Informatica per Statistica
Riassunto della lezione del 26/10/2012

Igor Melatti ed Ivano Salvo

Visibilità (*scoping*) di funzioni, variabili e costanti in C, e uso di variabili globali

- Si considerino le funzioni C alla Figura 1

```
1  int f2(int arg1)
2  {
3    int c = arg1;
4
5    return c;
6  }
7
8  int f1(int arg1)
9  {
10   int a, b;
11
12   a = c + b*arg1;
13   return a;
14 }
```

Figure 1: Scoping di variabili (errore)

  - compilando tali funzioni, si otterrebbe un errore alla riga 12: la variabile `c` non risulta dichiarata
  - questo nonostante una variabile `c` sia stata dichiarata alla riga 3
  - ma non va bene, perché si tratta di un'altra funzione, `f2`
  - ciò che è dichiarato dentro una funzione (come `f2` nell'esempio) resta *locale* a quella funzione, e non può essere usato all'esterno di essa

- Per correggere l'errore sintattico, si può procedere come in Figura 2
  - ovvero, dichiarando `c` anche localmente a `f1`

```
1  int f2(int arg1)
2  {
3    int c = arg1;
4
5    return c;
6  }
7
8  int f1(int arg1)
9  {
10   int a, b, c;
11
12   a = c + b*arg1;
13   return a;
14 }
```

Figure 2: Scoping di variabili (correzione 1)

  - da notare che le due variabili `c` in `f1` ed `f2` sono diverse, infatti finiranno sullo stack delle chiamate in momenti e posti diversi
    - potrebbero anche avere tipi diversi
  - da notare che questo programma è sì corretto sintatticamente (la compilazione va a buon fine), ma non calcola alcunché di significativo
  - oltretutto, a riga 12 usa due variabili locali (`b` e `c`) che non sono state inizializzate
  - ovvero, `b` e `c` vengono usate senza che prima sia stato assegnato loro alcun valore
  - questo vuol dire che conterranno un valore che può essere diverso ogni volta che si chiama `f2`
    - e dipende da cosa c'era prima sullo stack delle chiamate...

- Un modo alternativo per correggere l'errore sintattico di Figura 1, è quello di Figura 3
  - ovvero, dichiarando `c` come variabile *globale*
  - tutte le funzioni, che si trovino sotto la dichiarazione di una variabile globale, possono accedere a quella variabile
  - **le variabili globali non vengono memorizzate sullo stack delle chiamate**
  - infatti, per ogni variabile globale c'è *una sola area di memoria* in RAM, indipendentemente da quale funzione è attualmente in esecuzione o da qual è l'attuale catena di chiamate

```
1  int c; /* variabile globale */
2
3  int f2(int arg1)
4  {
5    int c = arg1;
6
7    return c;
8  }
9
10 int f1(int arg1)
11 {
12   int a, b;
13
14   a = c + b*arg1;
15   return a;
16 }
```

Figure 3: Scoping di variabili (correzione 2)

    - si tratta esattamente della soluzione "lapalissiana" discussa in lezione 10: non va bene per le variabili locali e gli argomenti delle funzioni, ma va benissimo per le variabili globali
- volendo fare ancora riferimento all'analogia con il modello di Von Neumann fatto in lezione 10:
    - mentre le variabili locali e gli argomenti delle funzioni sono scritti su un nuovo foglio ad ogni nuova chiamata di funzione...
    - ...le variabili globali sono scritte in un foglio unico, da tenere a parte rispetto alla pila di fogli contenenti la catena di chiamate a funzione
- ogni volta che una funzione modifica una variabile locale (o un argomento), quella modifica è visibile solo all'interno della funzione stessa
- ogni volta che una funzione modifica una variabile globale, quella modifica è visibile a tutte le funzioni che in seguito accederanno a quella variabile
- da notare che è possibile *ridichiarare* una variabile globale, come succede alla riga 5
    - ovvero, è possibile dichiarare una variabile locale ad una funzione, e dare a questa variabile il nome di una variabile globale
    - lo stesso può essere fatto con gli argomenti di una funzione: possono chiamarsi come una variabile globale
    - di nuovo, è possibile che il tipo della variabile ridichiarata sia diverso

- in questo caso, le modifiche alla variabile ridichiarata tornano ad essere visibili solo all'interno della funzione "ridichiarante"
  - nell'esempio, l'assegnamento `c = arg1` ha effetto solo all'interno di `f2`, quindi la variabile globale `c` non subisce alcuna modifica
  - se `f1` effettuasse un assegnamento a `c`, allora tale assegnamento avrebbe effetto sulla variabile globale

