



# Capitolo 3: L'allocazione dinamica della memoria

PUNTATORI E STRUTTURE DATI DINAMICHE:  
ALLOCAZIONE DELLA MEMORIA E  
MODULARITÀ IN LINGUAGGIO C



# Allocazione

---

REGOLE E FUNZIONI PER ALLOCARE/DE-ALLOCARE

# Allocare = collocare in memoria

- Allocare una variabile = associarvi una porzione di memoria (in cui collocare i dati)
- L'allocazione è:
  - implicita, automatica e statica se gestita dal sistema
  - esplicita se sotto controllo del programmatore
  - dinamica:
    - avviene in fase di esecuzione
    - permette di cambiare la dimensione della struttura dati
    - permette di realizzare “contenitori” cui si aggiungono o tolgono elementi.

# Da codice sorgente a eseguibile

Le variabili sono soggette a precise regole di:

- **esistenza**
- **memoria**
- **visibilità.**

Le variabili si distinguono in:

- **globali**
- **locali.**

## Variabili **globali**:

- definite al di fuori da funzioni (main incluso)
- permanenti
- visibili dovunque nel file a partire dalla loro definizione
- definite in generale nell'intestazione del file.

## Vantaggi:

- accessibili a tutte le funzioni
- non necessario passarle come parametri
- utilizzo semplice ed efficiente

## Svantaggi:

- minore modularità, leggibilità, affidabilità

## Variabili **locali**:

- variabili definite all'interno delle funzioni (main incluso)
- parametri alle funzioni
- temporanee (iniziano ad esistere quando è chiamata la funzione e cessano quando se ne esce)
- visibili solo nella funzione in cui sono dichiarate.

## Compilatore:

- programma che esegue un'analisi
  - lessicale
  - sintattica
  - semanticadel codice sorgente
- e genera un codice oggetto (in linguaggio macchina)



Il codice oggetto contiene riferimenti a funzioni di libreria

Linker:

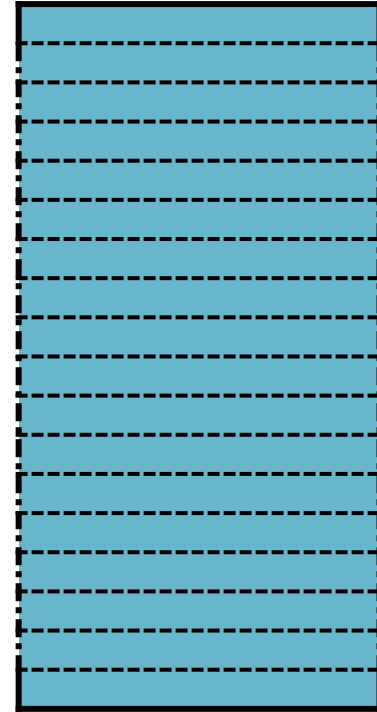
- programma che risolve:
  - i riferimenti a funzioni di libreria
  - I riferimenti reciproci tra più file oggetto.
- e genera un codice eseguibile

Loader:

- modulo del sistema operativo che carica in memoria centrale:
  - Il codice eseguibile (istruzioni)
  - I dati su cui opera

# Programma in memoria

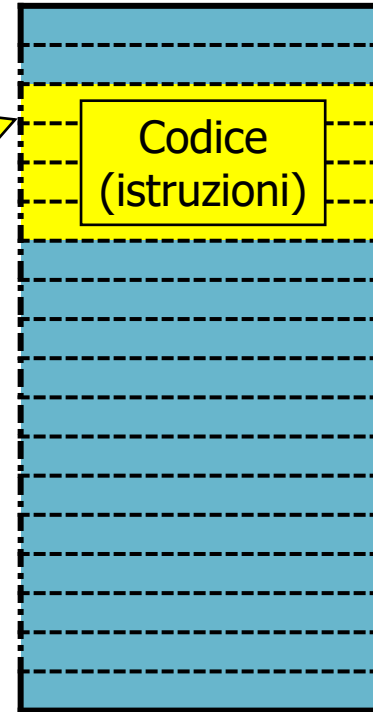
```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[],int n){
    int i, j, max;
    ...
};
```



RAM (1 GB)

