ESERCIZI COI PUNTATORI

Puntatori: Esercizio base

```
/* illustra l'utilizzo dei puntatori */
#include <stdio.h>
int main()
{
 int b,c; /* definisco 2 interi */
 int *punta; /* definisco punta come puntatore a intero */
 b=10; /* assegno a b il valore 10 */
 punta=&b; /* assegno a punta l'indirizzo di memoria di b */
 c=*punta; /* assegno a c il valore contenuto all'indirizzo di memoria che
'e specificato da punta. Quindi siccome punta contiene l'indirizzo in
memoria di b c sara' uguale a b */
 printf("punta = %d\n",punta);
 printf("b = %d\n",b);
 printf("c = %d\n",c);
 return 0;
}
```

```
punta = 2293328
b = 10
c = 10

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Array e puntatori 1

Scrivere un programma in cui il vettore v[3]=(1,2,3) venga modificato tramite l'utilizzo di puntatori in modo tale da diventare v[3]=(1,20,3).

```
#include<stdio.h>
main()

{
 int v[3]={1,2,3};
 …….;
 printf("\n%d\t%d\t%d",v[0],v[1],v[2]);
}
```

Array e puntatori 1

Scrivere un programma in cui il vettore v[3]=(1,2,3) venga modificato tramite l'utilizzo di puntatori in modo tale da diventare v[3]=(1,20,3).

```
#include<stdio.h>
main()

{
 int v[3]={1,2,3};
 *(v+1)=20;
 printf("\n%d\t%d\t%d",v[0],v[1],v[2]);
}
```

Array e puntatori 1

Scrivere un programma in cui il vettore v[3]=(1,2,3) venga modificato tramite l'utilizzo di puntatori in modo tale da diventare v[3]=(1,20,3).

```
#include<stdio.h>
main()

{
 int v[3]={1,2,3};
 *(v+1)=20;
 printf("\n%d\t%d\t%d",v[0],v[1],v[2]);
}
```

```
1       20      3
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Approfondimento array e puntatori

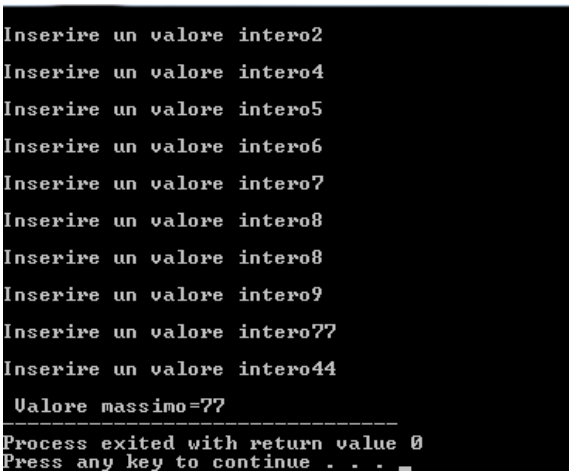
```
#include <stdio.h>
int array[5], count;
 main()
{
int *x;
 for (count=0; count < 5; count++)
 {
 printf("\nInserire un valore intero: ");
 scanf("%d", &array[count]);

 }
printf("%d",array[2] );
 x=array;
printf("\n%d",x[4] );
}
```

Se ad un puntatore assegniamo il valore dell'indirizzo del primo elemento di un array (cioè il suo nome) possiamo trattare il puntatore come se fosse l'array

```
Inserire un valore intero: 1
Inserire un valore intero: 2
Inserire un valore intero: 3
Inserire un valore intero: 4
Inserire un valore intero: 5
3
5
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Array e puntatori 2

Scrivere un programma che prenda interi da tastiera e ne calcoli il massimo, utilizzando i puntatori (in particolare: il passaggio di puntatori a funzione). Immagazzinare gli interi in un array con dimensione assegnata con una costante. L'inserimento termina inserendo uno 0




