Informatica per Statistica
Riassunto della lezione del 17/10/2012

Igor Melatti

Altri costrutti del linguaggio C: dichiarazioni di array e istruzioni

- Si consideri il programma C di Figura 1
  - è un'implementazione in C dell'algoritmo *Insertion Sort* in Figura 2
    - differentemente da come presentato in precedenza, l'algoritmo *Insertion Sort* in Figura 2 ha anche il `return`, a sottolineare maggiormente quale sia l'output
    - non è strettamente necessario, perché comunque il problema dell'ordinamento richiede di riordinare direttamente l'input, quindi ci si aspetta che l'output sia l'input A opportunamente modificato
  - ripasso: in Figura 1 sono presenti due *definizioni di funzioni*, quelle della funzione `main` e della funzione `insertion_sort`
  - la funzione `insertion_sort` ha l'intestazione alla riga 3 e il corpo dalla riga 4 alla riga 18
    - in particolare ha due parametri: il primo, `A`, è una *sequenza* o *array* di `double` (si veda più avanti per il caso generale)
    - lo si capisce che lo sia dal fatto che c'è l'asterisco (vedere più avanti)
    - il secondo è un intero non negativo
  - all'interno del corpo della funzione `insertion_sort` (righe 4-18):
    - la parte di dichiarazione variabili (righe 5-7) contiene tre variabili: una intera non-negativa (`j`), una intera (`i`) e una razionale (`key`)
      - `key` è di tipo `double` perché deve essere dello stesso tipo base dell'array (non a caso c'è la riga 10)
      - `j` è di tipo `unsigned` perché non diventa mai negativa
      - `i` è di tipo `int` perché potrebbe diventare negativa

```c
#include <stdio.h>

void insertion_sort(double *A, unsigned length_of_A)
{
  unsigned j;
  int i;
  double key;

  for (j = 1; j < length_of_A; j++) {
    key = A[j];
    i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
  }
}

#define LENGTH_OF_A 5

int main()
{
  double A[LENGTH_OF_A] = {0.1, 1.2, 3e-1, 3.4, -1e10};

  printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
  insertion_sort(A, LENGTH_OF_A);
  printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
  return 0;
}
```

Figure 1: Implementazione in C dell'*Insertion Sort*

```
1  Insertion-Sort(A)
2    for j ← 2 to length(A) do
3      key ← A[j]
4      /* Inserimento di A[j] nella sequenza ordinata A[1...j-1] */
5      i ← j-1
6      while i > 0 and A[i] > key do
7        A[i+1] ← A[i]
8        i ← i-1
9      A[i+1] ← key
10   return A
```

Figure 2: Algoritmo *Insertion Sort*

    * la parte di istruzioni descrive l'algoritmo *Insertion Sort*, ci si ritornerà più avanti
    * l'output della funzione `insertion_sort`, da un punto di vista strettamente sintattico, non c'è
        · la funzione, infatti, ha come tipo di ritorno `void`, che vuol dire "vuoto"
    * concettualmente, l'output della funzione `insertion_sort` è l'array `A`, esattamente come accadeva all'algoritmo di riferimento *Insertion Sort*
    * le funzioni di questo tipo, ovvero che ritornano un `void`, sono dette anche *procedure*
        · sono usate per provocare i cosiddetti *effetti collaterali*
        · esempi tipici: per modificare direttamente i parametri in input (come accade in Figura 1 con l'array `A`)
        · o per scrivere qualcosa su schermo con delle `printf`
        · o entrambe le cose
        · si vedrà meglio nelle lezioni a seguire
- il `main` ha l'intestazione alla riga 22 e il corpo dalla riga 23 alla riga 30
- all'interno del corpo del `main` (righe 23-30):
    * la parte di dichiarazione variabili contiene una sola variabile `A` (riga 24), dichiarata (si veda più avanti per il caso generale) come un array di esattamente 5 `double`
        · anziché scrivere proprio 5, alla riga 24 si è preferito usare una costante predefinita alla riga 20, ovvero `LENGTH_OF_A`
        · essendo una costante, `LENGTH_OF_A` non può cambiare valore (quindi ad esempio non può essere messa alla sinistra di un assegnamento)
        · scrivere la costante in questo modo (*simbolicamente*) aumenta la leggibilità del programma, e soprattutto la sua manutenibilità
        · se poi si vorrà cambiare il valore di `LENGTH_OF_A`, basta sostituire il 5 col valore voluto in riga 20 e ricompilare
        · se fosse stato scritto 5 direttamente (vedere Figura 3), ci si sarebbe dovuti ricordare di cambiarlo anche alla riga 6
    * la parte di istruzioni contiene 3 chiamate di funzione, due delle quali, identiche, chiamano la funzione di libreria `printf` (righe 26 e 28)
        · `%le` stampa un double in notazione mantissa esponente (ad esempio il numero 23.4×10^5 viene stampato `2.34e+06`)
        · `%lf` stampa un double in virgola fissa (ad esempio il numero 23.4×10^5 viene stampato `2340000`)

```
1 int main()
2 {
3   double A[5] = {0.1, 1.2, 3e-1, 3.4, -1e10};
4
5   printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
6   insertion_sort(A, 5);
7   printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
8   return 0;
9 }
```