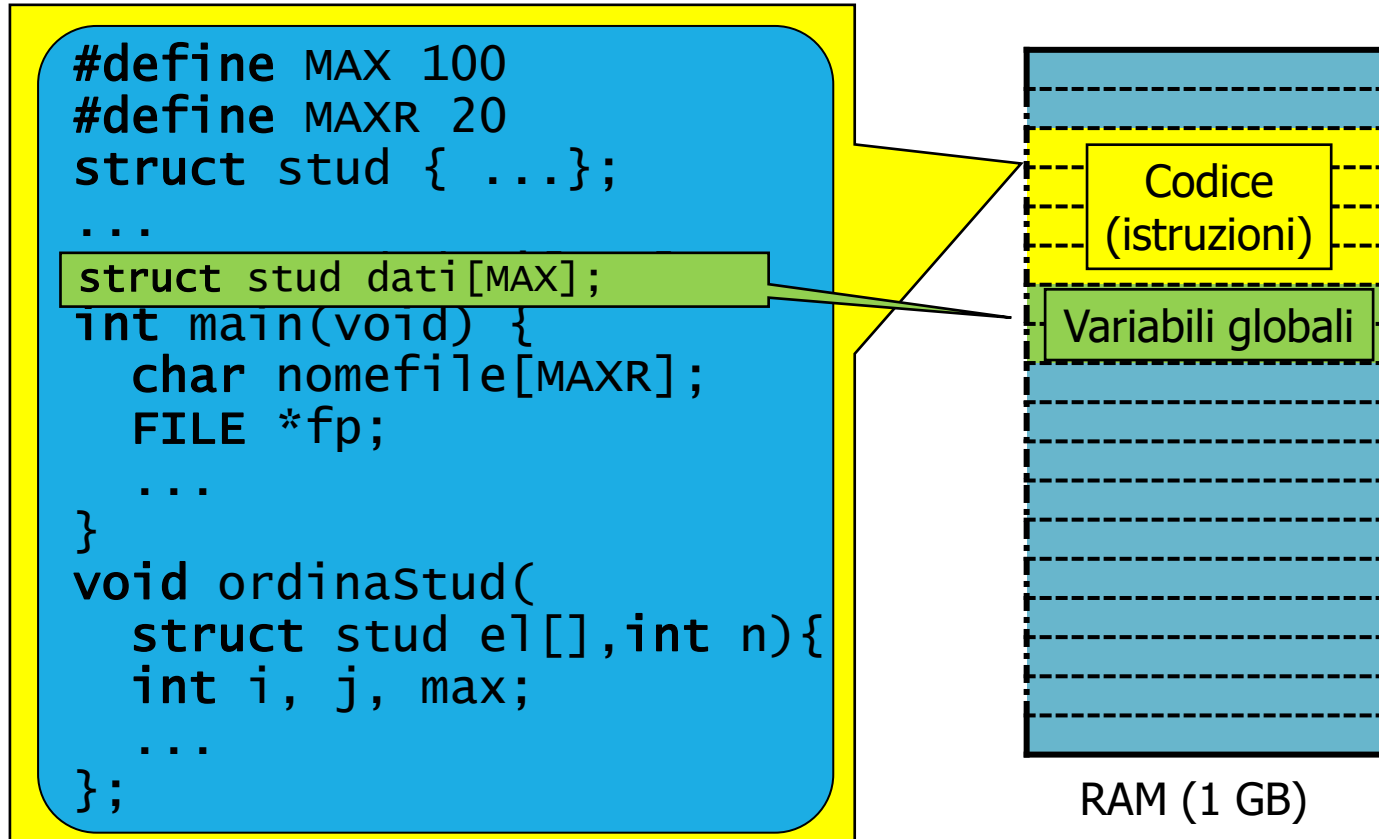
# Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[],int n){
    int i, j, max;
    ...
};
```

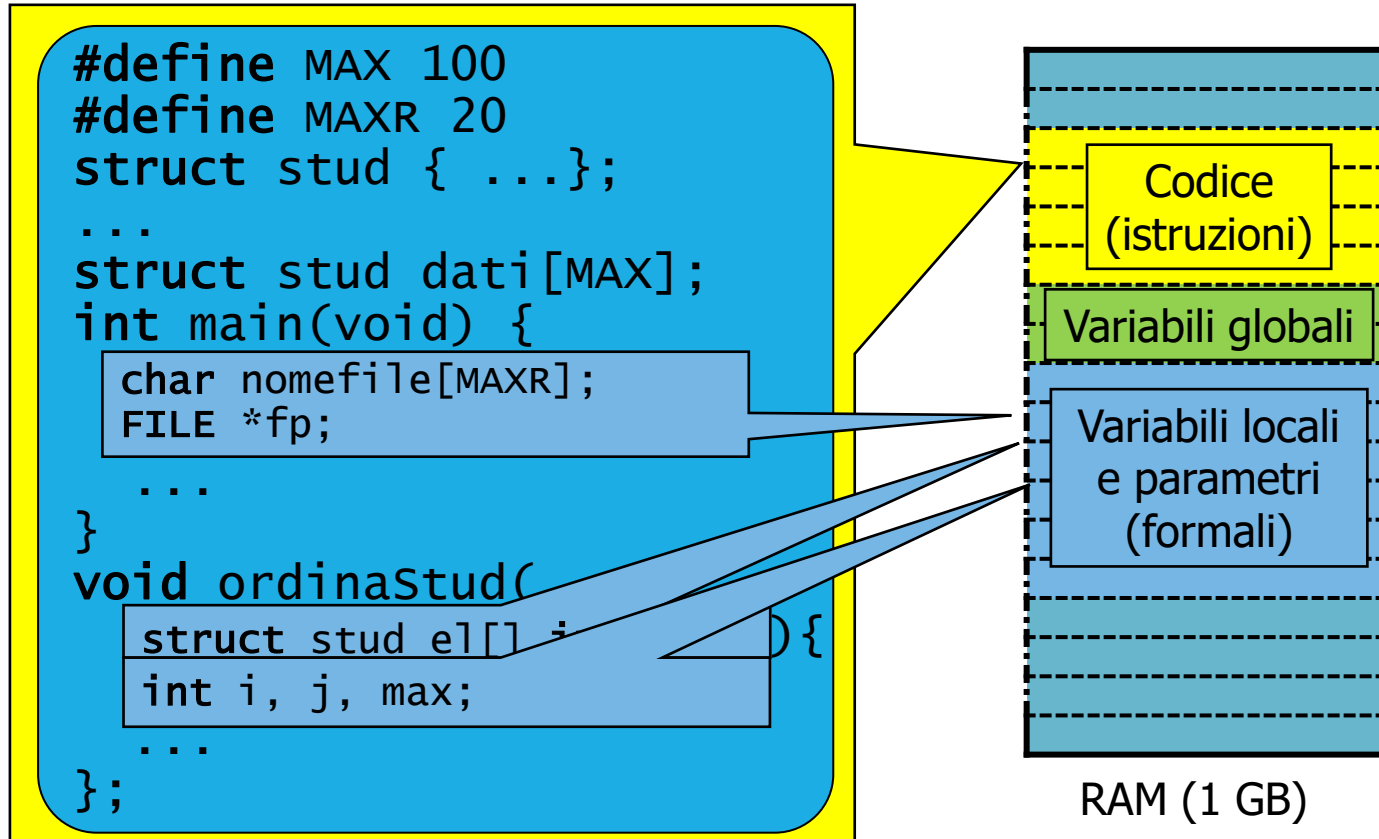


RAM (1 GB)

# Programma in memoria



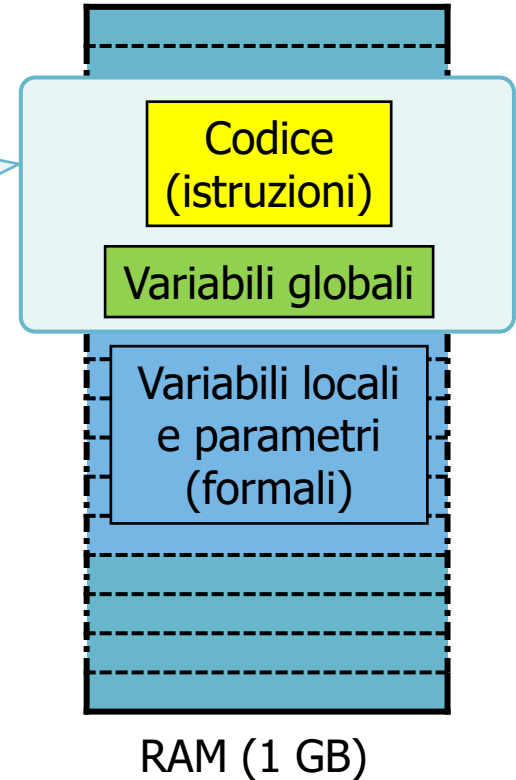
# Programma in memoria



# Programma in memoria

```
#define MAX 100
#define MAXR 20
struct ...
...
struct ...
int ...
char ...
FILE *fp;
...
}
void ordinaStud(
    struct stud el[], int n){
    int i, j, max;
    ...
};
```

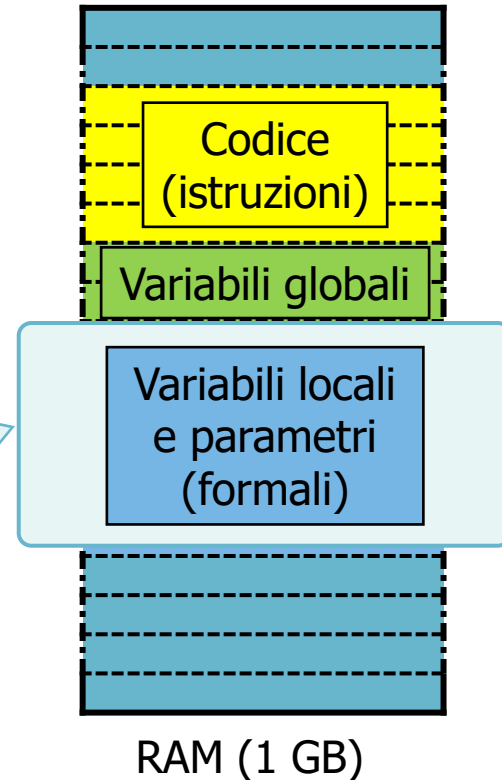
- in memoria (virtualmente) durante tutta l'esecuzione del programma
- indirizzi bassi



# Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *f;
    ...
}
void ...
struct ...
int ...
...
};
```

