

# Introduzione al C

## Lez. 4

Allocazione  
dinamica della memoria  
e  
stringhe

# Allocazione dinamica memoria

In C la memoria può essere anche gestita in modo **dinamico**, attraverso l'allocazione esplicita di blocchi di memoria

A cosa serve?

- Ad allocare array la cui dimensione non è nota a tempo di compilazione ma calcolata a tempo di esecuzione
- Per gestire strutture dati che crescono e decrescono durante l'esecuzione del programma (ad esempio liste o alberi)
- Per avere maggiore flessibilità sulla durata della memoria allocata. (Altrimenti la memoria verrà deallocata all'uscita del blocco nel quale è stata allocata.)

# Allocazione dinamica memoria

## Esempio

Leggere da input un intero n e creare un array di n interi

```
int n;  
scanf("%d", &n);
```

```
int a[n]; // ERRORE!
```

Non è consentito in ANSI C!

Il numero di elementi di un array deve essere noto a tempo di compilazione. Quindi, deve essere una costante e non una variabile.

# Allocazione dinamica memoria

- i blocchi sono allocati tipicamente in una parte della memoria chiamata **heap**;
- è possibile accedere a tali blocchi di memoria attraverso l'uso di **puntatori**;
- lo spazio allocato dinamicamente NON viene liberato all'uscita delle funzioni;
- sempre con l'uso di puntatori la memoria che non serve più va **deallocata** in modo da renderla nuovamente disponibile.

# Allocazione dinamica memoria

- i blocchi sono allocati tipicamente in una parte della memoria chiamata **heap**;
- è possibile accedere a tali blocchi di memoria attraverso l'uso di **puntatori**;
- lo spazio allocato dinamicamente NON viene liberato all'uscita delle funzioni;
- sempre con l'uso di puntatori la memoria che non serve più va **deallocata** in modo da renderla nuovamente disponibile.

Come si fa?

- Tre funzioni `malloc`, `calloc` e `realloc` ci permettono di allocare memoria dinamicamente.

# Malloc

Definita in `stdlib.h`. Includetelo!

```
#include <stdlib.h>
```

```
void * malloc(size_t size)
```

- dove `size` specifica la lunghezza **in byte** del blocco allocato.  
`size_t` è un tipo definito in `stdlib.h`. (In genere è un `unsigned int`.)

# Malloc

Definita in `stdlib.h`. Includetelo!

```
#include <stdlib.h>
```

```
void * malloc(size_t size)
```

- dove `size` specifica la lunghezza **in byte** del blocco allocato. `size_t` è un tipo definito in `stdlib.h`. (In genere è un `unsigned int`.)
- La funzione restituisce un puntatore al primo byte del blocco allocato. (`void` è un tipo generico). Con un assegnamento il tipo del puntatore viene convertito implicitamente.

```
int *p = malloc( 10 * sizeof(int) )
```

# Malloc

Definita in `stdlib.h`. Includetelo!

```
#include <stdlib.h>
```

```
void * malloc(size_t size)
```

- dove `size` specifica la lunghezza **in byte** del blocco allocato. `size_t` è un tipo definito in `stdlib.h`. (In genere è un `unsigned int`.)
- La funzione restituisce un puntatore al primo byte del blocco allocato. (`void` è un tipo generico). Con un assegnamento il tipo del puntatore viene convertito implicitamente.

```
int *p = malloc( 10 * sizeof(int) )
```

- Se non è possibile allocare memoria (ad es. è esaurita), la funzione restituisce il puntatore `NULL` (un altro modo per dire `0 :-)`). Controllarlo. Sempre!



