Informatica per Statistica
Riassunto della lezioni del 14/11/2012 e 16/11/2012

Igor Melatti

Cenni sui puntatori in C

- Per ogni variabile "normale" dichiarata, il compilatore riserva una precisa quantità di memoria (ovvero, *alloca*) in RAM
  - quanta sia questa memoria, dipende dal *tipo* della variabile stessa
  - sarà allocata sullo stack delle chiamate se è locale, al di fuori di esso se è globale
  - se la dichiarazione prevede che la variabile sia un array di dimensione n, e il tipo base richiede k bytes, allora in tutto vengono allocati nk bytes
- I puntatori, in questo corso, verranno usati per 2 motivi:
  1. per creare array dinamici, in cui non è noto a tempo di compilazione quanto valga n
     - per esempio, perché n è una variabile il cui valore viene letto da tastiera con una `scanf`
  2. per avere funzioni con parametri di output
- Si supponga quindi di voler creare un array `A` la cui lunghezza sia data da una variabile `n` letta con una `scanf`
  - questo vuol dire che, a tempo di compilazione, non si può sapere la lunghezza di `A`
  - questo perché, nel momento in cui si compila, il valore di `n` può essere qualunque
  - mentre, nei casi in cui si dichiara, ad esempio `int A[NUM_VALORI];`, con `NUM_VALORI` definito tramite una `#define`, il valore è noto: è quello scritto nella `#define` stessa
- Il C "standard" non permette di scrivere `int A[n];`, con `n` variabile

  - in realtà, `gcc` lo permette, ma come estensione fuori dallo standard

- Un *puntatore* (ad un certo tipo) in C è essenzialmente una "promessa" di memoria (di quel tipo)
- Prima o poi, questa promessa dovrà essere mantenuta, e una variabile di tipo puntatore conterrà l'*indirizzo* in RAM dove *comincia* una area
- Più di preciso, per quanto riguarda la creazione di un array dinamico `A` (i cui elementi abbiano un certo tipo), avviene quanto segue

  1. si dichiara un puntatore secondo la stessa sintassi vista nella lezione 8 per il passaggio di array come parametri di input a funzioni
     - quindi, la sintassi è `<tipo> *<nome_array>;`
     - esattamente come succedeva al primo parametro della funzione `insertion_sort`
     - esempio: `double *A;` è una dichiarazione di un vettore dinamico, chiamato `A`, i cui elementi saranno `double`
  2. nel momento in cui la dimensione dell'array risulta nota (ad esempio perché è stata letta da tastiera), si chiama la funzione `calloc` per allocare effettivamente memoria
     - per chiamare `calloc` occorre aver incluso `#include <stdlib.h>`
     - la `calloc` si aspetta 2 argomenti: il primo indica il numero di elementi dell'array, il secondo quanti bytes servono per ciascun elemento
     - per il secondo parametro si usa solitamente l'operatore `sizeof`
     - ad esempio, per allocare un array di 100 double, occorrerà scrivere `A = (double *)calloc(100, sizeof(double));`
       - sì è vero, così la dimensione è costante e non dinamica, ma niente vieta di usare questa sintassi anche per gli array a dimensione costante
       - invece, lo standard del C vieta di usare l'altra sintassi (quella in cui un array è dichiarato come `double A[100];`) per vettori dinamici
       - il `(double *)` prima della calloc è un'operazione di *cast*
       - senza entrare troppo nei dettagli, in questo corso si supporrà che il cast `(<tipo> *)` viene sempre fatto quando si chiama la `calloc` su un vettore dichiarato come `<tipo> *<nome_array>;`
     - la memoria allocata con la `calloc` non si trova sullo stack delle chiamate (neanche se il vettore è dichiarato locale alla variabile), ma in un'altra zona di memoria chiamata *heap*