- in memoria (virtualmente) durante l'esecuzione della relativa funzione: allocate e de-allocate automaticamente
- stack frame nello stack



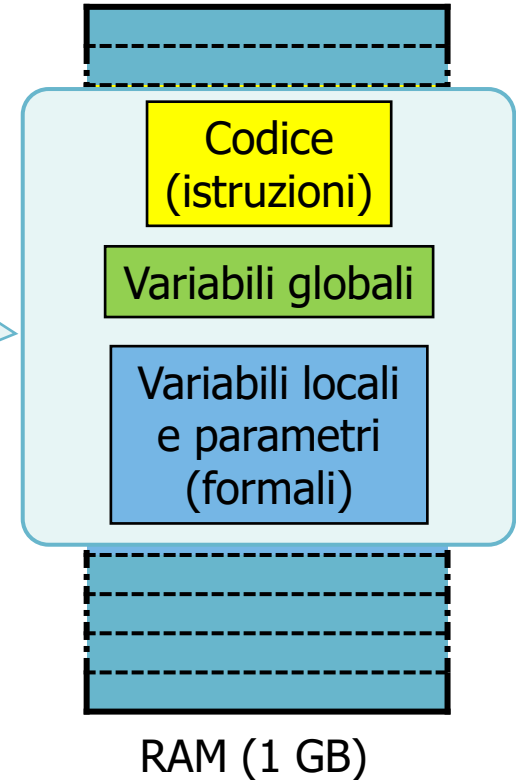
# Programma in memoria

```
#define MAX 100  
#define MAXR 20  
struct stud { ...};  
...  
struct stud el[MAX], int n){  
    int i, j, max;  
    ...  
};
```

la quantità di memoria da allocare  
è determinata (IMPLICITAMENTE)  
dal programmatore:

- istruzioni
- tipo e numero delle variabili
- dimensione dei vettori

```
struct stud el[], int n){  
    int i, j, max;  
    ...  
};
```





# Regole di allocazione automatica

- dimensioni:
  - variabili globali e locali hanno dimensione nota
  - vettori e matrici devono avere dimensione calcolabile
  - i vettori come parametri formali sono puntatori
- variabili globali:
  - allocate all'avvio del programma
  - restano in vita per tutto il programma
  - ricordano i valori assegnati da funzioni
  - l'attributo `static` limita la loro visibilità al file in cui compaiono

# Regole di allocazione automatica (2)

- variabili locali:

- raggruppate con i parametri formali in uno stack frame
- allocate nello stack ad ogni chiamata della funzione
- deallocate automaticamente all'uscita dalla funzione
- non ricordano i valori precedenti

- variabili locali con attributo `static`:

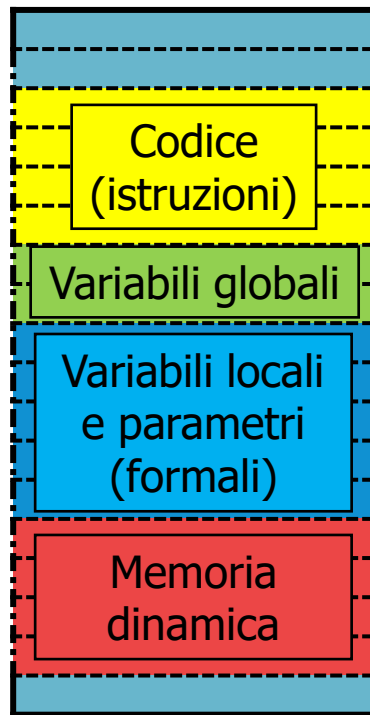
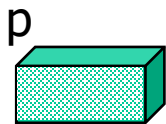
- visibilità limitata alla funzione
- allocate assieme alle variabili globali
- ricordano i valori (della chiamata precedente)

# Allocazione/rilascio espliciti

- Osservazione: **manca** un modo per poter **decidere**, durante l'esecuzione di un programma:
  - la **creazione/distruzione** un dato
  - il **dimensionamento** di un vettore o matrice
- Soluzione: istruzioni per allocare e de-allocare dati (memoria) **in modo esplicito**:
  - in funzione di dati forniti da chi esegue il programma
  - allocazioni e de-allocazioni sono (ovviamente) previste e gestite dall'autore del programma
  - la componente del sistema operativo che si occupa di allocazione/deallocazione è **l'allocatore di memoria dinamica**
  - la memoria dinamica si trova in un'area detta **heap**
  - alla memoria dinamica si accede **solo mediante puntatore**

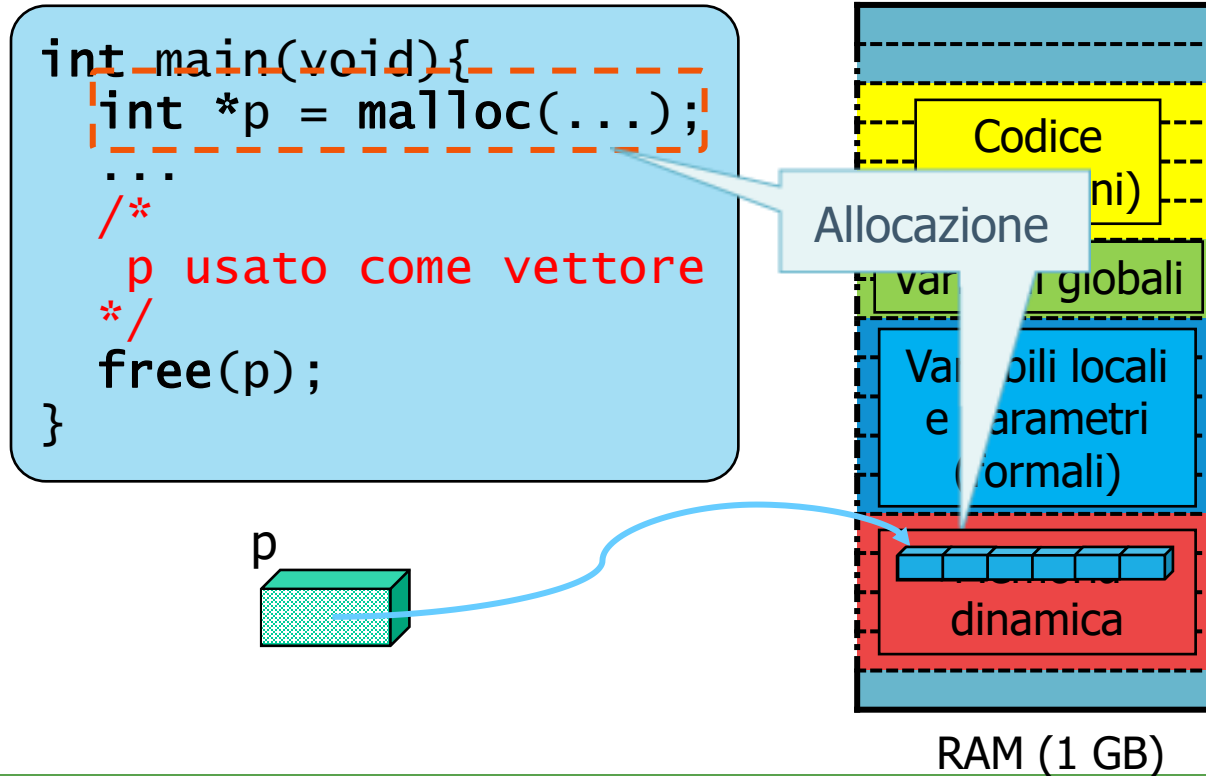
# Allocazione e rilascio espliciti: malloc e free

```
int main(void){  
    int *p = malloc(...);  
    ...  
    /*  
     p usato come vettore  
    */  
    free(p);  
}
```



