INFORMATICA - CdL in FISICA

COMPITO del 28/05/2003

SOLUZIONI PROPOSTE

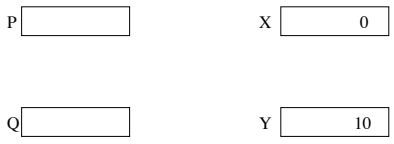
ESERCIZIO 1
Indicare i valori stampati dal seguente programma `C`.

```
#include <stdio.h>
main()
{
int *p, **q, x=0, y=10;
p = &x;
q = &p;
*q = &y;
*p = 100;
*q = &x;
**q = 200;
y = y + *p;
printf("x = %d   y= %d\m", x, y);
}
```

Soluzione Seguiamo l'evoluzione delle celle di memoria dopo ogni istruzione del programma.
Dopo la dichiarazione

`int *p, **q, x=0, y=10;`

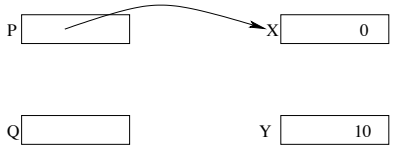
abbiamo



Dopo

`p = &x;`

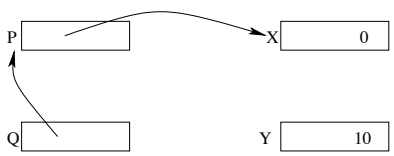
il puntatore `p` punterà ad `x`



Dopo

`q = &p;`

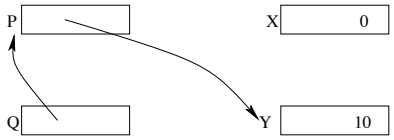
il puntatore `q` punterà a `p`



L'effetto di

```
*q = &y;
```

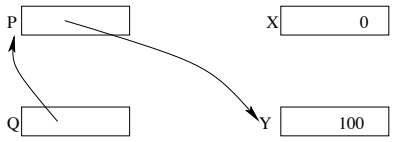
e' di far puntare `p` a `y`



L'effetto di

```
*p = 100;
```

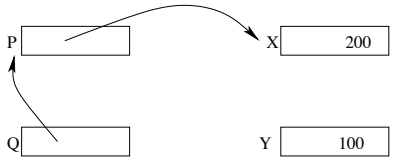
e' quello di assegnare `100` a `y`



Dopo

```
*q = &x;
```

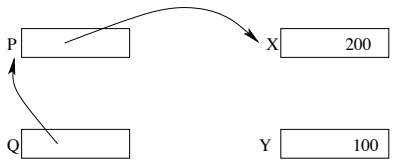
`p` punta a `x`



L'istruzione

```
**q = 200;
```

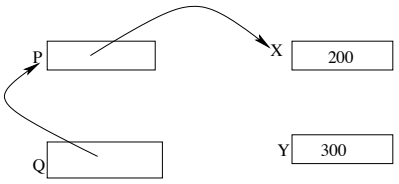
modifica la variabile `x`



Infine

```
y = y + *p;
```

ha il seguente effetto



I valori stampati sono dunque `200` e `300`.

ESERCIZIO 2
Indicare i valori stampati dal seguente programma.

```
#include <stdio.h>
main()
  {
   int x = 200;
   int y = 10;
   {
     int z;
     int y = 100;
     z = x + y;
     {
       int z=500;
       y=y+z;
     }
     x = x+y;
     y = y+1;
    }
   x = x+y;
   y = y+1;
   printf("x = %d\n", x);
   printf("y = %d\n", y);
 }
```

Soluzione Seguiamo l'evoluzione dello stack nei punti più significativi del programma.
Dopo la dichiarazione

```
int x = 200;
int y = 10;
```

x	200
y	10

Dopo le prime due dichiarazioni del primo blocco annidato

```
int z;
int y = 100;
```

z	?
y	100

x	200
y	10

Dopo l'ultima istruzione del primo blocco annidato

```
z = x + y;
```

z	300
y	100

x	200
y	10

Dopo la prima dichiarazione del secondo blocco annidato

```
int z=500;
```

z	500

z	300
y	100

x	200
y	10

Dopo l'ultima istruzione del secondo blocco annidato

```
y=y+z;
```

z	500

z	300
y	600

x	200
y	10

All'uscita del secondo blocco annidato

z	300
y	600

x	200
y	10

Eseguiamo le istruzioni

```
x = x+y;
y = y+1;
```

z	300
y	601

x	800
y	10

All'uscita del primo blocco annidato

x	800
y	10

Eseguiamo le istruzioni

```
x = x+y;
y = y+1;
```

x	810
y	11

I valori stampati sono dunque `810` e `11`.

