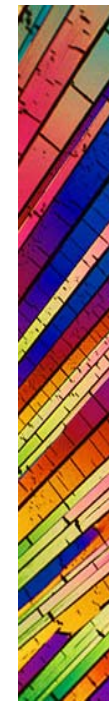


## Allocazione dinamica della memoria

Ver. 2.4

© 2010 - Claudio Fornaro - Corso di programmazione in C

2



## Allocazione della memoria

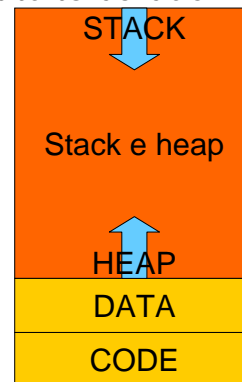
- Il termine *allocazione* viene utilizzato per indicare l'assegnazione di un blocco di memoria RAM ad un programma
- La memoria può essere allocata nello *stack*, nello *heap* o nel *Data Segment* di un programma
- La memoria può essere allocata:
  - Al run-time (dal programma in esecuzione)
  - Al compile-time (dal compilatore)
- La memoria può essere rilasciata (deallocata, resa disponibile) automaticamente o a richiesta
- Al run-time la dimensione della memoria allocata può essere fissa o modificabile

3

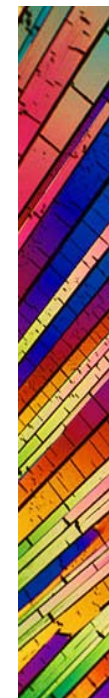


## Allocazione della memoria

- Il programma compilato è costituito da due parti distinte:
  - Code Segment
  - Data Segment
- Quando il programma viene eseguito, il Sistema Operativo alloca uno spazio di memoria per allocarvi stack e heap
- Se lo spazio per stack e heap ha dimensione fissa (dipende dal S.O.), questo può esaurirsi per effetto di ripetute chiamate a funzione (stack) non chiuse o allocazioni dinamiche (heap)



4



## Allocazione automatica

- Si ha *allocazione automatica* di un blocco di memoria quando in una funzione (*main* incluso) viene definita una variabile locale
- Il blocco (la variabile) viene creato:
  - al run-time
  - nello stack
- La dimensione non è modificabile al run-time (è nota al "compile-time", limite del C89/C90)
- Il blocco non è rilasciabile esplicitamente, ma avviene automaticamente quando termina la funzione dove è definita la variabile  
`int vett[1000];` (*interno ad una funzione*)

## Allocazione statica

- Si ha *allocazione statica* di un blocco di memoria quando viene definita una variabile di classe di allocazione statica (esterna o interna con la clausola *static*)
  - Il blocco di memoria viene allocato:
    - al compile-time
    - nel Data Segment
  - La dimensione non è modificabile al run-time (è nota al "compile-time")
  - Il blocco è non rilasciabile (non è riutilizzabile per altre allocazioni)
- `int vett[1000];` (*esterno alle funzioni*)

## Allocazione dinamica

- Si ha *allocazione dinamica* di un blocco di memoria quando si usano opportune funzioni
- Il blocco di memoria viene allocato:
  - al run-time
  - nello heap
- La dimensione è indicata alle funzioni ad ogni richiesta (più grande è il blocco richiesto, più tempo ci mette la funzione a riservarlo)
- Il blocco è rilasciabile (per poter essere utilizzato per altre allocazioni dinamiche) solo esplicitamente per mezzo di opportune funzioni (non è automatico)

## Funzioni di allocazione

- Si trovano in `<stdlib.h>`
  - Allocano un generico blocco contiguo di byte
  - Restituiscono l'indirizzo di memoria (di tipo *puntatore-a-void*) del primo byte del blocco, (NULL in caso di errore: allocazione non riuscita, controllare sempre)
  - `malloc(dim)`  
 alloca un blocco di byte non inizializzato composto da un numero di byte pari a *dim*
- ```
int *p;
p=(int *)malloc(sizeof(int));
```