Figure 3: Particolare della Figura 1 scritto senza `define`

    - la terza chiama la funzione `insertion_sort` (riga 27), passandogli come parametri un array ed un intero non-negativo, esattamente come da dichiarazione della funzione stessa

- Si consideri il caso generale per la dichiarazione di *array* o *vettori*
  - implementazione delle *sequenze* di cui si parla negli algoritmi (vedere lezione 6)
  - nel `main`, in questo caso, occorrevano 5 variabili di tipo double
  - anziché dichiarare davvero 5 diverse variabili, se ne dichiara una sola che è la sequenza di 5 variabili
  - molto comodo se occorre dichiarare molte variabili dello stesso tipo e concettualmente con lo stesso significato (come le sequenze degli algoritmi)
  - inoltre, dichiarandole in questo modo si possono usare indici per accedere a precise posizioni all'interno dell'array
    - se la dichiarazione fosse stata, ad esempio, `double A, B, C, D, E;`, non ci sarebbe stato modo di usare una variabile `i` e dire ad esempio che, quando `i` = 3, occorre considerare `C`
    - invece, con gli array posso scrivere `A[i]`...
  - inoltre, se si dovessero dichiarare solo 5 o 10 variabili, le si potrebbe anche scrivere esplicitamente; ma se fossero 1000 o più, diventerebbe complicato per il programmatore
  - si vedranno due modi alternativi di dichiarare array
  - il primo verrà usato nei parametri delle funzioni che hanno come argomento un array
    - in questo caso, la dimensione, ovvero il numero di valori dell'array, non è definito insieme all'array stesso
    - e infatti, ci sarà praticamente sempre un ulteriore parametro che dica qual è la lunghezza dell'array

    - esattamente come succede nella funzione `insertion_sort` di Figura 1 (riga 3)

- il secondo verrà usato per dichiarare un array all'interno delle sezione delle dichiarazioni di una funzione
    - in questo caso, la dimensione dell'array viene definita insieme all'array stesso (riga 24 di Figura 1)
    - l'array può essere anche *inizializzato* (la parte dopo l'uguale di riga 24)

- caso generale per il primo caso (dimensione ignota a priori, parametro di funzione)

  `<tipo> *<nome_array>`
    - per cosa possa essere `<tipo>`, vedere la lezione 5
    - `<nome_array>` dev'essere un *identificatore* (vedere la lezione 5)

- caso generale per il secondo caso (dimensione nota a priori)

  `<tipo> <nome_array>[<costante_naturale>];`
    - `<costante_naturale>` dev'essere un numero in $\mathbb{N}$ (ma attenzione a metterlo troppo grande...)