- Per meglio chiarire i vari aspetti legati a possibili ridichiarazioni, si consideri la funzione C alla Figura 4

```
1  int f(int arg1)
2  {
3    int a = 2, z, b = 1;
4    { /* inizio blocco */
5      int c = 4, b = 3;
6
7      if (b > 0) { /* inizio blocco */
8        double d, c;
9
10       c = 10;
11       d = 10/c;
12     } /* fine blocco */
13   } /* fine blocco */
14   { /* inizio blocco */
15     int d, c;
16
17     c = 10;
18     d = 10/c;
19   } /* fine blocco */
20 }
```

Figure 4: Blocchi di istruzioni con dichiarazioni

- sono state finora trattate le seguenti tipologie di istruzioni C:
  - sequenza di istruzioni, istruzione vuota, assegnamento, chiamata a funzione, costrutto `if`, costrutto `if`...`else`, ciclo `for`, ciclo `while`
- qui se ne aggiunge una nuova: il *blocco di istruzioni*: laddove può esserci un'istruzione, ci può essere anche un blocco di istruzioni
- un blocco di istruzioni è costituito una sequenza di dichiarazioni e una di istruzioni da *racchiusi tra parentesi graffe*
  - esattamente come il corpo di una definizione di funzione
  - quindi, in pratica dentro il corpo di una definizione ci può essere annidato un altro corpo di funzione

  - nell'esempio, c'è un blocco da riga 4 a riga 13, e un altro da riga 14 a riga 19
    * ed è un blocco di istruzioni anche tutto il corpo della funzione, quindi da riga 2 a 20
  - sintatticamente:
    {<`dichiarazioni`> <`istruzioni`>}
  - anche i <`blocco_istruzioni`> che si trovano nei costrutti `if` o nei cicli sono dei blocchi di istruzioni, e quindi possono contenere dichiarazioni al loro interno secondo la sintassi di sopra
  - ovviamente, possono essere annidati a piacere l'uno dentro l'altro
    * ad esempio, il blocco dell'`if` da riga 7 a riga 12 si trova a sua volta dentro il blocco da riga 4 a riga 13
  - semanticamente, vuol dire creare sullo stack delle chiamate lo spazio per le nuove variabili dichiarate in <`dichiarazioni`>, e poi eseguire <`istruzioni`>, facendo attenzione ad usare le nuove variabili se ci sono ridichiarazioni
    * quindi, aprire un blocco di istruzioni ha quasi lo stesso effetto di una chiamata a funzione (a parte il fatto che non viene passato nessun parametro)
  - una volta finito di eseguire <`istruzioni`> (quindi una volta arrivati alla parentesi graffa chiusa), lo spazio creato per le nuove variabili viene tolto dallo stack
    * quindi, chiudere un blocco di istruzioni ha quasi lo stesso effetto di una `return` di una funzione (a parte il fatto che non viene ritornato nessun valore)
  - finito il blocco, tali copie non sono più necessarie
  - date queste regole, nell'esempio in esame l'esecuzione procede come illustrato nelle Figure 5–11
  - da notare che l'ultimo blocco sovrascrive i dati del secondo

- Si considerino le funzioni C alla Figura 12
  - qui si chiama `f1` (riga 3) prima di averla definita (o dichiarata)
  - sarebbe sbagliato, tuttavia i compilatori danno al più un *warning* (avvertimento)
  - ovvero, non è un errore grave, ma sarebbe meglio non farlo

- Si considerino le funzioni C alla Figura 13
  - ora va bene (nessun warning): `f1` è definita prima di essere chiamata