## Funzioni di allocazione

- Il blocco di byte non ha di per sé alcun tipo, il cast sul puntatore restituito fa sì che il blocco di byte *sia considerato* dal compilatore come avente il tipo indicato nel cast
- Nell'esempio il cast `(int *)` fa sì che il compilatore consideri il blocco di byte come vettore di interi
- Non si può applicare l'operatore `sizeof` a un blocco di memoria allocato dinamicamente in quanto `sizeof` viene valutato dal compilatore

## Funzioni di allocazione

- Il cast *esplicito* non sarebbe necessario per un compilatore C standard, perché l'assegnazione di un puntatore `void` ad un puntatore non-`void` non lo richiede necessariamente
- Alcuni compilatori (in particolare quelli che compilano anche codice C++) lo richiedono comunque e quindi è bene aggiungerlo, anche per chiarezza e documentazione

## Funzioni di allocazione

- `calloc(numOgg, dimOgg)`  
richiede l'allocazione di un blocco di byte **inizializzati a 0** di dimensioni tali da contenere un vettore di *numOgg* elementi, ciascuno di dimensioni *dimOgg* byte, restituisce un puntatore `void` al primo byte (o `NULL`)

## Funzioni di allocazione

- `realloc(punt, dimOgg)`  
modifica la dimensione dell'oggetto puntato da *punt* portandola a *dimOgg* byte, restituisce il puntatore al *nuovo* blocco di byte
- Poiché per aumentare il blocco di memoria e mantenerne le celle contigue potrebbe essere necessario allocare un blocco nuovo e ricopiarvi i valori del blocco di partenza, il puntatore restituito può essere diverso da quello che punta al blocco di iniziale *punt*

## Funzioni di allocazione

- I contenuti del blocco sono preservati (tranne la parte che viene rimossa se la dimensione è inferiore alla precedente)
- Se la nuova dimensione è maggiore della precedente, la parte aggiuntiva non è inizializzata
- In caso di errore restituisce `NULL` e non modifica l'oggetto puntato da *punt*



## Rilascio della memoria

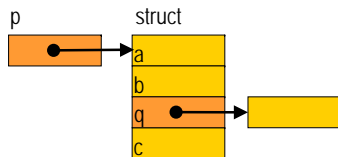
- La memoria allocata dinamicamente non viene rilasciata automaticamente (ovviamente questo capita comunque quando il programma termina in quanto *tutta* la memoria associata al programma viene rilasciata)
- Il C non ha un **garbage collector** che "recupera" al run-time la memoria inutilizzata
- Al run-time la memoria può essere rilasciata solo in modo esplicito, questo permette di riutilizzare quella porzione di memoria per servire successive allocazioni

## Rilascio della memoria

- `free(punt)`  
rilascia il blocco di memoria puntata dal puntatore *punt* (il sistema mantiene memoria del numero di byte che era stata allocato)
- Alcuni compilatori (in particolare i compilatori C++) richiedono un cast di *punt* a `(void *)`:  
`free((void *)punt);`

## Rilascio della memoria

- Se un puntatore *p* punta ad una variabile dinamica contenente un puntatore *q* ad un'altra variabile dinamica, bisogna rilasciare prima *q* e poi *p*:  
`free(p->q);`  
`free(p);`



## Esempi di allocazione Variabile scalare

- Istanziamento di una variabile scalare:  
`double *p;`  
`p=(double *)malloc(sizeof(double));`
- Utilizzo:  
`*p = 1.9;`

## Esempi di allocazione

### Vettore unidimensionale

17

- `int *p;`  
`p=(int *)malloc(sizeof(int)*100);`  
Viene allocato un blocco di byte delle dimensioni di 100 int, è l'assegnazione ad un *puntatore-a-int* (il cast è opzionale) che fa sì che il C lo "veda" come un *vettore-di-int*
- Utilizzo:  
`*p = 3;`  
`p[0] = 3;` } *equivalenti*  
`*(p+12) = 19;`  
`p[12] = 19;` } *equivalenti*
- Allocazione equivalente (ma inizializzata a 0):  
`p=(int *)calloc(100, sizeof(int));`

