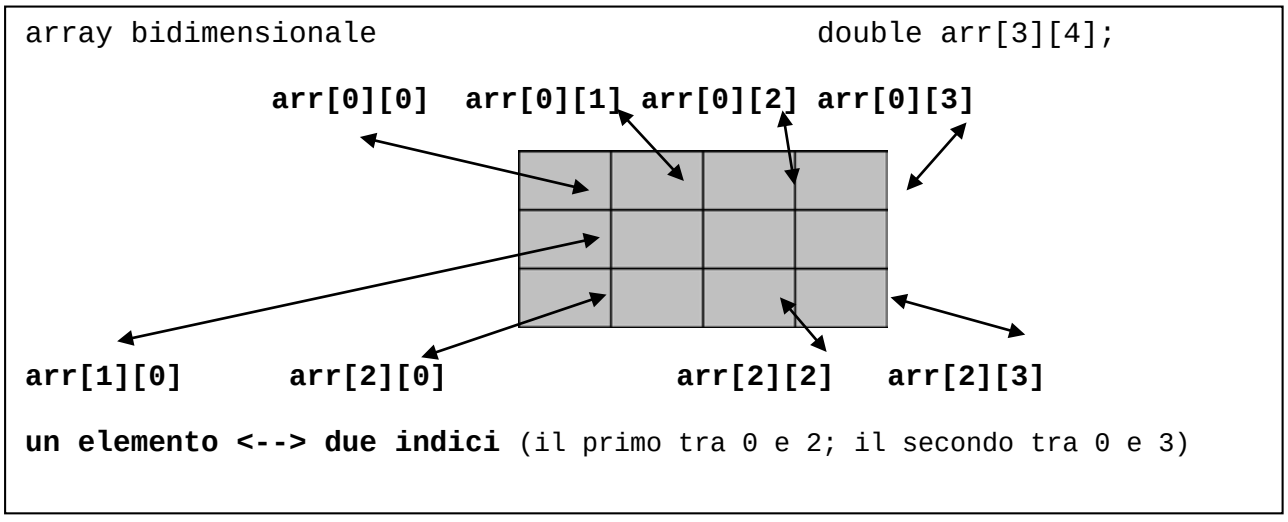
Le variabili sono allocate (hanno spazio riservato in memoria) sequenzialmente in un blocco di memoria centrale.

Il nome dell'array indica l'indirizzo dell'inizio del blocco.

Ogni variabile è identificata da un indice (intero da 0 a N-1).




Si è visto poi, sul libro e/o a lezione, che è possibile definire collezioni di elementi in cui ciascun elemento è associato a una coppia di indici; queste strutture dati sono sempre array, solo che si chiamano array bi-dimensionali.




Con questo costrutto si rappresentano agevolmente le matrici. In un array bidimensionale è infatti facile riconoscere una organizzazione basata su "righe" e "colonne". Nella matrice di esempio,

**arr[0][0], arr[0][1], arr[0][2], arr[0][3]** sono gli elementi della **riga 0;**

**arr[1][0], arr[1][1], arr[1][2], arr[1][3]** sono gli elementi della **riga 1;**

**arr[2][0], arr[2][1], arr[2][2], arr[2][3]** sono gli elementi della **riga 2;**

### *4.22. Esercizio (MATSOMM1.C)*

Scrivere un programma che fa uso di tre matrici di NxM elementi interi (porre N ed M a numeri sufficientemente piccoli – 3, 4 – altrimenti si dovranno immettere troppi dati da tastiera!).

Il programma deve leggere due matrici da input, stamparle, calcolare la somma delle due matrici (memorizzandola nell aterza matrice) e stamparla.

```
 lettura prima matrice:
 - riga 0: fornire 4 valori interi:
1 2 3 4
 - riga 1: fornire 4 valori interi:
5 6 7 8
 - riga 2: fornire 4 valori interi:
9 10
11
12
 lettura secnda matrice:
 - riga 0: fornire 4 valori interi:
10 20 30 40
 - riga 1: fornire 4 valori interi:
10 20 30 40
 - riga 2: fornire 4 valori interi:
40 30 20 10
 prima matrice:
    1    2    3    4
    5    6    7    8
    9   10   11   12
 seconda matrice:
   10   20   30   40
   10   20   30   40
   40   30   20   10
 matrice somma:
   11   22   33   44
   15   26   37   48
   49   40   31   22
FINE
```