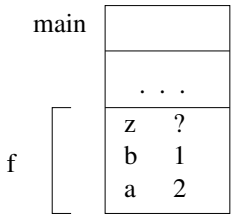


Figure 5: Situazione dello stack delle chiamate prima di riga 4 di Figura 4

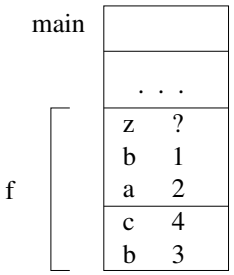


Figure 6: Situazione dello stack delle chiamate dopo riga 5 di Figura 4

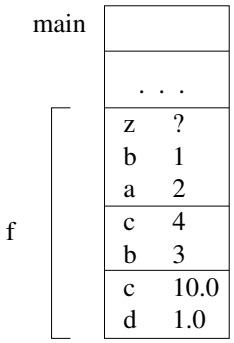


Figure 7: Situazione dello stack delle chiamate dopo riga 8 di Figura 4

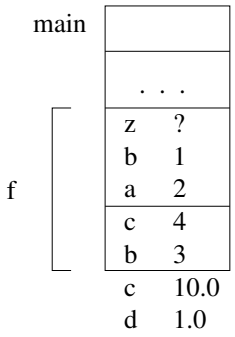


Figure 8: Situazione dello stack delle chiamate dopo riga 12 di Figura 4

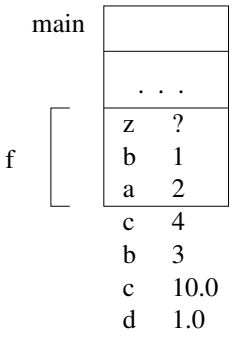


Figure 9: Situazione dello stack delle chiamate dopo riga 13 di Figura 4

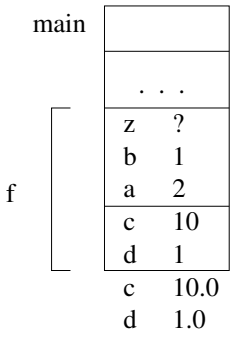


Figure 10: Situazione dello stack delle chiamate dopo riga 14 di Figura 4

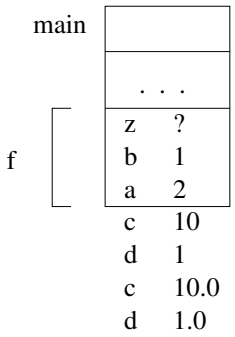


Figure 11: Situazione dello stack delle chiamate dopo riga 19 di Figura 4

```
1  int f2(int arg1)
2  {
3    int c = f1(arg1);
4
5    return c;
6  }
7
8  int f1(int arg1)
9  {
10   int a, b, c;
11
12   a = c + b*arg1;
13   return a;
14 }
```

Figure 12: Scoping di funzioni

```
1  int f1(int arg1)
2  {
3    int a, b, c;
4
5    a = c + b*arg1;
6    return a;
7  }
8
9  int f2(int arg1)
10 {
11   int c = f1(arg1);
12
13   return c;
14 }
```

Figure 13: Scoping di funzioni

- Regola generale sull'utilizzo di variabili e funzioni in C: **si può accedere (in lettura o scrittura) a qualsiasi variabile, e chiamare qualsiasi funzione, purché siano dichiarate o definite *prima* (ovvero, più in alto nel file sorgente) del punto di accesso o chiamata**
  - vale anche per le costanti definite con `#define`, solo che ovviamente possono essere usate solo in lettura

Coefficienti Binomiali

Ricordiamo la definizione di coefficiente binomiale ($n \geq 0, 0 \leq k \leq n$):

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Il loro uso è pervasivo, visto che il coefficiente binomiale $\binom{n}{k}$ è per esempio il coefficiente del monomio $a^k b^{n-k}$ nello sviluppo di $(a+b)^n$, oppure il numero dei sottoinsiemi di k elementi generati a partire da un insieme di n elementi. Prima di tutto, osserviamo che è, volendo scrivere una funzione che calcola i coefficienti binomiali, è assolutamente sconsigliato usare la definizione data sopra per scrivere il seguente programma:

```
unsigned long int cbin(int n, int k){
   return fatt(n)/(fatt(k)*fatt(n-k));
}
```

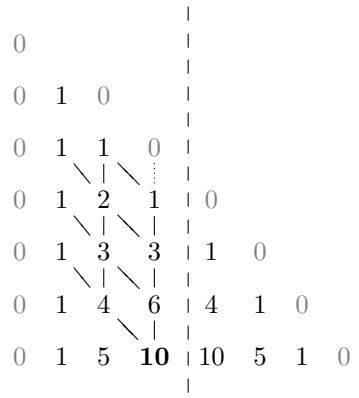
che brilla in concisione, ma ha l'enorme difetto di invocare la funzione `fatt` (che ovviamente calcola il fattoriale) su numeri potenzialmente molti grandi, a rischio di far uscire il risultato dal campo intero, anche quando il coefficiente binomiale sarebbe un numero non molto grande. Ad esempio 90! è un numero decisamente poco gestibile, ma il numero delle cinquine al Lotto $\binom{90}{5}$ è un numero decisamente ragionevole: 43.949.268 e il programma soprascritto non lo potrebbe calcolare.