## Esempi di allocazione

### Vettore di strutture

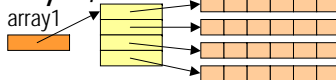
18

- `struct s {int x; int y;} *p, var;`  
`p=(struct s *)malloc(sizeof(struct s)*100);`  
oppure (equivalente):  
`p=(struct s *)malloc(sizeof(var)*100);`  
oppure (equivalente):  
`p=(struct s *)malloc(sizeof(*p)*100);`
- In tutti e tre gli esempi si alloca un blocco di memoria in grado di contenere 100 elementi (consecutivi) di tipo `struct s`, l'assegnazione ad un *puntatore-a-struct-s* che fa sì che il C lo "veda" come un *vettore-di-struct-s*  
`p[12].x = 19;`

## Esempi di allocazione

### Matrice bidimensionale

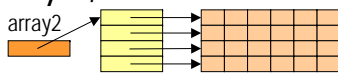
19

- Con vettore di puntatori a *Tipo*, *blocco non contiguo*:  
  
`int **array1;`  
`array1=(int **)malloc(NR*sizeof(int *));`  
`for (i=0; i<NR; i++)`  
`array1[i]=malloc(NC*sizeof(int));`
- Utilizzo:  
`array1[riga][colonna] = 12;`
- Crea un vettore di NR puntatori e per ogni puntatore alloca un vettore di int di lunghezza NC;  
NR e NC possono essere variabili

## Esempi di allocazione

### Matrice bidimensionale

20

- Con vettore di puntatori a *Tipo*, *blocco contiguo*:  
  
`int **array2;`  
`array2=(int **)malloc(NR*sizeof(int *));`  
`array2[0]=(int *)malloc(NR*NC*sizeof(int));`  
`for (i=1; i<NR; i++)`  
`array2[i]=array2[0]+i*NC;`
- Utilizzo:  
`array2[riga][colonna] = 12;`
- Crea un vettore di NR puntatori, alloca un blocco per tutta la matrice, calcola e assegna ad ogni puntatore l'indirizzo di ciascuna riga;  
NR e NC possono essere variabili

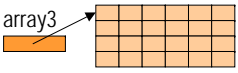
## Esempi di allocazione

### Matrice bidimensionale

21

- Con vettore di *Tipo*, simulato con calcolo esplicito:  

```
int *array3;  
array3=(int *)malloc(NR*NC*sizeof(int));
```


- Utilizzo:  

```
array3[riga*NC + colonna] = 12;
```
- Crea un blocco per tutta la matrice e accede agli elementi calcolandone la posizione (offset) riferita al primo elemento, la matrice in realtà è un vettore;  
NR e NC possono essere variabili

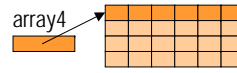
## Esempi di allocazione

### Matrice bidimensionale

22

- Con puntatore a vettori di *Tipo*, blocco contiguo:  

```
int (*array4)[NUMCOL];  
array4=(int (*)(NUMCOL))  
        malloc(NR*sizeof(*array4));
```


- Utilizzo:  

```
array4[riga][colonna] = 12;
```
- Crea un blocco per tutta la matrice e lo fa puntare da un puntatore (notare il cast); NR può essere variabile, NUMCOL è una costante
- Il tipo di array4 è "puntatore a vettore-di-NUMCOL-int", \*array4 è un "vettore di NUMCOL-int" e la sua dimensione quella di una riga della matrice (NUMCOL)

## Esempi di allocazione

### Passaggio di matrice dinamica

23

- Funzione che accetta una matrice *contigua* di dimensioni note al run-time:  

```
int f2(int *arrayp, int rows, int cols)  
{  
    ...  
    per accedere a matrice[i][j],  
    si deve usare arrayp[i*cols+j]  
}
```
- Chiamate:  

```
f2(array2[0], righe, colonne);  
f2(array3, righe, colonne);  
f2((int *)array4, righe, colonne);
```
- Nell'ultima, si può usare \*array4 che essendo un vettore-di-int decade a puntatore-a-int