```
#include <stdio.h>
#define MAX 10
int array[MAX],count;
int largest(int *x); //dichiarazione della funzione largest
int main()
{
 /*input non piu' di MAX valori da tastiera. per terminare l'inserimento
digitare lo zero*/
 for (count=0; count < MAX; count++)
 {
 printf("\nInserire un valore intero");
 scanf("%d", &array[count]);
 if (array[count]==0)
 count=MAX;
 }

  printf("\n Valore massimo=%d", largest(array)); /*chiama la funzione */
 return 0;
}
```

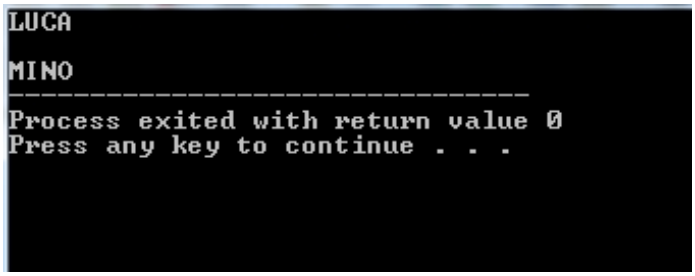
La funzione «largest»

```
int largest (int *x)  /*La funzione  restituisce il valore
massimo in un array di interi*/
{
 int count, biggest =-12000;
 for (count =0; x[count] != 0; count++)
 {
 if (x[count]>biggest)
 biggest = x[count];
 }
 return biggest;
}
```

Puntatore e array

Si scriva un programma che manipoli un array c di char tramite un puntatore c_ptr.

L'array deve avere valore iniziale ['L' 'U' 'C' 'A'] e applicando le nozioni di aritmetica dei puntatori a c_ptr si vuole trasformarlo in ['M' 'I' 'N' 'O']



```
#include <stdio.h>
int main() {
char c[4] = {'L', 'U', 'C', 'A'};
int i;
for(i = 0; i < 4; i++) {
printf("%c", c[i]);
}
printf("\n\n");
char* c_ptr = c; //equivalente a char* c_ptr = &c[0];
*c_ptr = 'M';
*(++c_ptr) = 'I';
*(c_ptr + 1) = 'N';
*(c_ptr + 2) = 'O';
for(i = 0; i < 4; i++) {
printf("%c", c[i]);
};
}
```

Ripassiamo il rapporto tra array puntatori

In C esiste un legame stretto tra array e puntatori.

Ad esempio se ptr è un puntatore a int e **a** è un array di int è possibile scrivere:

ptr = a;

invece di

ptr = &a[0];

Ma attenzione un array non è una variabile, quindi:

a = ptr e **a++** sono istruzioni NON VALIDE

Ripassiamo il rapporto tra array puntatori

a[i] può essere scritto come ***(a+i**)

cioè

&a[i] ≡ a + i

Inoltre, si può applicare l'operatore **[]** ad un puntatore, ottenendo il seguente significato

ptr[i] ≡ *(ptr+i)