(suggerimento segue)

**Suggerimento**: intanto non usare funzioni. Scrivere solo la definizione della funzione main() e delle costanti N ed M; inoltre, se è definito l'array **`int mat1 [N][M]`**,

- iterando per j=0...M-1 l'istruzione **`scanf("%d", &mat1[0][j])`** si leggono gli M elementi della prima riga di `mat1` (riga 0);
- iterando per j=0...M-1 l'istruzione **`scanf("%d", &mat1[1][j])`** si leggono gli M elementi della seconda riga di `mat1`;
- ...
- iterando per j=0...M-1 l'istruzione **`scanf("%d", &mat1[N-1][j])`** si leggono gli M elementi dell'ultima riga di `mat1`;

Quindi il ciclo

```
for (j=0; j<M; j++)
   scanf("%d", &mat1[i][j])
```

e questa lettura della riga i-esima va ripetuta per i valori di i da 0 a N-1, in modo da leggere, una riga per volta, tutta la matrice.

### *4.23. Esercizio (MATSOMM2.C)*

Identico esercizio del punto precedente! Però adesso bisogna definire tutte le funzioni necessarie per far funzionare il programma con la seguente funzione main():

```
int main ()
{
      ...
  printf(" lettura prima matrice:\n");
  leggiMatrice(mat1);

  printf(" lettura secnda matrice:\n");
  leggiMatrice(mat2);

  sommaMatrici(mat1, mat2, mat3);

  printf(" prima matrice:\n");
  stampaMatrice(mat1);

  printf(" seconda matrice:\n");
  stampaMatrice(mat2);

  printf(" matrice somma:\n");
  stampaMatrice(mat3);
...
}
```

### *4.24. Condensazione delle righe di una matrice (condens.c)*

Scrivere una funzione `condensa()` che, ricevuti un array arr e una matrice di NxM elementi double, riempie l'array in modo che l'elemento arr[i] corrisponda, per ogni i, alla somma della riga i-esima della matrice.
Nel file **`tCondens.c`** c'è una versione incompleta del programma che risolve questo esercizio. Questa versione può essere usata per collaudare la vostra soluzione, cioè la funzione `condensa()` che avete prodotto:

- Scrivete la funzione **`condensa()`**
- Piazzatela in fondo al file **`tCondens.c`**
- Eseguite il programma. Se la funzione **`condensa`**() e' scritta bene il programma ora funzionerà bene.

Negli esempi di output qui sotto, quello a sinistra è chiaro e pulito, mentre in quello a destra l'utente si è preso la libertà di definire due righe tutte sulla medesima linea di input … si produce un output un po' strambo, ma comunque le operazioni di input sono andate bene e la condensazione finisce bene lo stesso …

```
 lettura matrice:
 - riga 0: fornire 4 valori interi:
10 20 30 40
 - riga 1: fornire 4 valori interi:
50 60 70 80
 - riga 2: fornire 4 valori interi:
90 100 110 120
 la matrice data e':
   10   20   30   40
   50   60   70   80
   90  100  110  120
 e viene condensata nel vettore seguente:
  100
  260
  420
FINE
```

```
 lettura matrice:
 - riga 0: fornire 4 valori interi:
10 20 30 40
 - riga 1: fornire 4 valori interi:
50 60 70 80 90 100 110 120
 - riga 2: fornire 4 valori interi:
 la matrice data e':
   10   20   30   40
   50   60   70   80
   90  100  110  120
 e viene condensata nel vettore seguente:
  100
  260
  420
FINE
```

### *4.25. Condensazione delle colonne (!) di una matrice (condCol.c)*

Scrivere una funzione `condensaColonne()` che,

- ricevuti un array `arr` e una matrice di NxM elementi double,
- riempie l'array in modo che l'elemento `arr[k]` corrisponda, per ogni `k`, alla somma della **colonna** k-esima della matrice.