# Malloc Esempio

```
#include <stdlib.h>
```

```
...
```

```
int i, n, *p;  
scanf("%d", &n);  
p = malloc(n * sizeof(int)); //alloca spazio per n interi  
  
if (p == NULL) { // controlliamo  
    printf("Non posso allocare %d interi\n",100);  
    exit(1); // esce con un errore  
}
```

# Malloc Esempio

```
#include <stdlib.h>
```

```
...
```

```
int i, n, *p;  
scanf("%d", &n);  
p = malloc(n * sizeof(int)); //alloca spazio per n interi  
  
if (p == NULL) { // controlliamo  
    printf("Non posso allocare %d interi\n",100);  
    exit(1); // esce con un errore  
}  
... // se sono arrivato qui posso usare il blocco allocato...  
  
for(i = 0; i < n; i++) // qui mettiamo i primi n interi  
    p[i] = i;
```

Attenzione! Non si può accedere ad indirizzi fuori dallo spazio allocato. p[n] **NO!** Segmentation fault...

# Calloc

Definita in `stdlib.h`. Includetelo!

```
void * calloc(size_t nmemb, size_t size)
```

- simile alla `malloc` ma gli elementi sono inizializzati a 0.
- `nmemb` indica il numero di elementi nell'array mentre `size` specifica la lunghezza in byte di ciascun elemento.

# Calloc

Definita in `stdlib.h`. Includetelo!

```
void * calloc(size_t nmemb, size_t size)
```

- simile alla `malloc` ma gli elementi sono inizializzati a 0.
- `nmemb` indica il numero di elementi nell'array mentre `size` specifica la lunghezza in byte di ciascun elemento.

Esempio

Scrivere `calloc` usando la `malloc`

# Calloc

Definita in `stdlib.h`. Includetelo!

```
void * calloc(size_t nmemb, size_t size)
```

- simile alla `malloc` ma gli elementi sono inizializzati a 0.
- `nmemb` indica il numero di elementi nell'array mentre `size` specifica la lunghezza in byte di ciascun elemento.

Esempio

Scrivere `calloc` usando la `malloc`

```
void * my_calloc(size_t nmemb, size_t size) {  
    size_t i;  
    char *p = malloc( nmemb * size );  
    if (p == NULL) return NULL  
    for (i = 0; i < nmemb*size; i++) *(p+i) = 0;  
    return p;  
}
```

# Liberare la memoria

Quando un blocco di memoria non serve più è importante deallocarlo e renderlo nuovamente disponibile utilizzando la funzione

```
void free(void *)
```

L'argomento di free deve essere allocato precedentemente (o NULL) altrimenti il comportamento è indefinito.

# Esercizio

Input: Leggere da tastiera una sequenza di interi. Il primo di questi indica quanti interi compongono la sequenza, il resto sono immessi in un array allocato dinamicamente con malloc

Output:  $m \ x$   
dove  $m$  è la media degli interi e  $x$  è il numero di interi uguali a  $m$ .

Esempio

4

7

7

4

10

7 2

# Stringhe

Una stringa è una sequenza di caratteri. Ad esempio una parola, una frase, un testo...

In C non è previsto un tipo per le stringhe.

Una stringa è vista come un array di caratteri che, per convenzione, termina con il simbolo speciale `'\0'`.

Si usa

```
char s[N+1];
```

per memorizzare una stringa di N caratteri.



# Stringhe

Le costanti stringa vengono rappresentate tra virgolette.

Esempio

“ciao” è un array di caratteri di dimensione 5.

c	i	a	o	\0
---	---	---	---	----

# Stringhe

Le costanti stringa vengono rappresentate tra virgolette.

Esempio

“ciao” è un array di caratteri di dimensione 5.

c	i	a	o	\0
---	---	---	---	----

Una costante stringa viene trattata come un puntatore al primo carattere della stringa.

Esempio

```
char *s = "ciao"; // anche char s[] = "ciao"; s qui è costante
printf("%s %s\n", s, s+1); // stampa ciascun carattere fino a '\0'
printf("%c %c\n", s[0], s[1]);
```

Cosa stampa?

# Stringhe

Le costanti stringa vengono rappresentate tra virgolette.