ESERCIZIO 3
Data la seguente definizone di tipo

```
typedef enum {false, true} boolean;
```

scrivere il corpo della funzione `check` con prototipo

```
boolean check(int *vet, int dim)
```

che restituisce `true` se nel vettore `vet`, di dimensione `dim`, tutti gli elementi dopo il primo elemento dispari sono pari; restituisce `false` altrimenti.
Ad esempio, se `a` è un vettore rappresentato dalla seguente tabella

4	2	8	3	4	4	2	8	2	8

il valore restituito da `check(a, 10)` è `true`. Dato invece il vettore `b` rappresentato dalla seguente tabella

4	2	8	3	4	4	5	8	2	8

il valore restituito da `check(b, 10)` è `false`.

Soluzione Presentiamo una soluzione basata sugli schemi di programma visti (naturalmente esistono molte altre soluzioni, anche più compatte). L'idea e' quella di eseguire una prima ricerca incerta per cercare il primo elemento dispari (se esiste). Il risultato della ricerca è nella variabile `trovato1`. Successivamente applichiamo di nuovo lo schema della ricerca incerta per cercare il secondo elemento dispari (se esiste), questa volta per "controllare" che i successivi elementi siano pari.

```
boolean check(int *vet, int dim)
{ int j;
  boolean trovato1, trovato2;

  j=0;
  trovato1=false;
  while ((j<dim)&& !(trovato1))
  {
    if ((vet[j]%2)==1)
        trovato1=true;
    else
        j++;
  }
  trovato2=false;
  j++;
  while ((j<dim)&& !(trovato2))
  {
    if ((vet[j]%2)==1)
        trovato2=true;
    else
        j++;
  }
  if (!(trovato1) || (trovato1 && !(trovato2)))
        return true;
    else
        return false;
}
```

Osservazioni: Il secondo ciclo non viene eseguito se la prima ricerca ha avuto esito negativo (perché?). Inoltre la guardia booleana dell'ultimo comando condizionale può essere semplificata in `(!trovato1 || !trovato2)`.

ESERCIZIO 4
Si definisca una funzione *ricorsiva* con prototipo

```
int foo (int *vet, int from, int to)
```

dove `vet` è un array di interi, in modo che la chiamata `foo(a,f,t)` calcoli il seguente valore:

$$\texttt{foo}(\texttt{a},\texttt{f},\texttt{t}) = 2 \cdot \sum_{\texttt{i}=\texttt{f}}^{\texttt{t}-1} \texttt{a}[\texttt{i}]$$

Soluzione La funzione astratta $foo(a,f,t) = 2 \cdot \sum_{\texttt{i}=\texttt{f}}^{\texttt{t}-1} \texttt{a}[\texttt{i}]$ può essere definita induttivamente come segue

$$foo(a,f,t) = \begin{cases} 0 & \text{se } f >= t \\ 2 \cdot a[t-1] + foo(a,f,t-1) & \text{altrimenti} \end{cases}$$

Ne deriva la seguente soluzione del problema:

```
int foo (int *vet, int from, int to)
   {
   if (from >= to)
       return 0;
   else
       return ((2*(vet[to-1]))+foo(vet,from,to-1));
    }
```

ESERCIZIO 5
Supponendo date le seguenti definizioni:

```
struct El {int s; struct El *next;};
typedef struct El ElementoLista;
typedef ElementoLista *ListaDiInteri;
```

definire una procedura `InsertAfterMax` che, data una `ListaDiInteri` ℓ ed un intero *el* inserisca l'elemento *el* dopo la prima occorrenza del valore massimo presente in ℓ.
Ad esempio, data la lista rappresentata dalla seguente figura

```
--> 5 --> 2 --> 7 --> 3 --> 7
```

l'inserimento di 10 deve modificare la lista come segue:

```
--> 5 --> 2 --> 7 --> 10 --> 3 --> 7
```

Soluzione Osserviamo innanzitutto che la lista va passata per riferimento, poiché, nel caso in cui sia vuota, l'elemento va inserito come primo della nuova lista. Nel caso in cui la lista non sia vuota, bisogna determinare la posizione della prima occorrenza del massimo elemento della lista (ciò richiede necessariamente l'analisi di tutti gli elementi della lista). Una possibile implementazione è la seguente.

```
void InsertAfterMax (ListaDiInteri *lis, int el) {
    ListaDiInteri aux, pmax;
    if (*lis==NULL)
        InserisciInTesta(lis, el);
    else
    {
        pmax = *lis;
        aux = (*lis)->next;
        while (aux != NULL)
        {
            if (aux->s > pmax -> s)
                pmax = aux;
            aux = aux->next;
        }
        InserisciInTesta(&(pmax -> next), el);
    }
}
```

La procedura `InserisciInTesta` è definita come segue:

```
void InserisciInTesta (ListaDiInteri *lis,int el)
{
ListaDiInteri aux;
aux = malloc(sizeof(ElementoLista));
aux->s = el;
aux->next = *lis;
*lis = aux;
}
```