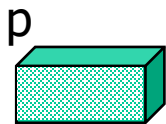
RAM (1 GB)

# Allocazione e rilascio espliciti: malloc e free



# Allocazione e rilascio espliciti: malloc e free

```
int main(void){  
    int *p = malloc(...);  
    ...  
    /*  
    p usato come vettore  
    */  
    free(p);  
}
```



Rilascio (de-allocazione)



RAM (1 GB)

# Tipologie di creazione/utilizzo di strutture dati

- dimensione
  - fissa determinata in fase di esecuzione
  - modificabile (aumentabile o diminuibile) mediante riallocazione
  - «contenitore» di singoli dati allocati a pezzi (aggiunta o eliminazione di elementi): es. liste concatenate, alberi, code, tabelle di simboli (tutte viste nel corso).
- Fasi di una struttura dati dinamica:
  - creazione (allocazione) esplicita
  - utilizzo con possibilità di:
    - riallocazione
    - Inserimenti
    - cancellazioni
  - distruzione (de-allocazione) esplicita.

# La funzione `malloc`

- La memoria in C viene allocata dinamicamente tramite la funzione `malloc`:

`void* malloc (size_t size);`

- `size` è il numero (intero) di byte da allocare
- il valore di ritorno è un puntatore:
  - contiene l'indirizzo iniziale della memoria allocata (NULL se non c'è memoria disponibile)
  - è tipo `void *`, tale da poter essere assegnato a qualunque tipo di puntatore



# La funzione `malloc`

- La memoria in C viene allocata dinamicamente tramite la funzione `malloc`:

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

- `size` è il numero (intero) di byte da allocare
- il valore di ritorno è un puntatore:
  - contiene l'indirizzo iniziale della memoria allocata (NULL se non c'è memoria disponibile)
  - è tipo `void *`, tale da poter essere assegnato a qualsiasi tipo di puntatore

`size_t` è un intero senza segno  
(si può usare come parametro  
attuale un `int`)

# Le regole

- Per usare `malloc` occorre includere `<stdlib.h>`
- La **dimensione** del dato è **responsabilità del programmatore**
  - Solitamente si ricorre all'operatore `sizeof` per determinare la dimensione (in byte) di un dato:
    - `sizeof(<tipo>)`
    - `sizeof <espressione riconducibile a tipo>`
- Al dato allocato si accede **unicamente** tramite **puntatore**
  - Il puntatore ritornato è **opaco**, **tocca al programmatore** passare al tipo desiderato mediante assegnazione a **opportuna variabile puntatore**

# Forma generale di chiamata a `malloc`:

Quattro forme, suddivise da

- tipo di `sizeof` (basato su un tipo o su una variabile/espressione)
- dalla presenza di cast esplicito o n o
  - cast implicito
    - `p = malloc(sizeof (<tipo>));`
    - `p = malloc(sizeof <espr>);`
  - cast esplicito (non obbligatorio, ma permette controllo di errore se `p` non è del tipo corretto):
    - `p = (<tipo> *) malloc(sizeof (<tipo>));`
    - `p = (<tipo> *) malloc(sizeof <espr>);`

# Esempi (1)

Data una `struct stud` ed una variabile `s` puntatore a `struct stud`

```
#define MAX 20
struct stud {char cognome[MAX];int matr; };
struct stud *s;
```

per calcolare la dimensione `struct` ci sono 2 modi:

`sizeof(struct stud)`

`sizeof(*s)`

dato puntato da `s`

## Esempi (2)

Allocazione di una `struct stud` puntata dalla variabile (puntatore) `s`:

- Con cast implicito:

```
s = malloc(sizeof(struct stud));
```

```
s = malloc(sizeof(*s));
```

- Con cast esplicito:

```
s = (struct stud *)malloc(sizeof(struct stud));
```

```
s = (struct stud *)malloc(sizeof(*s));
```

# Errori comuni

1. dimensione richiesta inferiore alla necessaria:

- a causa di uso del tipo errato in `sizeof`

```
double *pd;  
struct stud *ps, *v;  
int n;
```

```
...
```

```
pd = malloc (sizeof (int));
```

dovrebbe essere  
**sizeof (double)**

Un cast esplicito avrebbe reso più semplice  
l'identificazione dell'errore:

```
pd = (double *)malloc (sizeof (int));
```

# Errori comuni

1. dimensione richiesta inferiore alla necessaria:
  - a causa di uso del tipo errato in `sizeof`

```
double *pd;  
struct stud *ps, *v;  
int n;
```

```
...
```

```
pd = malloc
```

dovrebbe essere  
**sizeof (double)**

**L'errore SI VEDE!**

Un cast esplicito (da una parte double, dall'altra int  
l'identificazione dell'errore

```
pd = (double *)malloc (sizeof (int));
```

## Errori comuni (2)

2. uso del tipo puntatore a dato al posto del tipo del dato puntato

`sizeof(struct stud)`

```
ps=(struct stud *)malloc(sizeof(struct stud *));
```

3. dimensione n (serve per fabbricare un vettore) omessa o errata

`n * sizeof (*v)`

```
v = malloc (sizeof *v);
```

4. omissioni di sizeof

```
v = (struct stud *)malloc (n);
```

`n*sizeof(struct stud)`



# Errori comuni

5. assegnazione di puntatore incompatibile con il dato:

```
struct stud **ps;  
...  
ps = malloc(n*sizeof (struct stud));
```

**struct stud \***

```
struct stud **ps;  
...  
ps=(struct stud *)malloc(n*sizeof (struct stud));
```

il cast esplicito permette al compilatore di segnalare l'errore:

**ps vuole un asterisco in più**

# Errori comuni

5. assegnazione di puntatore incompatibile con il dato:

```
struct stud **ps;  
...  
ps = malloc(n*sizeof (struct stud));
```

**struct stud \***

```
struct stud **ps;  
...  
ps=(struct stud **)malloc(n*sizeof (struct stud *));
```

Versione corretta: si voleva un vettore di puntatori

# Errori comuni

5. assegnazione di puntatore incompatibile con il dato: secondo esempio

```
struct stud *ps;  
int *pi;
```