## Esempi di allocazione

### Passaggio di matrice dinamica

24

- Non può funzionare con **array1** che non è stato creato come blocco contiguo di byte
- f2 accetta anche una matrice statica:  

```
int array[NR][NC];
```

In genere questo funziona correttamente poiché gli elementi sono allocati contiguamente riga per riga:  

```
f2(array[0], NR, NC);
```
- Ma f2 vede la matrice come un vettore e questo non è strettamente conforme allo standard perché l'accesso ad elementi oltre la fine della prima riga è considerato indefinito, cioè l'accesso a `(&array[0][0])[x]` non è definito per `x >= NC`





## Esempi di allocazione

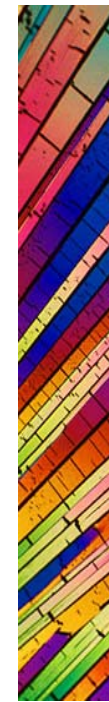
### Passaggio di matrice dinamica

25

- Funzione che accetta una matrice *anche non contigua* di dimensioni note al run-time:  

```
int f3(int **arrpp, int rows, int cols)
{
    si accede direttamente ad arrpp[i][j]
}
```
- Chiamate:  

```
f3(array1, righe, colonne);
f3(array2, righe, colonne);
```
- f3 non accetta direttamente una matrice statica come array perché, nel passaggio alla funzione, array "decade" non in un `*int`, non in un `**int`, ma in un `int (*)[NC]`



## Esempi di allocazione

### Passaggio di matrice dinamica

26

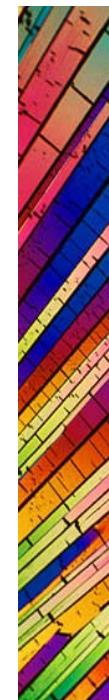
- Per passare a f3 una matrice statica è possibile creare un vettore dinamico di puntatori alle righe della matrice statica (come array2) e utilizzare questo nella funzione, il vettore di puntatori può essere definito:
  - **nel chiamante** prima della chiamata alla funzione, alla funzione viene passato il puntatore al vettore dinamico
  - **nella funzione stessa**, a cui viene passato il puntatore al primo elemento della matrice perché la funzione possa creare e inizializzare il vettore dinamico, il puntatore deve essere passato tramite un altro puntatore come richiesto dalla funzione



## Esercizi

27

1. Si scriva un programma che ordini in senso crescente i valori contenuti in un file di testo e li scriva in un'altro. Non è noto a priori quanti siano i valori contenuti nel file. Si utilizzi una funzione per l'ordinamento.  
Il programma, per allocare un vettore dinamico di dimensione appropriata, nel main:
  1. conta quanti sono i valori leggendoli dal file e scartandoli
  2. crea il vettore dinamico di dimensione adeguata
  3. lo riempie **ri**-leggendo il file
  4. lo passa alla funzione di ordinamento
  5. scrive il file di output con il contenuto del vettore riordinato.



## Esercizi

28

2. Si realizzi una funzione con prototipo identico alla f2 descritta precedentemente (salvo il tipo restituito che qui sia `void`) che riempia gli elementi della matrice con numeri progressivi e li visualizzi. In seguito si scriva un main che definisca i vettori statici e dinamici indicati precedentemente: array (5x15), array2 (20x10), array3 (20x10) e array4 (20x10) e li passi a questa funzione.

## Esercizi

3. Si scrivano due funzioni

```
int somma1(int *v,int nrow,int ncol);
```

```
int somma2(int **v,int nrow,int ncol);
```

che calcolino la somma degli elementi di una matrice di `int` passata per argomento insieme alle sue dimensioni. Si predisponga un `main` che crei una matrice statica `ms` 10x20 e una dinamica `md` 20x30 contigua come vettore di puntatori a `int` (come `array2`) e su ciascuna chiami le due funzioni.

Il passaggio della matrice statica a `somma2( )` può essere risolto con uno dei due metodi indicati.