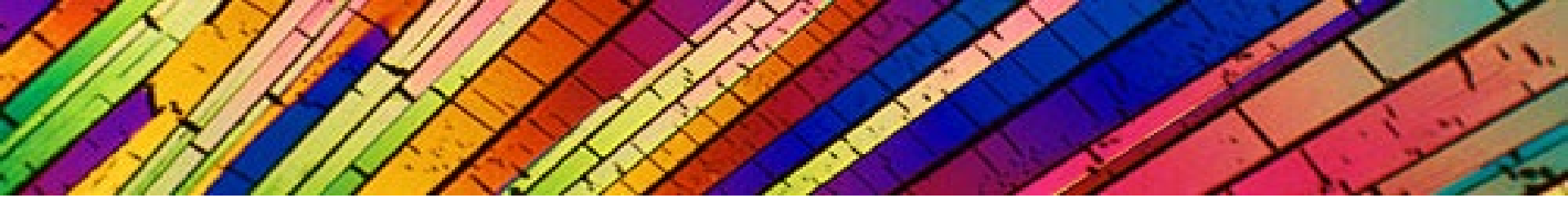
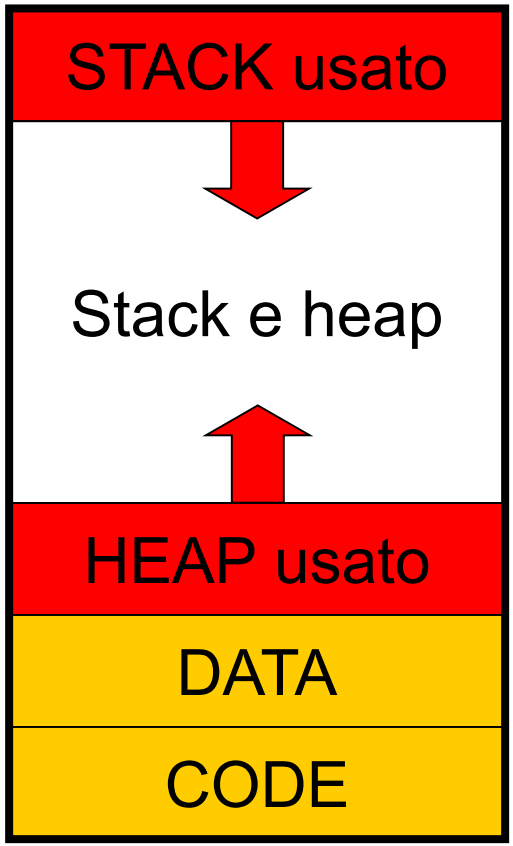


Allocazione dinamica della memoria

Ver. 3

Allocazione della memoria

- Il termine *allocazione* viene utilizzato per indicare l'assegnazione di un blocco di memoria RAM a un programma
- La memoria può essere allocata in modo:
  - statico
  - automatico
  - dinamico



Allocazione statica

- Si ha *allocazione statica* di un blocco di memoria (ad esempio per contenere una variabile) quando viene richiesta una *storage duration statica* (variabile esterna o *static*)
- Il blocco di memoria statico viene
  - *definito* al compile-time
  - *allocato* nel Data Segment prima dell'avvio del programma (load-time)
- Al run-time la dimensione non è modificabile e il blocco non è rilasciabile

  `int vett[1000];` *(esterno alle funzioni)*

Allocazione automatica

- Si ha *allocazione automatica* di un blocco di memoria quando in una funzione (anche `main`) viene definita una variabile locale con *storage duration automatica* (ossia non `static`)
- Il blocco con la variabile automatica viene:
  - *definito* o al compile-time
  - *allocato* nello stack al run-time
- Al run-time la dimensione non è modificabile e il blocco è rilasciato automaticamente quando termina la funzione dove è definita la variabile

  `int vett[1000];` *(interno ad una funzione)*

Allocazione dinamica

- Si ha *allocazione dinamica* di un blocco di memoria quando si usano opportune funzioni
- Il blocco di memoria viene
  - *definito* al run-time
  - *allocato* nello heap al run-time
- La dimensione è passata come argomento alle funzioni di allocazione (in genere più grande è il blocco richiesto, più tempo richiede allocarlo)
- Al run-time la dimensione è modificabile e il blocco è rilasciato solo esplicitamente mediante opportune funzioni (non è automatico)

Funzioni di allocazione

- Tutte le funzioni di allocazione hanno il prototipo e la definizione delle costanti quali `NULL` in `<stdlib.h>`
- Allocano un generico blocco contiguo di byte
- Restituiscono l'indirizzo di memoria del primo byte del blocco, `NULL` in caso di errore (allocazione non riuscita), controllare sempre
- La funzione `malloc` ha prototipo:
  `void *malloc(size_t` *size*`);`
  alloca un blocco di byte non inizializzato composto da *size* byte e restituisce un puntatore `void` al primo byte