- in entrambi i casi, quando (all'interno della sezione delle istruzioni) si fa riferimento ad un particolare valore all'interno della sequenza, si usano le parentesi quadre
    - esattamente come nello pseudocodice adottato per gli algoritmi (vedere lezione 6)
    - ma con un'importante differenza: in C, il primo elemento di un array `A` è `A[0]`, mentre nello pseudocodice qui adottato è `A[1]`
    - questo ovviamente implica che, se un array `A` è lungo n, allora il suo ultimo elemento in C è `A[`$n-1$`]`, mentre in pseudocodice è `A[`n`]`
    - questo spiega perché nell'implementazione dell'algoritmo di Insertion Sort sono stati cambiati alcuni controlli
        - per la precisione, il ciclo `for` va da 1 a $n-1$ (c'è il minore stretto...) in Figura 1, mentre va da 2 ad n in Figura 2
        - e il ciclo while controlla che `i >= 0` anziché $i > 0$
    - esempi di uso della notazione di *dereferenziamento* con le parentesi quadre in Figura 1:
        - in riga 10, alla variabile `key` (che è di tipo `double` "singolo") viene assegnato, tra quelli all'interno dell'array `A`, il `double` che si trova all'indice attualmente indicato dalla variabile `j`
        - in riga 13, il valore che all'interno dell'array `A` si trova all'indice attualmente indicato dalla variabile `i` aumentato di 1, viene sostituito con il valore che si trova all'indice attualmente indicato dalla variabile `i`

· quindi, se `i` vale 3, l'effetto è che il valore attualmente dentro `A[4]` viene cancellato e al suo posto vi viene copiato `A[3]` (ovvero l'elemento immediatamente precedente)

- La sintassi del `for` è la seguente:

  `for (<inizializzazione>; <condizione>; <iterazione>)`
  `<blocco_istruzioni>;`

  – come nelle altre istruzioni di controllo di flusso, se <blocco_istruzioni> è costituito da più di una istruzione, allora va racchiuso tra parentesi graffe; altrimenti, le parentesi graffe sono opzionali

  – <condizione> dev'essere un'espressione valutabile come un valore *booleano* (vero o falso; vedere l'`if` alla lezione 5)

  – <inizializzazione> e <iterazione> devono essere delle istruzioni; per ora si supporrà che siano delle istruzioni singole (e non sequenze di istruzioni)

  – per quanto riguarda la *semantica* dell'istruzione `for`, vale quanto segue:

  – **l'effetto è quello di eseguire una sola volta, all'inizio dell'esecuzione del ciclo, <inizializzazione>; poi si controlla se <condizione> è vera; se così è, viene eseguito <blocco_istruzioni>; dopodiché si esegue <iterazione>; infine si controlla nuovamente se <condizione> è vera e così via (<blocco_istruzioni>, poi <iterazione>, poi <condizione>...); se in qualsiasi momento (anche all'inizio) <condizione> è falsa, l'istruzione `for` termina e si passa alla successiva**

    * quindi, se si chiamano <inizializzazione> con In, <condizione> con C, <iterazione> con It e <blocco_istruzioni> con B, la sequenza di esecuzione è $In; C; B, It; C; B, It; C; B, It; C; B, It; \ldots; C$, finché l'ultimo C è falso

  – notare che mettere il punto e virgola dopo la parentesi chiusa non è sintatticamente sbagliato, ma porta a risultati inaspettati

    * il punto e virgola verrebbe interpretato come l'istruzione vuota, quindi si procederebbe con l'alternanza <iterazione>-<condizione> finché <condizione> non diventa falsa, e solo a quel punto si prosegue con <blocco_istruzioni>
    * quindi la sequenza sarebbe $In; C; It; C; It; C; It; C; It; \ldots; C; B$, dove l'ultimo C è falso

  – l'uso che del `for` verrà (almeno per ora) fatto sarà del tipo `for (i = da; i < a; i++)` `<blocco_istruzioni>` che vuol dire: *esegui*

<*blocco_istruzioni*> *per* `a - da` *volte, con* `i` *che di volta in volta vale* $\texttt{da}, \texttt{da}+1, \texttt{da}+2, \ldots, \texttt{a}-1$