3. si usa `A` come un vettore "normale", ovvero `A[`n`]` accede all'elemento $(n+1)$-esimo dell'array
   - per la precisione: `A` indica l'inizio dell'area di memoria in cui sono memorizzati consecutivamente tutti gli elementi di `A`
   - `A[`n`]` vuol dire: parti dall'indirizzo `A`, salta i primi nk bytes (con k dimensioni di ogni elemento dell'array) e prendi quello che trovi dopo
   - ad esempio, `A[0]` vuol dire: parti dall'indirizzo `A`, salta i primi 0 bytes e prendi quello che trovi dopo: non è altro che il primo elemento dell'array, come ci si aspetta
   - ad esempio, `A[10]` vuol dire: parti dall'indirizzo `A`, salta i primi $10k$ bytes e prendi quello che trovi dopo: non è altro che l'undicesimo elemento dell'array, come ci si aspetta
4. quando l'uso del vettore `A` si è concluso, ovvero non è più necessario accedere ai suoi elementi, occorre chiamare la funzione `free` per deallocare la memoria precedentemente allocata
   - anche se il vettore è dichiarato come locale ad una funzione, dato che la memoria allocata con la `calloc` non si trova sullo stack delle chiamate, allora non viene automaticamente liberata quando la funzione finisce
   - non sarebbe un errore non chiamare `free`, ma il problema è che così si usa meglio la memoria
   - infatti, chiamando la `free` l'effetto è quello di dire che la memoria precedentemente allocata per quel vettore è di nuovo libera, e quindi può essere usata per altre `calloc`
   - se non la si liberasse, le successive chiamate a `calloc` potrebbero finire col riempire l'intera memoria, quando invece, alternando opportunamente `calloc` e `free`, ce la si potrebbe fare
   - ad esempio: si supponga di avere un computer molto poco potente, con appena 4KB di RAM (e senza memoria virtuale)
   - e si supponga anche di dover eseguire un programma in cui si chiamano in sequenza 2 funzioni, le quali allocano entrambe, con una `calloc` ciascuna, un vettore di 400 `double`
   - senza `free`, la seconda chiamata fallirebbe: ci sono già $400 * 8 = 3200 \approx 3$KB di RAM allocati dopo la prima funzione, e gli altri 3KB che servirebbero per la seconda portano ad un totale di 6KB
   - se però la prima funzione chiama la `free`, allora la seconda può riusare i 3KB lasciati liberi, e quindi la seconda chiamata andrebbe a buon fine

- Precisazione: l'uso della coppia `calloc`–`free` non è sempre necessario

- è il caso di quando un vettore viene passato come parametro di funzione
- ad esempio, il primo parametro della funzione `insertion_sort` di lezione 8, `A`, non viene allocato né deallocato
- questo è dovuto al fatto, che anziché usare `calloc`, si può assegnare ad un puntatore della memoria già allocata in altro modo
- ad esempio, allocata da una `calloc` su un'altra variabile, o quella allocata in seguito ad una dichiarazione di un vettore non dinamico (vedere Figura 1)

```
1  int f1(int *A, unsigned dim_A)
2  {
3    ...
4    /* senza calloc e free */
5  }
6
7  int f2(unsigned a)
8  {
9    int *A, *B;
10   int C[30];
11
12   A = (int *)calloc(a, sizeof(int));
13   B = C;
14   B = A; /* ora B ed A fanno riferimento alla stessa area di RAM
15             (contengono lo stesso indirizzo di partenza) */
16   B[1] = 3; /* quindi ora anche A[1] vale 3! */
17   f1(A, a); /* l'A di f2 viene copiato nell'A di f1, esattamente come
18                in linea 14 */
19   free(B); /* ok, si libera anche la memoria per A */
20   free(A); /* sbagliato: non si puo' liberare la stessa zona di RAM
21               due volte */
22 }
```

Figure 1: Alcuni esempi di allocazione

- Per quanto riguarda invece le funzioni con parametri di output, il problema è il seguente
  - finora, ci si è focalizzati solo su un modo che hanno le funzioni del C per tornare un output: il tipo di ritorno
  - ce n'è anche un altro, in effetti già usato, ma senza precisarlo
  - si supponga di avere una funzione che debba ritornare, ad esempio, due numeri: il minimo ed il massimo dei valori contenuti in un array

  - con i metodi visti finora, non è chiaro capire come fare: una funzione può tornare un valore solo, quindi o il minimo o il massimo, ma non entrambi

- Si può procedere nel seguente modo

  1. se si vuole che una funzione ritorni più di un valore, allora per ognuno di questi output si aggiunge un argomento alla funzione, che sia un puntatore al tipo dell'output stesso
  2. si fa sostanzialmente finta che si tratti di un vettore di dimensione 1, quindi poi nel corpo della funzione, per mettere un valore dentro ad uno di questi output (si supponga che il nome sia `o`) si usa un assegnamento dove la parte sinistra è `o[0]`
  3. quando si effettua la chiamata, occorre usare delle variabili con davanti l'operatore `&`
     - esattamente come succede con la `scanf`
  4. una volta terminata la chiamata, nelle variabili passate per i parametri di output c'è il valore richiesto (Figura 2)