Scrivere anche un intero programma per collaudarla. (Si può sfruttare pesantemente il programma scritto per l'esercizio precedente, con qualche modifica alla funzione di stampa del vettore ... e con l'aggiunta della funzione qui richiesta).

```
 lettura matrice:
 - riga 0: fornire 4 valori interi:
10 20 30 40
 - riga 1: fornire 4 valori interi:
50 60 70 80
 - riga 2: fornire 4 valori interi:
90 100 110 120
 la matrice data e':
   10   20   30   40
   50   60   70   80
   90  100  110  120
 e viene condensata nel vettore seguente:
  150  180  210  240
FINE
```

Suggerimento segue

**Suggerimento**: se mat è un array bidimensionale NxM, gli elementi della sua colonna j-esima sono

**mat[0][j], mat[1][j], ..., mat[N][j]**

quindi lo schema per scandire, fissato j, tutti gli elementi della colonna j è del tipo

```
for (i=0; i<N; i++)
  processazione di mat[i][j]
```

### *4.26. Prodotto tra matrici (prodMat.c)*

Scrivere una funzione prodMat() che,

- ricevuti due array bidimensionali NxN
- restituisce, in un opportuno parametro di output, il prodotto tra le due corrispondenti matrici.

Per collaudare la funzione, inserirla nel file TPRODMAT.C, in cui c'è già un programma capace di usarla.

**Suggerimento 1**:
se $(a_{ij})$ e $(b_{ij})$ sono due matrici (i, j = 1..N),
la matrice prodotto è $(c_{ij})$ con

per ogni i, j = 1...N

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$$

**Suggerimento 2**: potremmo definire tre array bidimensionali che rappresentano le medesime matrici,

```
int prima[N][N], int seconda[N][N], int result[N][N]
```

considerando che, in questi array, gli indici andranno da 0 a N-1,
l'elemento generico della matrice risultato sara'

```
result[i][j] = somma dei prodotti
               prima[i][k]* seconda[k][j]
               con k = 0... N-1
```

suggerimento 2bis segue …

**Suggerimento 2bis**: rifrasando il suggerimento 2 …

```
result[i][j] = per k che va da 0 a N-1 compreso
               accumula i prodotti
               prima[i][k] * seconda[k][j]
```

ring a bell?

Suggerimento 3 segue …

**Suggerimento 3**:
nel suggerimento 2 abbiamo visto come calcolare il valore da assegnare a `risult[i][j]`,

questo calcolo bisogna farlo per tutti i `risult[i][j]` (cioe' per tutti gli elementi della matrice prodotto `risult`).
Quindi lo schema risolutivo per sarà

```
for (i=0; i<N; i++)
   for (i=0; i<N; i++)
      calcolo e assegnazione di result[i][j]
```

dove **calcolo e assegnazione di result[i][j]** corrisponde a quanto detto nel suggerimento 2.

### *4.27. Centraline inquinate (misurazioni.c)*

Bisogna monitorare lo stato di inquinamento di Via Andrea Doria, approssimativamente davanti al n.3. Lì potrebbe venir posta una centralina che misura e memorizza la quantità di idrocarburi incombusti presente nell'aria (in microgrammi).

Vengono fatte misurazioni per tutto il giorno e, alla fine, viene stampato un istogramma che descrive, ora per ora, la media di microgrammi presenti.

La media, se non è un intero esatto, viene troncata.

Ogni ora vengono eseguite 6 misurazioni; la media di queste misurazioni è la media oraria di quell'ora. Quando una opportuna struttura di memoria è stata riempita con le medie orarie calcolate nel giorno, viene stampato un istogramma come nella seguente schermata:

```
lettura dati:
da  0:00 a  0:59 ***************************
da  1:00 a  1:59 *******************************
da  2:00 a  2:59 ************************************
da  3:00 a  3:59 *******************************
da  4:00 a  4:59 ******************************
da  5:00 a  5:59 *************************
da  6:00 a  6:59 ********************
da  7:00 a  7:59 ************************
da  8:00 a  8:59 ****************
da  9:00 a  9:59 ***********************************
da 10:00 a 10:59 ********************************
da 11:00 a 11:59 *******************************
da 12:00 a 12:59 ******************************
da 13:00 a 13:59 *************************
da 14:00 a 14:59 ****************************************
da 15:00 a 15:59 ********************************************
da 16:00 a 16:59 *******************************
da 17:00 a 17:59 ************************
da 18:00 a 18:59 ***********************************
da 19:00 a 19:59 ********************************
da 20:00 a 20:59 ******************************
da 21:00 a 21:59 ********************
da 22:00 a 22:59 *********************
da 23:00 a 23:59 *********************************
FINE
```

Assumiamo che il numero di microgrammi misurabile ogni volta vada da 0 a 61 (così le righe dell'istogramma non saranno mai troppo lunghe.