  * all'uscita dal `for`, `i` varrà `a` (`a` < `a` è falso...)
  * ma <blocco_istruzioni> non verrà eseguito con `i` = `a` (sempre perché `a` < `a` è falso)
  * inoltre si assumerà che <blocco_istruzioni> non *modifichi* `i`, ma si limiti ad usarlo
  * quindi `i` non sarà mai nella parte sinistra di un assegnamento all'interno di <blocco_istruzioni>

- La sintassi del `while` è la seguente:

  `while (<condizione>) <blocco_istruzioni>;`

  – di nuovo, se <blocco_istruzioni> è costituito da più di una istruzione, allora va racchiuso tra parentesi graffe; altrimenti, le parentesi graffe sono opzionali

  – <condizione> dev'essere un'espressione valutabile come un valore *booleano* (vero o falso; vedere l'`if` alla lezione 5)

  – per quanto riguarda la *semantica* dell'istruzione `while`, vale quanto segue:

  – **l'effetto è quello di controllare se <condizione> è vera; se così è, viene eseguito <blocco_istruzioni>; dopodiché si controlla nuovamente se <condizione> è vera e così via (<blocco_istruzioni>, poi <condizione>, poi <blocco_istruzioni>, ...); se in qualsiasi momento (anche all'inizio) <condizione> è falsa, l'istruzione `while` termina e si passa alla successiva**

  – notare che mettere il punto e virgola dopo la parentesi chiusa non è sintatticamente sbagliato, ma porta a risultati inaspettati (sostanzialmente, il punto e virgola verrebbe interpretato come l'istruzione vuota, quindi si prosegue con <blocco_istruzioni> se <condizione> è falsa, altrimenti si resta bloccati per sempre nel `while`)

  – è possibile mostrare che `for` e `while` sono equivalenti, ovvero è sempre possibile scrivere l'uno per mezzo dell'altro

- Qualche regola di buon senso sull'*indentazione*

  – gli spazi (dove per spazi si intendono non solo gli spazi fatti con la barra spaziatrice della tastiera, ma anche le tabulazioni e le andate a capo) tra una categoria sintattica e l'altra possono essere ripetuti a piacimento

- basta che ce ne sia almeno uno, a meno che non ci sia già qualche altro separatore previsto, come una parentesi o un punto e virgola o una virgola
  - quindi scrivere `int main(){int i,j;unsigned k;}` va bene
  - scrivere `intmain(){int i,j;unsignedk;}` è sbagliato: manca lo spazio sia tra `int` e `main` che tra `unsigned` e `k`
- ad esempio, la Figura 4 è assolutamente equivalente alla Figura 1

```
1 #include <stdio.h>
2
3 void insertion_sort(double *A, unsigned length_of_A)
4 {unsigned j;int i;
5 double key;
6              for (j = 1; j < length_of_A; j++) {
7 key = A[j];i = j - 1;
8 while (i >= 0 && A[i] > key) {
9 A[i + 1] = A[i];
10 i = i - 1;}
12
13  A[i + 1] = key;
14 }
15 return;}
16 #define LENGTH_OF_A 5
17 int main
19
20 ()
21 {double A[LENGTH_OF_A] = {0.1, 1.2, 3e-1, 3.4, -1e10};
22 printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
    insertion_sort(A, LENGTH_OF_A);
23 printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
24 return 0;}
```

Figure 4: Versione mal indentata della Figura 1

- ma è assai meno leggibile
- una buona indentazione può essere la seguente
- tutti i <blocco_istruzioni> vanno scritti rientrati, con due spazi (ulteriori) all'inizio della riga
  - nella Figura 1, le dichiarazioni e le istruzioni nei due <blocco_istruzioni> che vanno dalla riga 5 alla riga 17 e dalla riga 24 alla riga 29 sono infatti scritte con due spazi all'inizio della riga
  - altri due spazi sono aggiunti dalla riga 10 alla riga 16 perché lì c'è un ulteriore blocco di istruzioni dentro il `for`; pertanto il totale di spazi ad inizio riga sale a 4