...

```
pi = malloc (sizeof (struct stud));
```

pi e sizeof(struct stud) non sono compatibili

```
struct stud *ps;  
int *pi;
```

...

```
pi=(struct stud *)malloc(sizeof (struct stud));
```

il cast esplicito permette al compilatore di segnalare l'errore:

pi deve puntare a intero

# Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

```
struct stud *ps;  
int *pi;  
...  
pi = malloc (sizeof *ps);
```

`pi` e `sizeof(*ps)` non sono compatibili: non lo dice il compilatore  
Si deve vedere «a occhio»

# Conseguenze degli errori sui puntatori

Errori individuati dal compilatore: occorre correggerli (il programma non compila correttamente)

Errori NON individuati dal compilatore

- succede spesso, perché la `malloc` riceve solo un NUMERO e ritorna in INDIRIZZO, quindi non conosce le «intenzioni» del programmatore
- Due possibilità
  - La dimensione allocata è SUPERIORE a quella necessaria. Non succede NULLA, se non lo SPRECO di memoria (allocata e non usata)
  - La dimensione allocata è inferiore al necessario
    - NON capita di solito NULLA nell'allocazione
    - DOPO, mentre si accede ai dati, SI RISCHIA DI USARE (mediante puntatore) MEMORIA NON ALLOCATA oppure ALLOCATA per un altro DATO

Cosa succede DOPO, se si accede a memoria non allocata oppure allocata ad altri dati?

- **crash** del programma per accesso a indirizzo non ammesso (esito auspicabile)
- accesso ad indirizzo legale, ma al di fuori della struttura dati allocata (errore subdolo) : si **SPORCA un altro dato**

# Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

```
struct stud *ps;  
int *pi;  
...  
pi = malloc (sizeof *ps);
```

Errore probabilmente non distruttivo: struct stud è più grande di un intero

# Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

```
struct stud *ps;  
int *pi;  
...  
ps = malloc (sizeof *pi);
```

Errore con conseguenze: un intero è probabilmente più piccolo di struct stud

## Errori comuni (2)

Si rischia di allocare sempre meno del necessario

2. uso del tipo puntatore a dato al posto del tipo del dato puntato

```
ps=(struct stud *)malloc(sizeof(struct stud *));
```

`sizeof(struct stud)`

3. dimensione n (serve per fabbricare un vettore) omessa o errata

```
v = malloc (sizeof *v);
```

`n * sizeof (*v)`

4. omissione di sizeof

```
v = (struct stud *)malloc (n);
```

`n*sizeof(struct stud)`



# Memoria dinamica insufficiente

Succede poco, indica che non c'è più memoria allocabile nell'heap, per la dimensione richiesta (provare ad allocare  $n * \text{sizeof}(\text{int})$ , con  $n$  molto grande):

- `malloc` ritorna `NULL`
- opportuno testare, segnalando o uscendo con `exit` o `return`

```
int *p;  
...  
p = malloc(sizeof(int));  
if (p == NULL)  
    printf("Errore di allocazione\n");  
else  
    ...
```

# La funzione `calloc`

- Equivale a:

`malloc(n*size);`

con memoria ritornata azzerata

`void* calloc (size_t n, size_t size);`

La `calloc` ha costo (in tempo)  $O(n)$ , a causa dell'azzeramento, mentre `malloc` è  $O(1)$ . Tuttavia, molto sovente, l'azzeramento è opportuno e necessario (andrebbe fatto comunque)

# La funzione **free**

- Tutta la memoria allocata dinamicamente con `malloc/calloc` viene restituita tramite la funzione `free` (`<stdlib.h>`)  
**`void free (void* p);`**
  - `p` punta alla memoria (precedentemente allocata) da liberare
- Viene di solito chiamata quando è terminato il lavoro sulla struttura dinamica, affinché la memoria possa essere riutilizzata
- **ATTENZIONE:** l'allocatore mantiene internamente una tabella di ciò che ha allocato:
  - Si può solo chiamare `free` per un indirizzo precedentemente ritornato da `malloc/calloc` (o `realloc`)
  - **NON** si può liberare un PEZZO della memoria ottenuta (allocata) con `malloc/calloc` (o `realloc`)

# Uso di `free` consigliato, ma non obbligatorio

- al termine dell'esecuzione di un programma la memoria viene comunque liberata (in molti casi questo è sufficiente)
- Ma è possibile che SIA OPPORTUNO LIBERARE per poter OTTENERE nuova memoria DURANTE l'esecuzione: es. programma che ripete iterativamente un lavoro che richiede allocazione
- Attenzione ai **memory leak** (dimenticare di de-allocare):
  - la mancata de-allocazione di una porzione di memoria
  - Non si può riutilizzare la memoria per un nuovo dato da allocare. Effetto: aumenta la probabilità (con programmi che allocano molto) di `malloc/calloc` che ritornano `NULL`.

# Esempio di memory leak

```
int *vett = malloc(10 * sizeof(int));  
...  
// uso di vett, SENZA liberazione  
vett = malloc(25 * sizeof(int));
```

ora la porzione di memoria allocata dalla prima `malloc` non è più indirizzabile né utilizzabile per ulteriori allocazioni (è ancora allocata, ma non puntata e non usata)

# La funzione `realloc`

- In C la dimensione della memoria allocata può essere modificata aggiungendo o togliendo una porzione in fondo tramite `realloc`:

```
void* realloc (void* p, size_t size);
```

- `p` punta a memoria precedentemente allocata
- `size` è la nuova dimensione richiesta (maggiore o minore)
- il valore di ritorno è un puntatore

- Uso tipico:

```
p = malloc (oldSize);  
...  
p = realloc (p, newSize);  
// si lavora sulla struttura dati espansa o ristretta  
...
```

`newSize` è la nuova  
dimensione, diversa da  
`oldSize`

# Cosa succede?

- La riduzione della dimensione è sempre possibile
- L'aumento della dimensione può:
  - essere impossibile (non c'è memoria extra disponibile) e si ritorna NULL
  - essere possibile: esiste memoria disponibile contigua, alla FINE del blocco già allocato (quindi espandibile). Si ritorna il puntatore `p` invariato
  - essere possibile, ma altrove (non c'è spazio sufficiente alla fine del blocco):
    - Si alloca un nuovo intervallo in memoria (di dimensione `newSize`)
    - si ricopia con costo lineare nella dimensione ( $O(n)$ ) il contenuto della vecchia porzione di memoria nella nuova
    - si ritorna un puntatore `p` aggiornato.

# Cosa succede?

- La riduzione della dimensione è sempre possibile
- L'aumento della dimensione può:
  - essere impossibile (non c'è memoria extra disponibile) e si ritorna NULL
  - essere possibile: esiste memoria disponibile contigua, alla FINE del blocco già allocato (quindi espandibile). Si ritorna il puntatore `p` invariato
  - essere possibile, ma altrove (non c'è spazio sufficiente alla fine del blocco):
    - Si alloca un nuovo intervallo in memoria (di dimensione `newSize`)
    - si ricopia con costo lineare nella dimensione ( $O(n)$ ) il contenuto della vecchia porzione di memoria nella nuova
    - si ritorna un puntatore `p` aggiornato.