Esempio

“ciao” è un array di caratteri di dimensione 5.

c	i	a	o	\0
---	---	---	---	----

Una costante stringa viene trattata come un puntatore al primo carattere della stringa.

Esempio

```
char *s = "ciao"; // anche char s[] = "ciao"; s qui è costante
printf("%s %s\n", s, s+1); // stampa ciascun carattere fino a '\0'
printf("%c %c\n", s[0], s[1]);
```

Cosa stampa?

```
ciao iao
c i
```

Nota: la `stdlib` contiene molte funzioni per gestire stringhe.

# Stringhe

Esempio

```
void my_printf(char *s) {  
  
    int i = 0;  
    while(s[i]) // s[i] == 0  
        printf("%c", s[i++]);  
  
}  
  
int main() {  
  
    char s[101]; // stringhe fino a 100 caratteri  
    scanf("%s", s);  
    my_printf(s);  
    return 0;  
  
}
```

# Stringhe

Alternativa

```
void my_printf(char *s) {  
    while(*s)  
        printf("%c", *s++); // è s ad essere incrementato  
}  
  
int main() {  
    char s[101]; // stringhe fino a 100 caratteri  
    scanf("%s", s);  
    my_printf(s);  
    return 0;  
}
```

# Stringhe

Ovviamente potete allocare dinamicamente memoria per contenere una stringa.

```
char * s = malloc((N+1)*sizeof(char));  
if(!s) exit(1);
```

# Esercizio 3

Scrivere la funzione

```
int my_strlen(char *s)
```

che restituisce il numero di caratteri della stringa s.

Scrivere un programma che provi questa funzione leggendo una stringa da tastiera. Si può assumere che la stringa in input contenga non più di 1000 caratteri.

Strumenti utili:

Si può leggere leggere una stringa da tastiera con

```
char str[1001];  
scanf("%s", str);
```

# Esercizio 4

Scrivere la funzione

```
char * my_strcat(char *s1, char *s2)
```

che restituisce un puntatore alla **nuova** stringa ottenuta concatenando le stringhe puntate da `s1` e `s2`.

Scrivere un programma che legga due stringhe da tastiera e stampando la stringa e stampi la concatenazione delle due.

Si può assumere che le stringhe in input contengano non più di 1000 caratteri.

Notare che il comportamento di `my_strcat()` è diverso da quello della funzione `strcat()` presente nella libreria `stdlib`.



# Esercizio 5

Scrivere la funzione:

```
int anagramma(char *s1, char *s2) ;
```

che restituisca 1 se le stringhe puntate da s1 e s2 sono una l'anagramma dell'altro e 0 altrimenti.

Es.:

```
anagramma("pizza", "pazzi") == 1
```

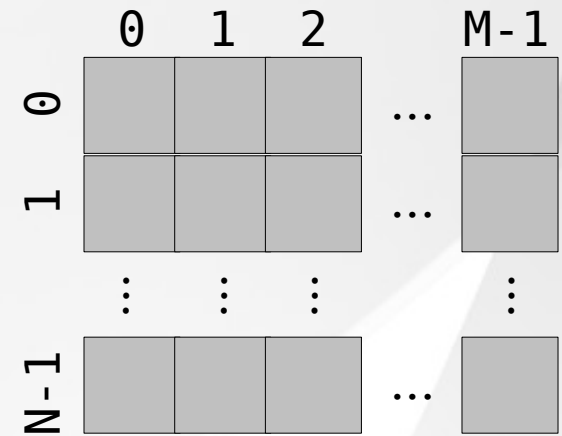
Hit. Data una stringa S, costruire un array aS[256] tale che aS[i] memorizza il numero di occorrenze del carattere i in S.

Come sono gli array aS e aZ di due stringhe S e Z che sono una l'anagramma dell'altra?