    - altri due spazi sono aggiunti dalla riga 13 alla riga 14 perché lì c'è un ulteriore blocco di istruzioni dentro il `while`; pertanto il totale di spazi ad inizio riga sale a 6
  - le parentesi graffe aperte vanno scritto sulla stessa riga del costrutto di controllo di flusso da cui dipendono
    - ad esempio alle righe 9 e 12
  - fanno eccezione le parentesi graffe che iniziano un corpo di funzione, che vanno scritte su una riga da sole
    - ad esempio alle righe 4 e 23
  - vanno sempre scritte su una riga da sole le parentesi graffe chiuse, con gli stessi spazi iniziali del corrispondente costrutto che chiudono
    - quindi ad esempio alla riga 15 la parentesi è preceduta da 4 spazi, perché 4 spazi precedevano il corrispondente `while` alla riga 12
    - quindi ad esempio alla riga 17 la parentesi è preceduta da 2 spazi, perché 2 spazi precedevano il corrispondente `for` alla riga 9
    - le parentesi che chiudono i corpi di funzione ovviamente non hanno spazi (righe 18 e 30)
  - si lascia esattamente una riga vuota tra una definizione di funzione e l'altra, o tra definizioni di funzioni e direttive al compilatore del tipo `#define` o `#include`

- Perché usare più funzioni?
  - da un punto di vista sintattico, qualsiasi programma C può essere scritto definendo solamente la funzione `main`
  - per esempio, il programma C in Figura 5 è equivalente a quello di Figura 1
  - tuttavia, messo così è meno *usabile*
  - se in seguito nasce la necessità di usare l'*Insertion Sort* in un altro programma, in cui ad esempio l'array da ordinare si chiama `B` e la sua lunghezza è in una variabile `c`, allora occorre riscrivere tutto il blocco di istruzioni che va dalla riga 14 alla riga 23
  - prestando attenzione al fatto di sostituire (a mano, ovvero lo deve fare il programmatore) `LENGTH_OF_A` con `c` e `A` con `B`
  - si tratta di un'operazione facilmente soggetta ad errori
  - con la chiamata a funzione, invece, basta scrivere `insertion_sort(B, c);` e le sostituzioni le fa automaticamente il compilatore
  - la definizione della funzione `insertion_sort` resta così com'è, non sono necessarie modifiche
  - gli errori si riducono di molto

```c
#include <stdio.h>

#define LENGTH_OF_A 5

int main()
{
  double A[LENGTH_OF_A] = {0.1, 1.2, 3e-1, 3.4, -1e10};
  unsigned j;
  int i;
  double key;
  unsigned length_of_A;

  printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
  length_of_A = LENGTH_OF_A;
  for (j = 1; j < length_of_A; j++) {
    key = A[j];
    i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
  }
  printf("%le\t%le\t%le\t%le\t%le\n", A[0], A[1], A[2], A[3], A[4]);
  return 0;
}
```

Figure 5: Versione della Figura 1 con una sola definizione di funzione

- **Esercizio:** il programma C in Figura 1 è ad input fisso: per cambiare l'input, occorre modificare il programma stesso, ricompilarlo e rieseguirlo. In molti casi, questo non è desiderabile: si vuole un programma che possa accettare input *dopo* essere stato lanciato in esecuzione, ad esempio leggendoli da tastiera. Modificare il programma in Figura 1 in modo tale che:

  1. `LENGTH_OF_A` sia molto alto (ad esempio 1000)
  2. come prima cosa, venga letta da tastiera (chiamando opportunamente `scanf`) la dimensione dell'array (deve essere minore di `LENGTH_OF_A`, altrimenti deve scrivere un messaggio d'errore e terminare il programma)
  3. dopodiché, tutti gli elementi di `A` vanno letti da tastiera (chiamando opportunamente `scanf`), usando un ciclo `for`
  4. per scrivere gli elementi di `A` su schermo, usare ancora un ciclo `for`, chiamando opportunamente `printf`