Capitolo 1

Lezione 19 febbraio 2019

1.1 Come compilare ed eseguire un programma senza Makefile

1. Aprire un terminale

2. Impostare come directory corrente quella contenente i file sorgente.

   Esempio:
   cd /home/silvestri/labprog/lezione1

3. Eseguire gcc
   `gcc -ansi -pedantic -Wall -o NOME_ESEGUIBILE NOME_SORGENTE.c`

   Esempio:
   gcc -ansi -pedantic -Wall -o hello hello.c

   Significato delle opzioni: -ansi
   disabilita alcune funzionalità di GCC che non sono compatibili con ISO C90

   -pedantic
   impedisce la compilazione di codice C non compatibili con ISO C (nella varsione specificata, nel nostro caso ANSI C90)

   -Wall
   abilita tutti i warning (presenti in gcc) relativi a condizioni che è consigliabile evitare. Ad esempio la dichiarazione di variabili successivamente non utilizzate.

4. Eseguire il programma
   `./NOME_ESEGUIBILE`

   Ad esempio:
   `./hello`

5. Nel caso vengano utilizzate librerie aggiungere le relative opzioni DOPO I FILE SORGENTE CHE LE UTILIZZANO (aggiungerle alla fine della linea di comando solitamente è una scelta sicura)

   Per le funzioni matematiche, ad esempio, è necessario includere math.h nel file sorgente ed aggiungere l'opzione -lm :
   `gcc -ansi -pedantic -Wall -o pitagora pitagora.c -lm`

   Nel caso il vostro programma si articoli in più di un file sorgente, è possibile eseguire compilazione e link con un unico comando, come nel caso di programmi composti da un unico file sorgente, indicando tutti i file sorgenti necessari. Attenzione alla posizione dei flag per le librerie, si riferiscono a tutti i file che li precedono nella linea di comando (tranne in alcuni casi particolari ed inusuali, la posizione preferibile per i flag è alla fine della linea di comando). Ad esempio:
   `gcc -ansi -pedantic -Wall -o area_triangolo pitagora.c triangolo.c -lm`

Come nei casi visti precedentemente, l'esecuzione del programma consisterà nell'invocazione della funzione `main`, quindi è importante che ne esista una (ed una sola). La presenza di più funzioni `main` (anche in file diversi) o l'assenza saranno indicate dal linker con un messaggio di errore.

1.2 Esercizio: Hello

Scrivere 10 linee contenenti il numero della linea ed il testo Hello

Soluzione

```c
#include <stdio.h>

int main(){
  int i;
  for (i=0; i<10; i++) {
    printf("%d: Hello!\n",i);
  }
  return 0;
}
```

Output

```
0: Hello!
1: Hello!
2: Hello!
3: Hello!
4: Hello!
5: Hello!
6: Hello!
7: Hello!
8: Hello!
9: Hello!
```

1.3 Esercizio: Somma numeri naturali

Calcolare la somma dei numeri naturali minori o uguali ad un numero n inserito dall'utente

Soluzione

```c
#include <stdio.h>

int main(){
  int somma = 0;
  int i, n = 0;
  printf("Inserisci un numero: \n");
  scanf("%d",&n);
  for ( i=1 ; i<=n ; i++ ) {
    somma += i;
  }
  printf("\nSomma = %d\n", somma);
  return 0;
}
```

Output

```
Inserisci un numero:
30

Somma = 465
```

1.4 Esercizio: casting tipi numerici

Confrontare i valori assegnati alle variabili nei seguenti casi e spiegare l'origine delle differenze.
ATTENZIONE: QUESTO VI DOVREBBE AIUTARE AD EVITARE UN TIPO DI ERRORE FREQUENTE NELL'ESERCIZIO SEGUENTE

```c
#include <stdio.h>

int main(){
  int n=123456;
  double a = 1/(n*n);
  double b = 1.0/(n*n);
  double c = 1.0/((double)n*n);
  printf("n=%d\n a=%.16f\n b=%.16f\n c=%.16f\n\n",n,a,b,c);
  return 0;
}
```

1.5 Esercizio: serie armonica generalizzata ($\alpha = 2$)

Somma degli inversi dei quadrati dei primi n numeri naturali Calcolare la somma degli inversi dei quadrati dei numeri naturali minori o uguali ad un numero n inserito dall'utente. Calcolare inoltre la differenza rispetto a

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \cdots = \frac{\pi^2}{6}$$

Come valore di π utilizzare il seguente (copiare la linea contenente #define all'inizio del file sorgente)

```c
#define PI 3.14159265358979323846

/* es: */
double limite = PI * PI / 6;
```

Soluzione

```c
#include <stdio.h>
#define PI 3.14159265358979323846

int main(){
  int i,n=10;
  double risultato = 0.0;

  printf("Inserire il numero di termini: ");
  scanf("%d", &n);

  for (i=1 ; i<=n ; i++) {
    risultato += 1.0 / ((double)i*i);
  }

  printf("Risultato: %.15f\nLimite: %.15f\nDifferenza: %.15f\n",risultato, PI*PI/6, risultato-PI*
    PI/6 );
  return 0;
}
```

Output

```
Inserire il numero di termini: 10000
Risultato: 1.644834071848065
Limite: 1.644934066848226
Differenza: -0.000099995000161
```

1.6 Esercizio: serie armonica generalizzata ($\alpha = 2$, solo termini dispari)

Somma degli inversi dei quadrati dei primi n numeri naturali dispari Calcolare la somma degli inversi dei quadrati dei primi n numeri naturali dispari, con n numero intero inserito dall'utente.