**ATTENZIONE:** la `realloc` nasconde quindi un costo lineare



# Implementazione ad alto livello della `realloc`

```
// non è la vera implementazione - serve solo per capirla
void* realloc (void* p, size_t size) {
    size_t oldSize = cercaDimensione (p, ...); // cerca in tabella la dimensione precedente
    if (/*si può espandere o ridurre*/) {
        cambiaDimensione (p, ...); // toglì o aggiungi un pezzo in fondo
        return p;
    }
    else {
        void *newp = malloc(size); // nuova allocazione
        copiaMemoria(newp,p,min(size,oldSize));
        free(p);
        return newp;
    }
}
```

# Implementazione ad alto livello della `realloc`

// non è la vera implementazione - serve solo per capirla

```
void* realloc (void* p, size_t size) {  
    size_t oldSize = cercaDimensione (p, ...); // cerca in tabella la dimensione precedente  
    if (/*si può espandere o ridurre*/) {  
        cambiaDimensione (p, ...); // toglì o aggiungi un pezzo in fondo  
        return p;  
    }  
    else {  
        void *newp = malloc(size); // nuova allocazione  
        copiaMemoria(newp,p,min(size,oldSize));  
        free(p);  
        return newp;  
    }  
}
```

`copiaMemoria` ha complessità  
 $O(\min(\text{size}, \text{oldSize}))$

# Strutture dati dinamiche

---

VETTORI E MATRICI ALLOCATI DINAMICAMENTE

# Strutture dati dinamiche

La dimensione delle strutture dati dinamiche:

- è nota solo in fase di esecuzione
- può variare nel tempo.

Possono anche contenere dati aggregati in quantità non note a priori e variabili nel tempo (liste, Cap. 4).

# Vettori dinamici

- La dimensione è nota solo in fase di esecuzione del programma
- Può variare per riallocazione
- Si evita il sovradimensionamento del vettore nonché i suoi limiti:
  - è necessario conoscere la dimensione massima (costante)
  - data la dimensione massima e una parte iniziale del vettore effettivamente utilizzata, quella restante è sprecata.
- Soluzione:
  - uso di puntatore, sfruttando la dualità puntatore-vettore, con entrambi le notazioni
  - allocazione mediante `malloc/calloc`
  - rilascio mediante `free`
  - Il resto è identico al vettore sovradimensionato in modo statico.

# Esempio

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di dati non è nota al programmatore, né sovradimensionabile, ma è acquisita come primo dato da tastiera

# invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d",&N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f",&v[i]);
    }
```

```
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```

# invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d",&N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f",&v[i]);
    }
```

```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```


allocazione vettore dinamico

Controllo allocazione riuscita




# invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d",&N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f",&v[i]);
    }
```



```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```



# invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d",&N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f",&v[i]);
    }
```

```
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```



de-allocazione

# La dimensione del vettore dinamico

- **ATTENZIONE:** bisogna conoscere il numero di dati per creare e usare il vettore dinamico!
- Se il numero di dati (ignoto) fosse segnalato da un terminatore (es. input del valore 0):
  - 2 letture da file (quindi non va bene da tastiera): la prima per calcolare il numero di dati, la seconda per memorizzarli
  - uso di `realloc`, tenendo presente il suo costo lineare nascosto

# Esempio (modificato)

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di numeri non è nota al programmatore, né sovradimensionabile, ~~ma è acquisita come primo dato da tastiera~~

# Esempio (modificato)

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di numeri non è nota al programmatore, né sovradimensionabile. I dati terminano con un dato non valido.

# Ri-allocazione: soluzione A

Il vettore viene ri-allocato ad ogni iterazione:

- vettore dinamico di dimensione iniziale  $N=1$
- riallocazione ad ogni nuovo dato con  $N$  incrementato di 1
- **$O(N^2)$** : *pur trattandosi di un caso peggiore, difficilmente realizzabile, può accadere (quasi ogni volta che si alloca, non si riesce ad «allargare», e si deve «spostare» il vettore).*

# Ri-allocazione: soluzione A

Il vettore viene ri-allocato ad ogni iterazione:

- vettore dinamico di dimensione iniziale  $N=1$
- riallocazione ad ogni nuovo dato con  $N$  incrementato di 1
- **$O(N^2)$** : *pur trattandosi di un caso peggiore, difficilmente realizzabile, può accadere (quasi ogni volta che si alloca, non si riesce ad «allargare», e si deve «spostare» il vettore).*

**SCONSIGLIATA !**

# invertiOrdine.c (con ri-allocazione A)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N=1, i;
    v = malloc(N*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==N) {
            // attivato sempre eccetto con i==0
            N = N+1;
            v = realloc(v,N*sizeof(float));
            // controllo di errore omissso
```

```
        }
        v[i++] = d;
        printf("Elemento %d: ", i) ;
    }
}
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```



# invertiOrdine.c (con ri-allocazione A)

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

int main() {
    float *v; int N=1, i;
    v = malloc(N*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==N) {
            // attivato sempre eccetto con i==0
            N = N+1;
            v = realloc(v,N*sizeof(float));
            // controllo di errore omissso
            v[i++] = d;
            printf("Elemento %d: ", i) ;
        }
    }
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```

allocazione iniziale

Riallocazione  
(con N incrementato)

# Ri-allocazione: soluzione B

Il vettore viene ri-allocato un numero logaritmico di volte:

- vettore dinamico di dimensione iniziale  $N=1$
- controllo se vettore pieno
- riallocazione se pieno con  $N$  raddoppiato (sovradimensionamento)
- **$O(N \log N)$** : compromesso memoria (sovradimensionata, al peggio quasi doppia) tempo (linearitmico anziché quadratico)

**CONSIGLIATA !**

# invertiOrdine.c (con ri-allocazione B)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, MAXN=1, i=0;
    v = malloc (MAXN*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==MAXN) {
            // numero logaritmico di attivazioni
            MAXN = MAXN*2;
            v = realloc(v,MAXN*sizeof(float));
            // controllo di errore omissso
```

```
        }
        v[i++] = d;
        printf("Elemento %d: ", i) ;
    }
}
N = i; // compreso tra MAXN/2 e MAXN
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```

# invertiOrdine.c (con ri-allocazione B)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, MAXN=1, i=0;
    v = malloc (MAXN*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==MAXN) {
            // numero logaritmico di attivazioni
            MAXN = MAXN*2;
            v = realloc(v,MAXN*sizeof(float));
            // controllo di errore omezzo
        }
        v[i++] = d;
        printf("Elemento %d: ", i) ;
    }
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
}
```

allocazione iniziale

controllo se pieno

riallocazione con raddoppio

# Matrici dinamiche

Due possibilità:

- Soluzione 1 (meno flessibile):
  - vettore **MONODIMENSIONALE** dinamico di  $nr \times nc$  elementi
  - organizzazione **manuale** di righe e colonne su vettore: l'elemento  $(i,j)$  si trova in posizione  $nc*i + j$ .
- Soluzione 2:
  - vettore **BIDIMENSIONALE** dinamico di  $nr$  puntatori a righe
  - iterazione sulle  $nr$  righe ( $nc$  colonne) per allocare un vettore di tipo desiderato di  $nc$  ( $nr$ ) elementi
  - Il resto è identico alla matrice sovradimensionata in modo statico.

# Esempio

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in una matrice
- Stampare successivamente la matrice trasposta (righe e colonne scambiate di ruolo)
- Le dimensioni della matrice (righe e colonne) non sono note al programmatore, né sovradimensionabili, ma sono acquisite come primo dato da tastiera

# Con vettore dinamico monodimensionale

