Laurea In Ingegneria dell'Informazione

Esercitazioni Guidate di Tecniche della Programmazione

Note introduttive: Seguire le raccomandazioni date nelle precedenti EG …

4. Esercitazione 4 (prima parte)

4.1. ESECUZIONE PASSO-PASSO

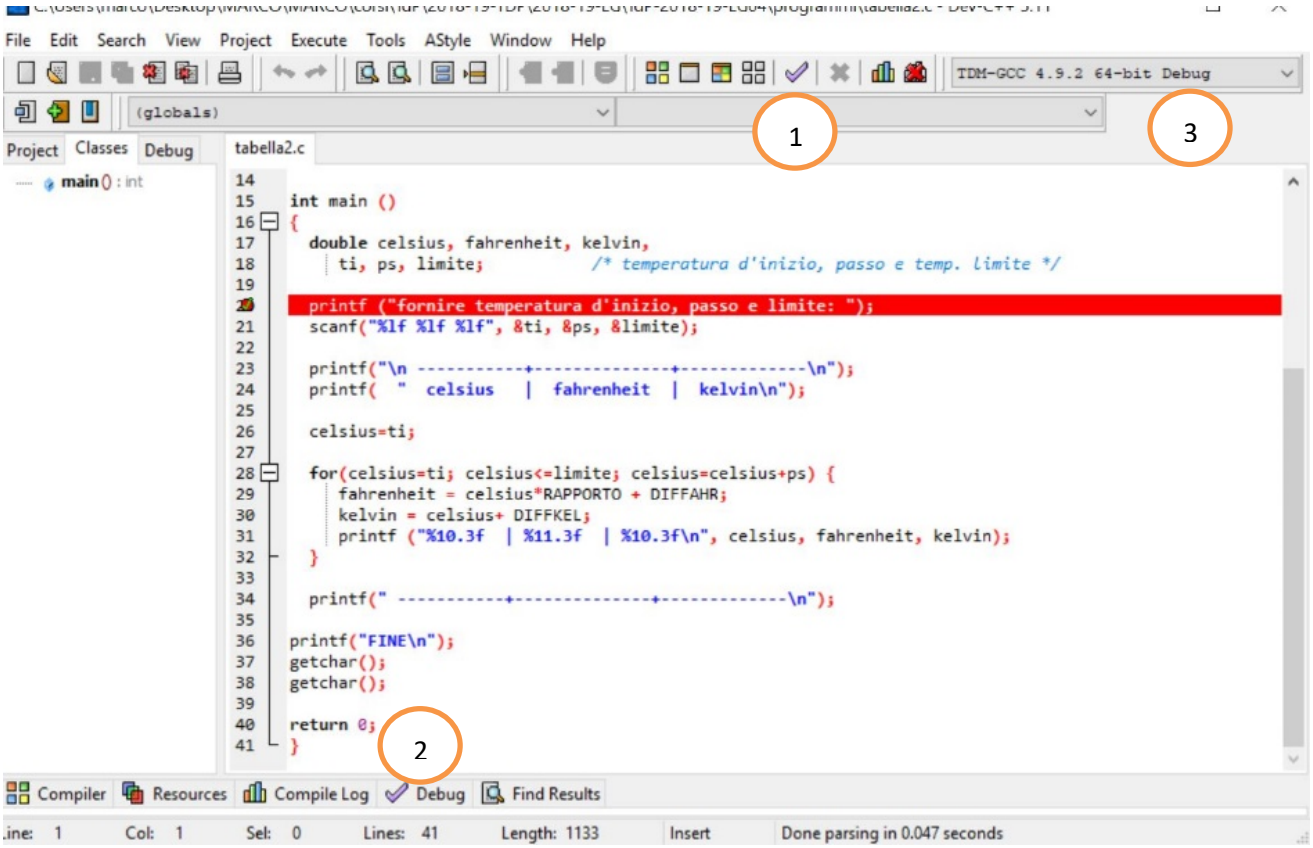
Riprendere il programma tabella2.c.
Adesso ci esercitiamo sull'uso del Debugger.
In DEV è possibile aggiungere al programma compilato informazioni utili per eseguire poi il programma passo-passo, vedendo come si comporta l'esecuzione. Questo e' utile quando si deve correggere un programma: si esegue 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.