- Quindi, l'output di una funzione è dato non solo dalla `return`, ma anche dai valori messi nei parametri dichiarati come puntatori

- Si consideri l'algoritmo *DaDecimaleABinario* in Figura 3: è possibile implementarlo in diversi modi, vedere Figure 4–7

```c
void swap(int *a, int *b)
{
  int c;

  c = a[0]; /* primo elemento dell'array (fittizio) a */
  a[0] = b[0];
  b[0] = c;
}

void swap_sbagliato(int a, int b)
{
  int c;

  c = a;
  a = b;
  b = c;
  /* qui, a e b sono effettivamente scambiati... */
}

int f2()
{
  int scambia_questo = 3, con_questo = 4;

  swap_sbagliato(scambia_questo, con_questo);
  /* ... ma qui no: scambia_questo = 3, con_questo = 4; */
  swap(&scambia_questo, &con_questo);
  /* ora si': scambia_questo = 4, con_questo = 3; */
}
```

Figure 2: Funzione `swap`: scambia i valori dei suoi due parametri

```
DaDecimaleABinario(n ∈ ℕ)
  H = n.
  Finché  H ≠ 0 ripeti i seguenti passi:
    K = H mod 2, H = H div 2.
    Dai in output il bit K,
      posizionandolo a sinistra del bit precedente.
```

Figure 3: Conversione da Decimale a Binario

```c
#include <stdio.h>

void da_decimale_a_binario(unsigned long n)
{
  unsigned long H = n;

  while (H != 0) {
    unsigned K = H%2;

    H = H/2;
    printf("%u", K);
  }
  /* la printf scrive sempre verso destra, quindi... */
  printf("\nIl risultato va letto al contrario\n");
  /* questo perche' l'algoritmo diceva "a sinistra" */
}
```

Figure 4: Implementazione ingenua dell'algoritmo di Figura 3

```c
#include <stdio.h>
/* al massimo, su una macchina a 64 bits, serviranno 64 bits per
   rappresentare un unsigned long */
#define LENGTH 64

void da_decimale_a_binario_array(unsigned long n)
{
  unsigned long H = n;
  unsigned i = 0;
  int j;
  unsigned risultato[LENGTH];

  while (H != 0) {
    unsigned K = H%2;

    H = H/2;
    risultato[i] = K;
    i = i + 1;
  }
  /* ora basta stampare il vettore risultato al rovescio */
  /* l'ultimo valore per j sara' -1: per questo e' dichiarato int e non
     unsigned */
  for (j = i - 1; j >= 0; j--)
    printf("%u", risultato[j]);
  printf("\n");
}
```

Figure 5: Implementazione con array fisso dell'algoritmo di Figura 3

```c
#include <stdio.h>
#include <stdlib.h> /* per calloc e free */

void da_decimale_a_binario_array_din(unsigned long n)
{
  unsigned long H = n;
  unsigned i = 0;
  int j;
  unsigned *risultato;

  /* primo ciclo per sapere quanti bit ha il risultato */
  while (H != 0) {
    H = H/2;
    i = i + 1;
  }
  /* il risultato ha i bit, si puo' allocare */
  risultato = (unsigned *)calloc(i, sizeof(unsigned));
  /* i ed H vanno reinizializzati, il primo ciclo li ha cambiati */
  i = 0;
  H = n;
  while (H != 0) {
    unsigned K = H%2;

    H = H/2;
    risultato[i] = K;
    i = i + 1;
  }
  for (j = i - 1; j >= 0; j--)
    printf("%u", risultato[j]);
  free(risultato);
  printf("\n");
}
```

Figure 6: Implementazione con array dinamico dell'algoritmo di Figura 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void da_decimale_a_binario_array_din_1loop(unsigned long n)
{
  unsigned long H = n;
  unsigned i = 0;
  int j;
  unsigned *risultato;

  /* la formula ⌊log2(n)⌋ + 1 da' direttamente il numero di bit per n */
  risultato = (unsigned *)calloc(floor(log(n)/log(2)) + 1,
                                 sizeof(unsigned));
  while (H != 0) {
    unsigned K = H%2;

    H = H/2;
    risultato[i] = K;
    i = i + 1;
  }
  for (j = i - 1; j >= 0; j--)
    printf("%u", risultato[j]);
  free(risultato);
  printf("\n");
}
```

Figure 7: Implementazione con array dinamico, con lunghezza calcolata con una formula, dell'algoritmo di Figura 3