Soluzione

```c
#include <stdio.h>

int main(){
  int i,n=10;
  double risultato = 0.0;

  printf("Inserire il numero di termini: ");
  scanf("%d", &n);

  for (i=1 ; i<=n ; i++) {
    /* i-esimo numero dispari  = i*2-1 */
    risultato += 1.0 / ((double)(i*2-1)*(i*2-1));
  }

  printf("Risultato: %.15f\n",risultato );
  return 0;
}
```

Output

```
Inserire il numero di termini: 10000
Risultato: 1.233675550136196
```

1.7 Esercizio: serie armonica generalizzata ($\alpha = 2$, solo cubi perfetti)

Somma degli inversi dei quadrati dei primi n numeri naturali che siano il cubo di un numero naturale Calcolare la somma degli inversi dei quadrati dei primi n numeri naturali che siano il cubo di un numero naturale, con n numero intero inserito dall'utente.

Ad esempio per n=5, calcolare $1 + \frac{1}{8^2} + \frac{1}{27^2} + \frac{1}{64^2} + \frac{1}{125^2}$

Soluzione

```c
#include <stdio.h>

int main(){
  int i,n=10;
  double risultato = 0.0;

  printf("Inserire il numero di termini: ");
  scanf("%d", &n);

  for (i=1 ; i<=n ; i++) {
    risultato += 1.0 / ((double)(i*i*i)*(i*i*i));
  }

  printf("Risultato: %.15f\n",risultato );
  return 0;
}
```

Output

```
Inserire il numero di termini: 5
Risultato: 1.017304882737483
```

1.8 Esercizio: misurazione tempi

Misurare il tempo di esecuzione delle soluzioni relative agli esercizi 4,5,6. Dovrebbe essere comparabile nei tre casi. Nel caso fosse significativamente maggiore per il terzo caso, potete risolvere il terzo esercizio (e probabilmente anche il secondo) in modo più efficiente. Ripetere la misurazione più volte per accertarsi che le differenze non siano dovute all'effetto di altri programmi in esecuzione.

Come misurare i tempi d'esecuzione

Per misurare il tempo di esecuzione è possibile utilizzare la funzione clock() che misura il tempo in clock ticks. I ticks possono essere convertiti in secondi dividendone il valore per CLOCKS_PER_SEC.

```c
#include <stdio.h>      /* printf */
#include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */

int main (){
  clock_t t,diff;
  t = clock();
/****************************************
inserire qui il codice per il quale si vuole
misurare il tempo di esecuzione
****************************************/
  printf("Tempo di esecuzione: %f secondi\n",(clock()-t)/(double)CLOCKS_PER_SEC);
  return 0;
}
```

L'accuratezza dipende dal sistema operativo utilizzato e dall'hardware. Ad esempio il codice seguente produce un output diverso se compilato ed eseguito in Windows e Linux:

```c
#include <stdio.h>      /* printf */
#include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */

int main (){
  int conta_diff = 0;
  clock_t t,nt, somma_diff = 0;

  /* calcolo accuratezza media misurazione tempo di esecuzione */
  t = clock();
  while (t<1000000) {
    nt = clock();
    /* se il numero di tick è cambiato, allora aggiorna la media */
    if (t!=nt) {
       somma_diff += (nt-t);
       conta_diff++;
       t = nt;
    }
  }
  printf("Accuratezza media: %f (%.0f tick)\n",somma_diff*1.0/conta_diff/CLOCKS_PER_SEC,somma_diff
    *(1.0)/conta_diff);
  return 0;
}
```

Output (Ubuntu in Windows con WSL)

```
Accuratezza media: 0.015625 (15625 tick)
```

Tempi di esecuzione per una soluzione efficiente

```c
#include <stdio.h>      /* printf */
#include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */

int main (){
  int i,n=10;
  double risultato;
  clock_t t;
```

```
8
9   printf("Inserire il numero di termini: ");
10  scanf("%d", &n);
11
12  /* somma primi n termini */
13  risultato = 0.0;
14  t = clock();
15  for (i=1 ; i<=n ; i++) {
16    risultato += 1.0 / ((double)i*i);
17  }
18  printf("Primi n termini: %f secondi\n",(clock()-t)/(double)CLOCKS_PER_SEC);
19
20  /* somma primi n termini in posizione dispari*/
21  risultato = 0.0;
22  t = clock();
23  for (i=1 ; i<=n ; i++) {
24    /* i-esimo numero dispari  = i*2-1 */
25    risultato += 1.0 / ((i*2.0-1)*(i*2.0-1));
26  }
27  printf("Primi n termini in posizione dispari: %f secondi\n",(clock()-t)/(double)CLOCKS_PER_SEC);
28
29  /* somma primi n termini in posizione che sia il cubo di un intero*/
30  risultato = 0.0;
31  t = clock();
32  for (i=1 ; i<=n ; i++) {
33    risultato += 1.0 / ((double)(i*i*i)*(i*i*i));
34  }
35  printf("Primi n termini in posizione che sia il cubo di un intero: %f secondi\n",(clock()-t)/(
      double)CLOCKS_PER_SEC);
36
37  return 0;
38 }
```