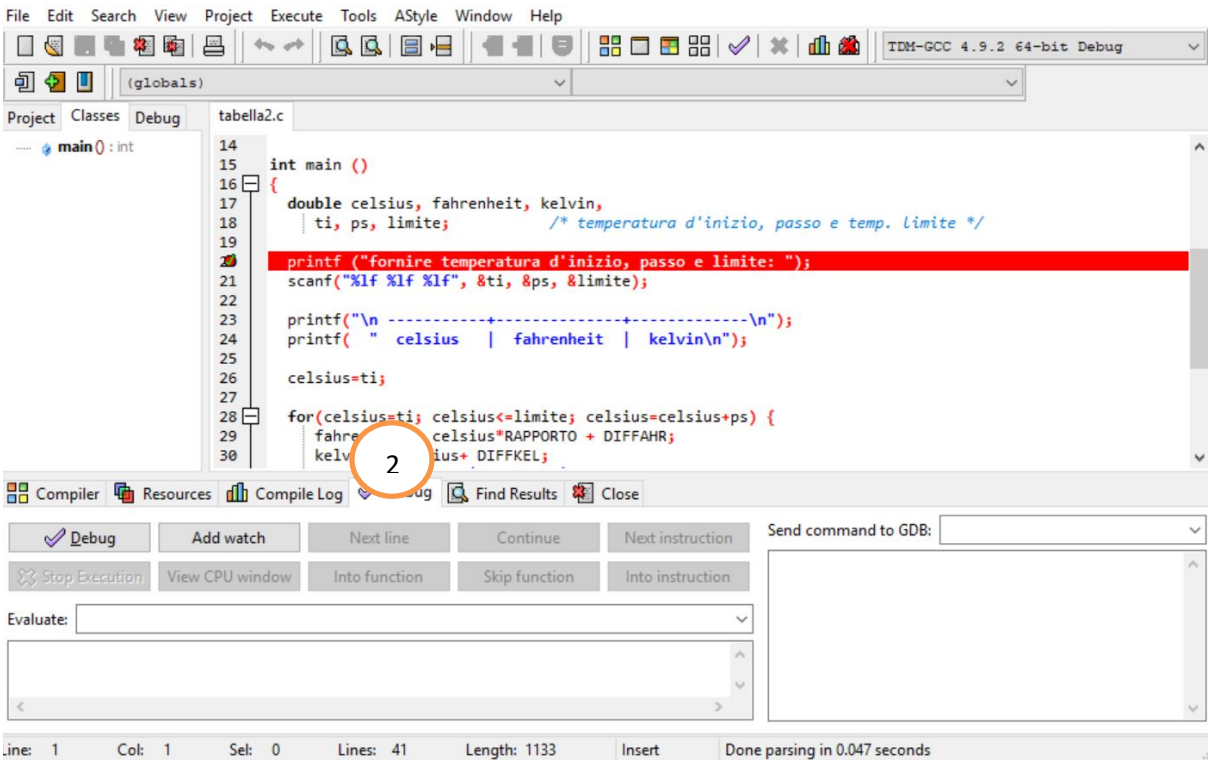
… ma ora il programma non è ancora in esecuzione.
Se lo compiliamo e mandiamo in esecuzione come facciamo di solito, lo vedremo funzionare, ma non vedremo la sua esecuzione in *modalità Debug*.

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

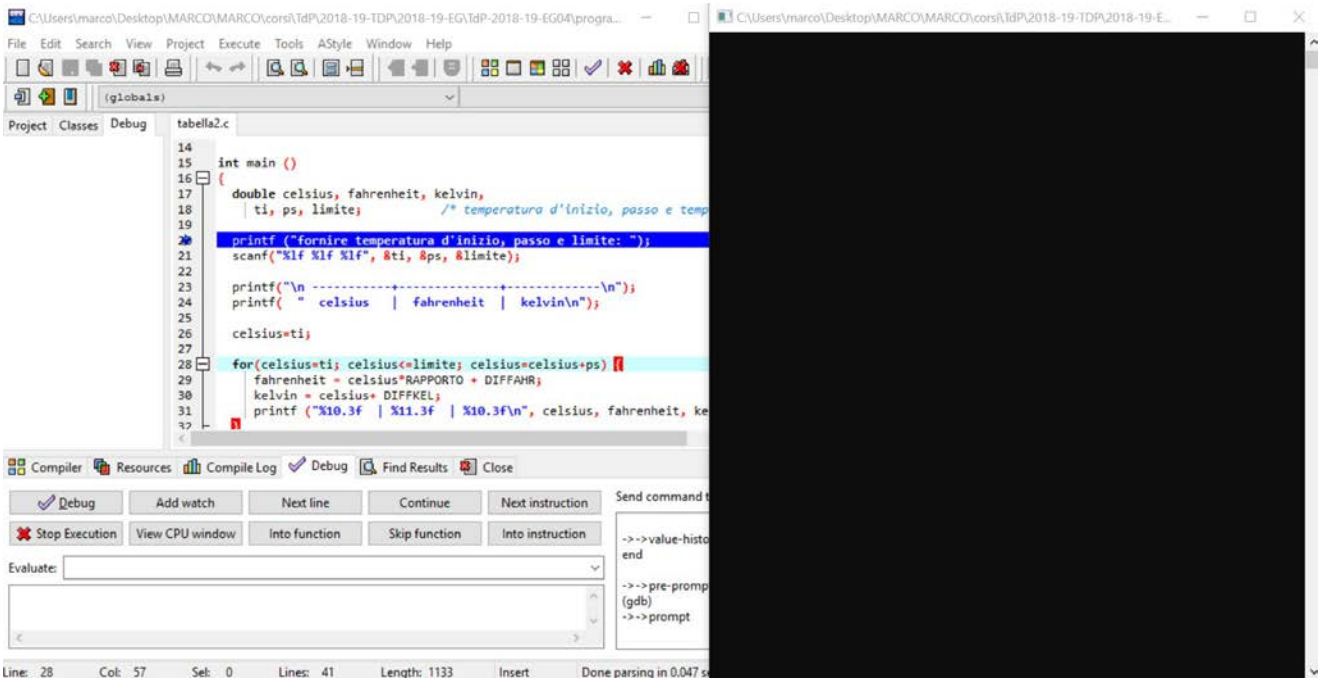
2) oppure aprendo il pannello di debug mediante clic sull'opzione corrispondente nella barra in basso (quella che si apre per farci vedere i risultati della compilazione). Se qui si clicca sul bottone segnalato con (2), si apre la porzione di finestra dedicata al *debugging* (vedi figura seguente).