Suggerimenti seguono nelle pagine successive …

**Suggerimento 1**
Usare un array di 24 componenti intere;

**Suggerimento 2**
Ogni 6 misurazioni, la media (troncata) viene messa in uno degli elementi dell'array.

**Suggerimento 3**
Definire una funzione `misurazione()` che, quando viene chiamata, restituisca una misurazione. Questa funzione può consistere in una semplice lettura da input di un intero, oppure nella determinazione di un valore calcolato lì per lì in qualche modo: inserendo il valore da input possiamo mettere valori significativi, e anche fare esperimenti significativi, magari con dati reali … ma dobbiamo lavorare noi ☺; se i valori vengono calcolati "lì per lì" lavora la macchina e possiamo sperimentare il programma più volte (anche per vedere se funziona bene).

**Suggerimento 4**
Per calcolare i valori "lì per lì" potremmo usare la funzione `rand()` della libreria `stdlib.h`.
Una chiamata `rand()` produce un valore intero pseudocasuale.
Inoltre, l'espressione

```
rand()%k
```

produce un valore pseudocasuale compreso tra 0 e k (ricordiamoci che nel nostro caso vogliamo numeri da 0 a 61...)

**Suggerimento 5**
Per inizializzare il meccanismo di generazione pseudocasuale dei numeri da parte di rand() bisogna aver chiamato almeno una volta, prima della prima chiamata di rand(), la funzione `srand(),` che prende un parametro `unsigned int`. Un modo carino è di eseguire la chiamata `srand(clock())`…

**Suggerimento 6**
La chiamata `clock(),` restituisce il numero di "colpi di clock" eseguiti dal calcolatore dal momento dell'accensione. Quindi è un buon numero imprevedibile con cui inizializzare il processo di generazione pseudocasuale.
Per usare `clock()` bisogna aver incluso la libreria `time.h.`
Ulteriori info su `rand(), srand()` e `clock()` sono nell'help del DEV ...

**Suggerimento 7**
Nel programma che e' disponibile come soluzione, sono definite le costanti simboliche

```
#define PUNTO '*'
#define ESTREMO 61
#define DIM 24
```

la funzione misurazione() viene definita cosi'

```
int misurazione(int estremo) {
    return(rand() % estremo);
}
```

e usata cosi'

```
...
     for (j=0; j<6; j++) {        /* lettura di 6 dati nell'ora */

        dato = misurazione(ESTREMO);    /* lettura */
...
...
```

**Suggerimento 8**
Per stampare l'istogramma, viene usata una funzione

```
void stampaIstogramma (int arr[], char c)
```

che per ogni componente di arr, arr[i],
stampa una linea di caratteri c lunga arr[i]

suggerimento 9 segue, con una stampa diversa dell'istogramma

**Suggerimento 9**
In alternativa, per stampare l'istogramma, viene usata una funzione

```
void stampaIstogramma2 (int arr[], char c, int inizio, int fine)
```

che

per ogni componente di arr, arr[i],
con i compreso tra inizio e fine,
stampa una linea di caratteri c lunga arr[i].

```
lettura dati:
stampa dati: fornire ora inizio e fine (es. 8 23): 10 18
da 10:00 a 10:59 **********************************
da 11:00 a 11:59 ********************************
da 12:00 a 12:59 ****************************
da 13:00 a 13:59 ************************
da 14:00 a 14:59 ****************************************
da 15:00 a 15:59 ********************************************
da 16:00 a 16:59 ********************************
da 17:00 a 17:59 ************************
FINE
```

(Perche' e' necessario ideare anche la seconda versione della stampa istogramma?

perché con la prima versione, quando viene stampato l'istogramma in output, è molto probabile che una larga parte iniziale dell'istogramma scompaia mentre la parte restante viene stampata. La finestra di output non ci consente di vedere molte linee ...
Con stampaIstogramma2() è possibile risolvere parzialmente il problema: se la chiamata contiene l'indicazione di un intervallo di misurazioni non troppo esteso, tutto l'istogramma prodotto nella finestra di output rimarrà visibile – che bello!).