Funzioni di allocazione

- Il blocco di byte non ha di per sé alcun tipo, è il cast (esplicito o implicito per assegnazione) sul puntatore `void` restituito che fa sì che il blocco di byte *sia considerato* dal compilatore come un oggetto del tipo della variabile

```
int *p;
p=(int *)malloc(sizeof(int));
```

Qui il compilatore considera il byte puntato da `p` come un valore di tipo `int`, ma non sapendo il puntatore la dimensione dell'oggetto a cui punta, è lecito che possa essere usato come un puntatore a un vettore di `int` (se lo è davvero)

Funzioni di allocazione

- Il cast *esplicito* non è necessario perché l'assegnazione di un puntatore `void` a un puntatore non-`void` è lecita e l'assegnazione attua un cast implicito
- Può essere utile aggiungerlo per chiarezza e autodocumentazione del codice
- Esempi di allocazione si trovano nelle slide successive
- Non si può usare l'operatore `sizeof` a un blocco di memoria allocato dinamicamente

Funzioni di allocazione

- La funzione `calloc` ha prototipo:
  `void *calloc(size_t` *num*`, size_t` *size*`);`
  alloca un blocco di byte **inizializzati a 0** composto da un numero di byte pari a *num*size* (ossia di dimensioni tali da contenere un vettore di *num* elementi, ciascuno di dimensione pari a *size* byte), restituisce un puntatore `void` al primo byte
- È meno efficiente della `malloc`, ma assicura l'inizializzazione a 0 e "" dello spazio

Funzioni di allocazione

- La funzione `realloc` ha prototipo:
  `void *realloc(void*` *ptr*`, size_t` *size*`);`
  ridimensiona a *size* byte l'oggetto puntato da *ptr* (che deve provenire da precedente `malloc`/`calloc`), preservandone il contenuto
- Restituisce il puntatore al *nuovo* blocco di byte o `NULL` in caso di errore
- In caso di errore non modifica l'oggetto puntato da *ptr*
- Se *ptr* è NULL equivale a una `malloc`
- Se *size* =0, dealloca la memoria

Funzioni di allocazione

- Il nuovo blocco prodotto dalla `realloc` è comunque costituito da celle contigue
- Se il blocco viene ridotto di dimensione, il puntatore restituito è lo stesso *ptr*
- Se il blocco viene ingrandito, per mantenere la contiguità di allocazione del blocco la `realloc` potrebbe dover allocare un nuovo blocco e copiarvi i valori del blocco di partenza (non inizializza i byte successivi), il puntatore restituito può quindi essere diverso da quello che punta al blocco di iniziale *ptr* (e richiedere un certo tempo di esecuzione)

Rilascio della memoria

- La memoria allocata dinamicamente deve essere rilasciata quando non è più necessaria per lasciare spazio a heap/stack
- La memoria può essere rilasciata al run-time solo in modo esplicito (viene comunque rilasciata quando il programma termina)
- L'uscita da una funzione dove è stato allocato un blocco dinamico non dealloca il blocco, dealloca il puntatore se questo è automatico
- L'accesso a memoria deallocata (con un puntatore detto in questo caso "pending") ha un comportamento indefinito

Rilascio della memoria

- La memoria non più referenziata da alcun puntatore non è più raggiungibile e quindi utilizzabile, questa situazione viene chiamata: “memory leak”
- Il C non ha un ***garbage collector*** ossia un sistema *automatico* che “recupera” al run-time la memoria non referenziata