Un po' meglio va osservando che $\frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times \ldots \times (n-k+1)}{(n-k)!}$. Per esercizio potete provare a utilizzare questa idea. Comunque, anche questa procedura causerebbe il calcolo di numeri più grandi del necessario, anche avendo cura di fare le divisioni non appena possibile (cioè quando la divisione da' risultato intero). Inoltre, fare somme è meglio che fare prodotti.

Di conseguenza è preferibile scrivere un programma che sfrutta la seguente notevole proprietà dei coefficienti binomiali:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \text{e} \quad \binom{n}{0} = \binom{n}{n} = 1$$

Queste relazioni portano alla costruzione del cosidetto *Triangolo di Tartaglia* (o di Newton, o di Pascal... a seconda se siete in Italia, Inghilterra o Francia ☺), che abbiamo riportato qui sotto (contorniato di zeri fantasma, su cui torneremo più tardi):



Questa relazione porta ad un immediato ed elegante programma ricorsivo:

```
unsigned long int cbin(int n, int k){
   if (n==k || n==0) return 1;
   return cbin(n,k-1)+cbin(n-1,k-1);
   }
```

che purtroppo soffre della stessa sindrome (in forma più acuta) della funzione ricorsiva che calcola i numeri di fibonacci, e va a ricalcolare più volte coefficienti binomiali già calcolati. Il modo migliore di procedere per calcolare il coefficiente binomiale $\binom{n}{k}$ è quello di generare le prime n righe del triangolo di Tartaglia.

Come "dimostrato" dalla figura, inoltre, è sufficiente generare le righe fino alla posizione k. Nella figura, si vede che il coefficiente binomiale $\binom{5}{3}$ dipende solo da coefficienti binomiali $\binom{n'}{k'}$ con $n' < 5$ e $k' \leq 2$.

A questo punto non ci resta di vedere come sia possibile generare la $n+1$-esima riga del triangolo di Tartaglia, avendo a disposizione in un vettore `t` l'n-esima. Saremmo tentati di usare un vettore di appoggio `s` e scrivere quanto segue (avendo il vettore `t` memorizzato la riga precedente (compreso lo zero fantasma)):

```
for (j=2; j<=k; j++) s[j]=t[j]+t[j-1]
```

Dovendo, ovviamente, poi ricopiare `s` in `t` per prepararsi all'iterazione successiva. Una furbizia tuttavia ci fa risparmiare parecchio: andando all'indietro questo problema non c'è. Infatti l'operazione `t[j]=t[j]+t[j-1]` distrugge il valore precedente di `t[j]`, ma se sto andando da destra a sinistra, questo valore non è necessario per calcolare i valori che stanno a sinistra di `t[j]`.

```
unsigned long int cbin(int n, int k){
   unsigned long int t[k+1];
   int i,j;

   if (k>n-k) return cbin(n, n-k);

   for (i=1; i<=k; i++) t[i]=0;
   t[0]=1;

   for (i=1; i<=n; i++)
       for (j=k; j>0; j--) t[j]=t[j]+t[j-1];

   return t[k];
}
```

Figure 14: Calcolo dei coefficienti binomiali

Un'ultima furbizia ci porta al programma in figura: ovviamente, il triangolo di Tartaglia è simmetrico, ed infatti $\binom{n}{k} = \binom{n}{n-k}$, e chiaramente, conviene applicare la procedura vista sopra per il minore tra n e $n-k$.

Un'ultima osservazione relativa alla dichiarazione dell'array `t`: tradizionalmente gli array vengono dichiarati sovradimensionati come variabili globali o al più nel main con un *numero massimo* di elementi *costante* (cioè non dipendente dall'esecuzione del programma). Il C standard tuttora non prevede dichiarazioni come quella fatta nella funzione `cbin`, ma la maggior parte dei compilatori la ammettono (compreso il `gcc`). Ci sono alcune piccole differenze con gli array "tradizionali", che dal vostro punto di vista potete ignorare: per esempio non è ammessa l'inizializzazione con la notazione a parentesi graffe {}) e vengono memorizzati in una zona di memoria diversa dalla memoria dedicata al record di attivazione della funzione, denominata *heap*. Torneremo più avanti in modo più dettagliato su cosa sia l'heap.