```
#include <stdio.h>
int modifica_ultimo ( int a [] , int size );
int main ()
{
int a [5] = {1 ,2 ,3 ,4 ,5};
modifica_ultimo (a ,5) ;
printf ( "%d\n " ,a [4]) ; /* stampa 10 */
}
int modifica_ultimo ( int a [] , int size )
{
a[ size -1] = a[ size -1]+5;
}
```

Note: a[] denota un parametro formale che `e un array di cui si ignora la dimensione. Solitamente la dimensione dell'array si passa con un altro parametro intero (size in questo caso)

Allocazione statica: i limiti

- Per quanto sappiamo finora, in C le variabili sono sempre definite staticamente
- la loro esistenza deve essere prevista e dichiarata a priori

• Questo può rappresentare un problema soprattutto per variabili di tipo array, in cui dover specificare a priori le dimensioni (costanti) è particolarmente limitativo.

Sarebbe molto utile poter dimensionare un array “al volo”, dopo aver scoperto quanto grande deve essere

Allocazione dinamica

Per chiedere nuova memoria “al momento del bisogno” si usa una funzione di libreria che “gira” la richiesta al sistema operativo:

malloc()

- La funzione malloc():
- chiede al sistema di allocare un’area di memoria grande tanti byte quanti ne desideriamo (tutti i byte sono contigui)
- restituisce l’indirizzo dell’area di memoria allocata

La funzion «malloc»

- La funzione malloc(size_t dim):
- chiede al sistema di allocare un'area di memoria grande dim byte
- restituisce l'indirizzo dell'area di memoria allocata (NULL se, per qualche motivo,l'allocazione non è stata possibile)
- E' sempre opportuno controllare il risultato di malloc() prima di usare la memoria fornita
- Il sistema operativo preleva la memoria richiesta dall'area heap

La funzion «malloc»

- Praticamente, occorre quindi:
- specificare quanti byte si vogliono, come parametro passato a malloc()
- mettere in un puntatore il risultato fornito da malloc() stessa
- Attenzione:
- malloc() restituisce un puro indirizzo, ossia un puntatore "senza tipo"
- per assegnarlo a uno specifico puntatore occorre un cast esplicito

Esempio

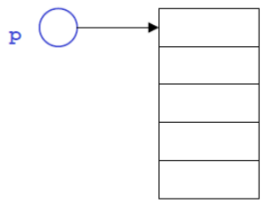
- Per allocare dinamicamente 12 byte:
- float *p;
- p = (float*) malloc(12);

- Per farsi dare lo spazio necessario per 5 interi
- (qualunque sia la rappresentazione usata per gli interi):
- int *p;
- p = (int*) malloc(5*sizeof(int));

sizeof consente di essere indipendenti dalle scelte dello specifico compilatore/sistema di elaborazione

Esempio

Allocazione:
int *p;
p = (int*) malloc(5*sizeof(int));
Risultato:

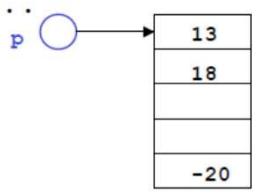


Sono cinque celle contigue, adatte a contenere un int

Aree dinamiche: uso

L'area allocata è usabile, in maniera equivalente:
- o tramite la notazione a puntatore (*p)
- o tramite la notazione ad array ([ ])

```
int *p;
p=(int*)malloc(5*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20;
```

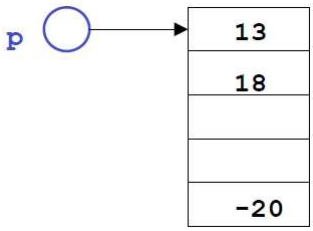


Attenzione a non "eccedere"
l'area allocata dinamicamente.
Non ci può essere alcun controllo

Aree dinamiche: uso

Abbiamo costruito un array dinamico, le cui dimensioni:
- non sono determinate a priori
- possono essere scelte dal programma in base alle esigenze del momento
- L'espressione passata a malloc() può infatti contenere variabili

```
int *p, n=5;
p=(int*)malloc(n*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20
```



Aree dinamiche: deallocazione

Quando non serve più, l'area allocata deve essere esplicitamente deallocata

– ciò segnala al sistema operativo che quell'area è da considerare nuovamente disponibile per altri usi

La deallocazione si effettua mediante la funzione di libreria **free()**

```
int *p=(int*)malloc(5*sizeof(int));
...
free(p);
```

Non è necessario specificare la dimensione del blocco da deallocare, perché il sistema la conosce già dalla malloc() precedente

Aree dinamiche: deallocazione

Il Tempo di vita di una area dati dinamica non è legato a quello delle funzioni
- in particolare, non è legato al tempo di vita della funzione che l'ha creata

Quindi, **una area dati dinamica può sopravvivere anche dopo che la funzione che l'ha creata è terminata**. Ciò consente di
- creare un'area dinamica in una funzione...
-... usarla in un'altra funzione...
-... e distruggerla in una funzione ancora diversa