# Array Bidimensionali

Una matrice che ha N righe ed M colonne

```
int a[N][M];
```

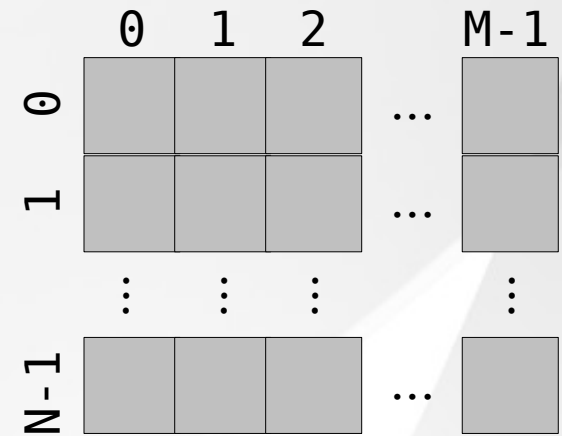


# Array Bidimensionali

Una matrice che ha N righe ed M colonne

```
int a[N][M];
```

Si può utilizzare `a[i][j]` per accedere all'elemento in riga `i` colonna `j`.



# Array Bidimensionali

Una matrice che ha N righe ed M colonne

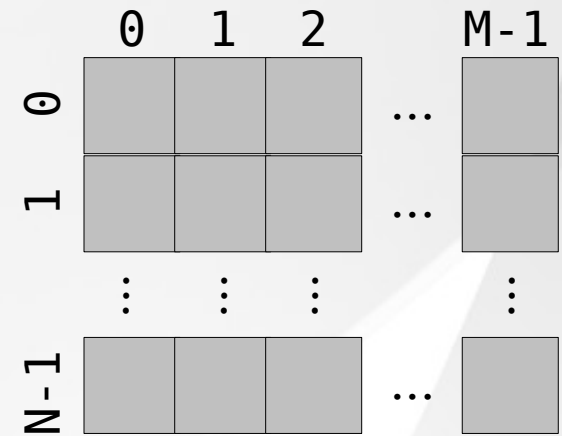
```
int a[N][M];
```

Si può utilizzare `a[i][j]` per accedere all'elemento in riga `i` colonna `j`.

Si può passare un array bidimensionale ad una funzione **ma** si deve specificare la lunghezza della seconda componente nella dichiarazione della funzione!

Esempio

```
void foo(int a[][5], int righe);
```



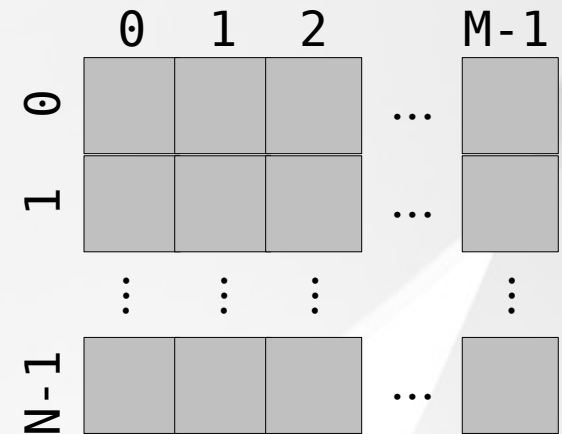
# Array Bidimensionali

Una matrice che ha N righe ed M colonne

```
int a[N][M];
```

Si può utilizzare `a[i][j]` per accedere all'elemento in riga `i` colonna `j`.

Si può passare un array bidimensionale ad una funzione **ma** si deve specificare la lunghezza della seconda componente nella dichiarazione della funzione!



Esempio

```
void foo(int a[][5], int righe);
```

Questa funzione opererà solo con matrici aventi 5 colonne. Forte limitazione!

Si possono rimpiazzare con array classici calcolando opportunamente gli indici.

# Array Bidimensionali

Esempio. Calcolo della somma degli elementi in una matrice  $N \times 5$

```
int sum(int a[][5], int righe) {  
  
    int i, j, sum = 0;  
    for( i = 0; i < righe; i++ )  
        for ( j = 0; j < 5; j++ )  
            sum += a[i][j];  
    return sum;  
  
}
```