```
...  
float *v;  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
v = (float *) malloc(nr*nc*(sizeof (float)));  
if (v==NULL) exit(1);  
for (i=0; i<nr; i++) {  
    printf("Inserisci riga %d\n", i);  
    for (j=0; j<nc; j++)  
        scanf("%f", &v[nc*i+j]);  
}
```

```
printf("Matrice trasposta\n");  
for (j=0; j<nc; j++) {  
    for (i=0; i<nr; i++)  
        printf("%6.2f", v[nc*i+j]);  
    printf("\n");  
}  
free(v);  
...
```

# Con vettore dinamico monodimensionale

```
...  
float *v;  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
v = (float *) malloc(nr*nc*(sizeof (float)));  
if (v==NULL) exit(1);  
for (i=0; i<nr; i++) {  
    printf("Inserisci riga %d\n", i);  
    for (j=0; j<nc; j++)  
        scanf("%f", &v[nc*i+j]);  
}
```

allocazione

```
printf("Matrice trasposta\n");  
for (j=0; j<nc; j++) {  
    for (i=0; i<nr; i++)  
        printf("%.2f", v[nc*i+j]);  
    printf("\n");  
}  
free(v);  
...
```

controllo  
di errore

Gestione manuale: [nc\*i+j]



# Con matrice dinamica bidimensionale

```
...
float **m;
int nr,nc,i,j;

printf("nr nc: "); scanf("%d%d", &nr, &nc);
m = (float **) malloc(nr*(sizeof (float *)));
if (m==NULL) exit(1);
for (i=0; i<nr; i++) {
    printf("Inserisci riga %d\n", i);
    m[i] = (float *) malloc(nc*sizeof (float));
    if (m[i]==NULL) exit(1);
```

```
    for (j=0; j<nc; j++)
        scanf("%f", &m[i][j]);
}
printf("Matrice trasposta\n");
for (j=0; j<nc; j++) {
    for (i=0; i<nr; i++)
        printf("%6.2f", m[i][j]);
    printf("\n");
}
for (i=0; i<nr; i++)
    free(m[i]);
free(m);
...
```

# Con matrice dinamica bidimensionale

```
...  
float **m;  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
m = (float **) malloc(nr*(sizeof (float *)));  
if (m==NULL) exit(1);  
for (i=0; i<nr; i++) {  
    printf("Inserisci riga %d\n", i);  
    m[i] = (float *) malloc(nc*sizeof (float));  
    if (m[i]==NULL) exit(1);
```

vettore di vettori di float

float \*\*: vettore di puntatori

allocazione di vettore di  
nr puntatori a riga

```
for (i=0; i<nc; i++)  
    printf("%6.2f", m[i][j]);  
printf("\n");  
}  
for (i=0; i<nr; i++)  
    free(m[i]);
```

vettore di float: uno per riga

# Con matrice dinamica bidimensionale

```
...  
float **m;  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
m = (float **) malloc(nr*(sizeof (float *)));  
if (m==NULL) {  
    printf("Inserisci riga %d\n", i);  
    return 1;  
}  
for (i=0; i<nr; i++)  
    m[i] = (float *) malloc(nc*sizeof (float));  
if (m[i]==NULL) {  
    printf("Inserisci riga %d\n", i);  
    return 1;  
}  
for (j=0; j<nc; j++)  
    f("%f", &m[i][j]);  
printf("Matrice trasposta\n");  
for (j=0; j<nc; j++) {  
    for (i=0; i<nr; i++)  
        printf("%6.2f", m[i][j]);  
    printf("\n");  
}  
for (i=0; i<nr; i++)  
    free(m[i]);  
free(m);  
...
```

liberazione memoria  
dinamica (niente di automatico)

Prima le singole righe  
(una alla volta)

Poi il vettore dei  
puntatori alle righe

# Vettori e matrici creati da funzioni

- Un vettore o matrice dinamici sono accessibili a partire da un puntatore
- Il puntatore è un dato: può quindi essere passato e/o ritornato da funzioni, come pure copiato tra variabili
- Le variabili puntatore esistono fintanto che è in essere la funzione dove sono dichiarate e sono visibili in essa
- Matrici e vettori creati in una funzione  $f$  (ad esempio un vettore dinamico  $v$ ) sono
  - a volte usate solamente nella funzione  $f$  in cui sono allocate e de-allocate
  - altre volte può essere necessario che siano accessibili da altre funzioni (chiamate da  $f$ , oppure che chiamano  $f$ )

- Per fare in modo che una funzione  $g$  (chiamata da  $f$ ) usi il vettore  $v$  (un puntatore), è sufficiente passare il puntatore per valore:

$g(\dots, v, \dots)$

- Per rendere  $v$  accessibile al programma chiamante (funzione  $h$  che chiama  $f$ ):
  - si dichiara il puntatore come variabile globale (**sconsigliato!**)
  - si inserisce il puntatore tra i parametri della funzione e lo si modifica (passaggio by pointer/reference, quindi, in C “by value” di un puntatore a puntatore). Ad esempio,  $h$  chiamerà:  $f(\dots, \&v, \dots)$
  - si ritorna il puntatore come valore di ritorno della funzione.  
Ad esempio:  $v = f(\dots)$

# Esempio

- Si realizzino due funzioni che allocano (`malloc2d`) o liberano (`free2d`) una matrice bidimensionale di elementi di tipo `Item` con `nr` righe e `nc` colonne.
- La funzione di allocazione `malloc2d` ha 2 versioni:
  - Puntatore ritornato come **risultato**: `malloc2dR` dove il puntatore alla matrice è restituito come valore di ritorno della funzione
  - Passaggio by **pointer**/reference: `malloc2dP` dove il puntatore alla matrice è restituito tra i parametri della funzione

# Puntatore come valore di ritorno della funzione

```
typedef ... Item;
Item **malloc2dR(int nr, int nc);
void free2d(Item **m, int nr);
...
void h (/* parametri formali */) {
    Item **matr;
    int nr, nc;
    ...
    matr = malloc2dR(nr, nc);
    ... /* lavoro su matr */
    free2d(matr,nr);
}
```

```
Item **malloc2dR(int nr, int nc) {
    Item **m;
    int i;
    m = malloc (nr*sizeof (Item *));
    for (i=0; i<nr; i++) {
        m[i] = malloc (nc*sizeof (Item));
    }
    return m;
}
void free2d(Item **m, int nr) {
    int i;
    for (i=0; i<nr; i++) {
        free(m[i]);
    }
    free(m);
}
```

# Puntatore come valore di ritorno della funzione