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 …]



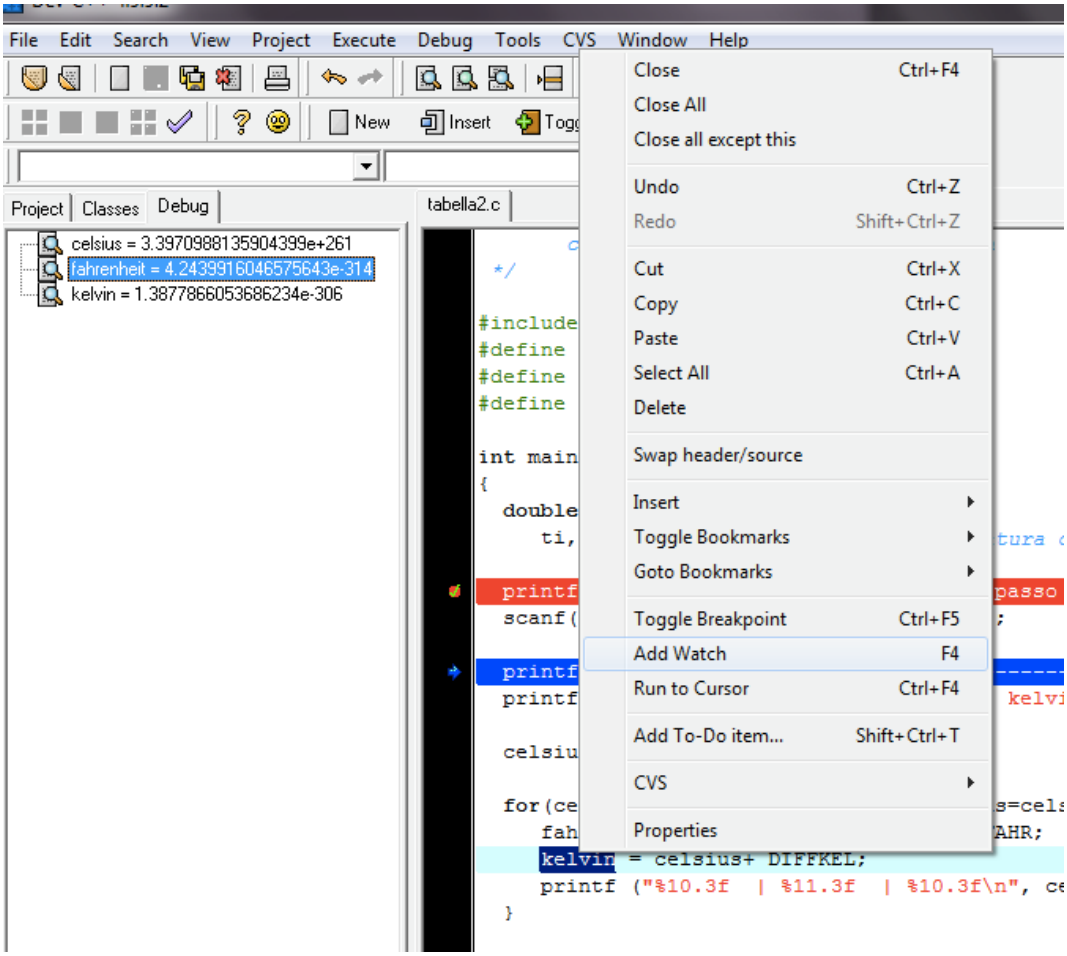
Il programma ha iniziato l'esecuzione e si è fermato sulla linea breakpoint.
Questa ora è blu perché è la prossima linea che verrà eseguita, quando lo chiediamo.

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.

Continua nella seconda parte …