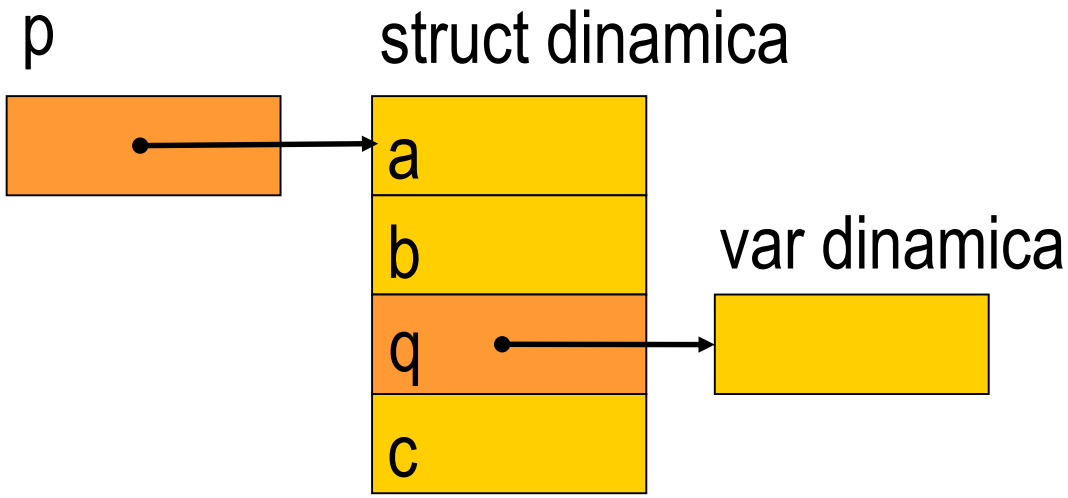
Rilascio della memoria

- La funzione `free` ha prototipo:
  `void free(void *`*ptr*`);`
  rilascia il blocco di memoria puntata dal puntatore *ptr* (il sistema mantiene memoria del numero di byte che era stata allocato)
- Il puntatore *ptr* deve derivare da allocazione dinamica, altrimenti il comportamento è indefinito, se vale `NULL` la `free` non fa nulla
- Alcuni compilatori richiedono un cast di *ptr*:
  `free((void *)`*ptr*`);`

Rilascio della memoria

- Se un puntatore `p` punta a 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

- ```
  double *p;
  p=(double *)malloc(sizeof(double));
  …
  *p = 1.9;
  ```
- ```
  struct punto {
     int x, y;
  } *pt;
  pt=(struct punto)malloc(sizeof(*pt));
  …
  pt->x = 12;
  ```

Esempi di allocazione
Vettore unidimensionale

- `int *p;`
  `p=(int *)malloc(sizeof(int)*100);`
- La `malloc` alloca un blocco di byte delle dimensioni di 100 `int` e restituisce un *puntatore-a-*`void`
- L'assegnazione del *puntatore-a-*`void` a un *puntatore-a-*`int` che fa sì che il compilatore lo consideri come un *vettore-di-*`int`
- È compito del programmatore trattare il puntatore come *vettore-di-*`int` e non come puntatore ad un unico `int` (il compilatore non può saperlo né sa le dimensioni: è al run-time)

Esempi di allocazione
Vettore unidimensionale

- 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));
```

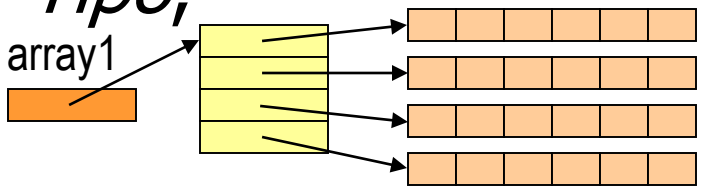
Esempi di allocazione
Vettore di strutture

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

```
p[12].x = 9;          /* stessa var */
(p+12)->x = 10;       /* stessa var */
(*(p+12)).x = 11;     /* stessa var */
```

Esempi di allocazione
Matrice bidimensionale

- 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));
```

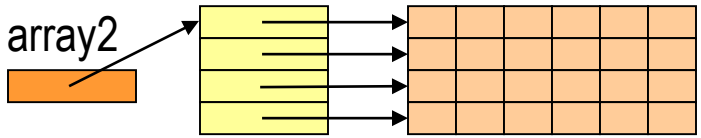
- Utilizzo:

```
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

- 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;
```

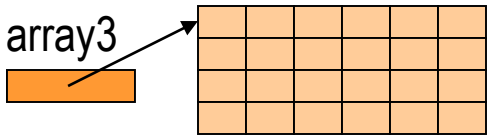
- Utilizzo:

```
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

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




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

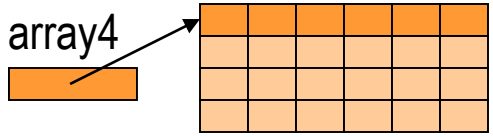
- Utilizzo:

```
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

- 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`", l'elemento `*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

- Funzione che accetta un vettore utilizzato come matrice 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);
```

Esempi di allocazione
Passaggio di matrice dinamica

- Non può funzionare con **`array1`** che non è stato creato come blocco contiguo di byte
- `f2` accetta anche una matrice non dinamica:
  `int array[NR][NC];`
  In genere questo funziona 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 a 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

- 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

- Per passare a `f3` una matrice statica: creare un *vettore dinamico* di puntatori alle righe della *matrice statica* (es. `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

Membri vettore flessibili

- Nel caso serva definire una `struct` con una dimensione non nota a priori (es. stringhe di lunghezza diversa, ma anche per altri scopi) si può utilizzare lo "struct hack", ossia si definisce una struttura così:

```
struct vstring {
   int len;
   char string[1];
};
struct vstring *str =
malloc(sizeof(struct vstring)+n-1);
str->len = n;
```

alcuni compilatori permettono anche `[0]`

Membri vettore flessibili

- Il metodo visto, non compatibile con lo Standard, usa un vettore (qui una stringa) che si sa essere presente, ma che in effetti non fa parte della `struct`

Esercizi

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

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 nrows,int ncols);
int somma2(int **v,int nrows,int ncols);
```

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.