```
typedef ... Item;
Item **malloc2dR(int nr, int nc);
void free2d(Item **m, int nr);
...
void h (/* parametri formali */) {
    Item **matr;
    int nr, nc;
    ...
    matr = malloc2dR(nr, nc);
    ... /* lavoro su matr */
    free2d(matr,nr);
}
```

funzione di tipo puntatore  
a vettore di Item

matrice di Item

```
Item **malloc2dR(int nr, int nc) {
    Item **m;
    int i;
    m = malloc (nr*sizeof (Item *));
    for (i=0; i<nr; i++) {
        m[i] = malloc (nc*sizeof (Item));
    }
    return m;
}

void free2d(Item **m, int nr) {
    int i;
    for (i=0; i<nr; i++) {
        free(m[i]);
    }
    free(m);
}
```



# Puntatore come valore di ritorno della funzione

```
typedef ... Item;
```

funzione di tipo puntatore  
a vettore di Item

```
void h (/* parametri formali */) {
```

m variabile locale (doppio  
puntatore): puntatore a vettore di  
puntatori (righe)

```
... /* lavoro su matr */
```

```
free2d(matr,nr);
```

```
}
```

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        m[i] = malloc (nc*sizeof (Item));  
    }  
    return m;  
}  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

m ritornato come risultato

# Puntatore come valore di ritorno della funzione

```
typedef ... Item;  
Item **malloc2dR(int nr, int nc);
```

Free più facile:

- riceve puntatore /by value)
- non restituisce risultato

```
int nr, nc;
```

```
...
```

```
matr =
```

```
... /%
```

```
free2d
```

```
}
```

Prima libera le singole righe,  
poi il vettore di puntatori

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        m[i] = malloc (nc*sizeof (Item));  
    }
```

```
    return m;  
}
```

```
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

# Puntatore come parametro by pointer/reference

```
typedef ... Item;
void malloc2dP(Item ***mp, int nr, int nc);
void free2d(Item **m, int nr);
...
void h (/* parametri formali */) {
    Item **matr;
    int nr, nc;
    ...
    malloc2dP(&matr, nr, nc);
    ... /* lavoro su matr */
    free2d(matr,nr);
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {
    Item **m;
    int i;
    m = (Item **)malloc (nr*sizeof(Item *));
    for (i=0; i<nr; i++)
        m[i] = (Item *)malloc (nc*sizeof(Item));
    *mp = m;
}
void free2d(Item **m, int nr) {
    int i;
    for (i=0; i<nr; i++) {
        free(m[i]);
    }
    free(m);
}
```

# Puntatore come parametro by pointer/reference

```
typedef ... Item;
void malloc2dP(Item ***mp, int nr, int nc);
void free2d(Item **m, int nr);
...
void h (/* parametri formali */) {
    Item **matr;
    int nr, nc;
    ...
    malloc2dP(&matr, nr, nc);
    ... /* lavoro su matr */
    free2d(matr,nr);
}
```

funzione di tipo void, con  
parametro puntatore a  
a matrice di Item

matrice di Item

```
void malloc2dP(Item ***mp, int nr, int nc) {
    Item **m;
    ...
    (Item *));
    ...
    zeof(Item));
    *mp = m;
}

void free2dP(Item **m, int nr) {
    int i;
    for (i=0; i<nr; i++) {
        free(m[i]);
    }
    free(m);
}
```

# Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);  
void free2d(Item **, int nr);  
...
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
}  
  
void free2dP(Item ***mp, int nr) {
```

mp: puntatore a matrice di Item  
puntatore (triplo): puntatore a un  
puntatore a un vettore (puntatore a Item)  
di righe

```
... /* lavoro su matr */  
free2d(matr,nr);  
}
```

```
for (i=0; i<nr; i++) {  
    free(m[i]);  
}  
free(m);  
}
```

# Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);
```

punta a variabile del programma chiamante (puntatore doppio) in cui occorre trasferire il risultato

```
int nr, ...  
...  
malloc2dP(&matr, nr, nc);  
... /* lavoro su matr */  
free2d(matr, nr);  
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    ;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
}  
  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

# Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);
```

m variabile locale (doppio puntatore):  
non obbligatoria ma comoda  
«dentro» alla funzione

```
Item matr;  
int nr, nc;  
...  
malloc2dP(&matr, nr, nc);  
...  
free;  
}
```

m copiata in \*mp (risultato)  
al termine della funzione  
(equivale a matr = m)

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    int i;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
}  
  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

VARIANTE (meno leggibile ma più compatta):  
si lavora direttamente sulla variabile del programma  
chiamante (\*mp) senza usare una variabile locale.



```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

le parentesi tonde sono necessarie per la  
precedenza degli operatori

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

le parentesi tonde sono necessarie per la  
precedenza degli operatori

Molti programmatori seguono questa strategia (a volte perché non prendono in considerazione la variabile locale):  
AUMENTA LA PROBABILITA' DI ERRORE (proprio perché si dimenticano parentesi): \*mp[i] è SBAGLIATO!!!

# Vettori a dimensione variabile

Se serve «solo» dimensionare vettori e matrici in fase di esecuzione, anche in C esistono i

vettori a lunghezza variabile

(variable length arrays)

- Si dichiara un vettore/matrice come variabile locale usando, come dimensioni, variabili o espressioni anziché costanti.
- Allocazione e deallocazione sono automatiche e implicite (è comodo).

L'uso di vettori a lunghezza variabile è scoraggiato in quanto:

- con essi si realizza un sottoinsieme di ciò che si può fare con l'allocazione dinamica
- presentano svantaggi:
  - non si può controllare se l'allocazione è andata a buon fine (mentre invece il puntatore ritornato da `malloc`/`calloc`/`realloc` si può confrontare con `NULL`): l'effetto è un crash, quando lo stack è troppo piccolo.
  - Il vettore è cancellato all'uscita dalla funzione, ma il programmatore può pensare che esista ancora e continua a farvi riferimento.

# Esempio

```
void inverti(int N) {  
    int i;  
    float v[N];  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inverso\n");  
    for (i=N-1; i>=0; i--)  
        printf("Elemento %d: %f\n", i, v[i]);  
}
```

# Esempio con ERRORE !!! (ritornare v)

```
float *inverti(int N) {  
    int i;  
    float v[N];  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inversa\n");  
    for (i=N-1; i>=0; i--)  
        printf("Elemento %d: %f\n", i, v[i]);  
    return v;  
}
```

V è nello stack.

Quando la funzione termina viene deallocato automaticamente.

Il programma chiamante riceve un puntatore a memoria appena rilasciata

# Esempio corretto (con malloc)

```
float *inverti(int N) {  
    int i;  
    float *v = (float *)malloc(N*sizeof(float));  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inverso\n");  
    for (i=N-1; i>=0; i--)  
        printf("Elemento %d: %f\n", i, v[i]);  
    return v;  
}
```

V è nello heap.

Quando la funzione termina NON viene deallocato automaticamente.

Il programma chiamante riceve un puntatore a memoria ancora allocata (andrà liberata con free)