Output: 1000000000 (1000M) termini

```
Inserire il numero di termini: 1000000000
Tempo di esecuzione somma primi n termini: 0.234375 secondi
Primi n termini in posizione dispari: 0.234375 secondi
Primi n termini in posizione che sia il cubo di un intero: 0.218750 secondi
```

Tempi di esecuzione per una soluzione non efficiente

```
1  #include <stdio.h>      /* printf */
2  #include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */
3  #include <math.h>       /* pow per la radice cubica */
4
5  int main (){
6    int i,j,k,n=10;
7    double risultato,radice_cubica;
8    clock_t t;
9
10   printf("Inserire il numero di termini: ");
11   scanf("%d", &n);
12
13   /* somma primi n termini */
14   risultato = 0.0;
15   t = clock();
16   for (i=1 ; i<=n ; i++) {
17     risultato += 1.0 / ((double)i*i);
18   }
19   printf("Primi n termini: %f secondi\n",(clock()-t)/(double)CLOCKS_PER_SEC);
20
21   /* Primi n termini in posizione dispari, esaminando tutte le posizioni */
22   risultato = 0.0;
23   t = clock();
24   i=1;  /* indice */
25   k=0;  /* numero di termini sommati */
26   while ( k<n ) {
```

```
27      /* se i e' dispari*/
28      if (i % 2 == 1)  {
29          risultato += 1.0 / ((double)i*i);
30          k++;
31      }
32      i++;
33    }
34    printf("Primi n termini in posizione dispari (if): %f secondi\n",(clock()-t)/(double)
        CLOCKS_PER_SEC);
35
36    /* somma primi n termini in posizione che sia il cubo di un intero (verifica cubo con radice
        cubica)*/
37    risultato = 0.0;
38    t = clock();
39    i=1;  /* indice */
40    k=0;  /* numero di termini sommati */
41    while ( k<n ) {
42       radice_cubica = pow (i, 1.0/3.0);
43       if (fabs(radice_cubica - floor(radice_cubica+0.5)) < 1E-10) {
44          risultato += 1.0 / ((double)i*i);
45          k++;
46      }
47      i++;
48    }
49    printf("Primi n termini in posizione che sia il cubo di un intero (radice cubica): %f secondi\n"
        ,(clock()-t)/(double)CLOCKS_PER_SEC);
50
51    /* somma primi n termini in posizione che sia il cubo di un intero (verifica cubo con ciclo for)
        */
52    risultato = 0.0;
53    t = clock();
54    i=1;  /* indice termine corrente */
55    k=0;  /* numero di termini sommati */
56    while ( k<n ) {
57       j=1;
58       /* verifichiamo se esiste un intero j<i tale che i ne sia il cubo */
59       while (j<i && j*j*j != i)
60          j++;
61       if (j*j*j == i) {
62          risultato += 1.0 / ((double)i*i);
63          k++;
64       }
65       i++;
66    }
67    printf("Primi n termini in posizione che sia il cubo di un intero (for nidificato): %f secondi\n
        ",(clock()-t)/(double)CLOCKS_PER_SEC);
68
69    return 0;
70  }
```

Output: 100 termini

```
Inserire il numero di termini: 100
Primi n termini: 0.000000 secondi
Primi n termini in posizione dispari (if): 0.000000 secondi
Primi n termini in posizione che sia il cubo di un intero (radice cubica): 0.046875 secondi
Primi n termini in posizione che sia il cubo di un intero (for nidificato): 44.109375 secondi
```

Output: 1000 termini

```
Inserire il numero di termini: 1000
Primi n termini: 0.000000 secondi
Primi n termini in posizione dispari (if): 0.000000 secondi
Primi n termini in posizione che sia il cubo di un intero (radice cubica): 43.515625 secondi
---- esecuzione interrotta dopo 600 secondi ----
```

1.9 Come verificare se due numeri floating point sono uguali

Le soluzione non efficienti all'esercizio 1.7, utilizzate per la misurazione dei tempi di esecuzione, contengono alcuni spunti di discussione utili per gli esercizi futuri. Ad esempio, per verificare se `i` sia un cubo perfetto al posto dell'espressione

```
radice_cubica == floor(radice_cubica)
```

si è preferito utilizzare l'espressione

```
fabs(radice_cubica - floor(radice_cubica+0.5)) < 1E-10
```

Si verifica quindi che la differenza tra `radica_cubica` e l'intero più vicino (`floor(radice_cubica+0.5)`) sia inferiore ad una soglia per non essere influenzati dall'approssimazione nel calcolo della radice cubica. Un numero avente radice cubica 3.9999999999996 è quindi considerato, ai fini dell'esercizio, un cubo perfetto.