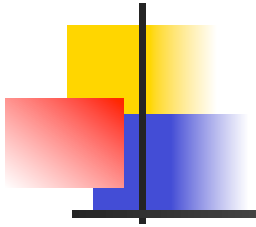
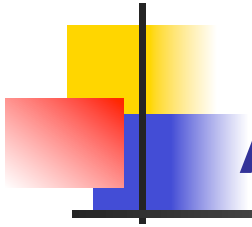


Puntatori e strutture dati dinamiche: allocazione della memoria e modularità in linguaggio C

Capitolo 3: L'allocazione dinamica della memoria

G. Cabodi, P. Camurati, P. Pasini, D. Patti, D. Vendraminetto

MAGGIOLI EDITORE APOGEO education 2016



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.

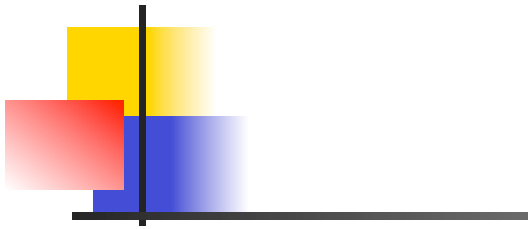
Da codice sorgente a eseguibile

Le variabili sono soggette a precise regole di:

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

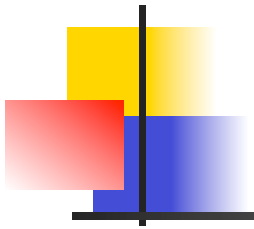
Le variabili si distinguono in:

- **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.

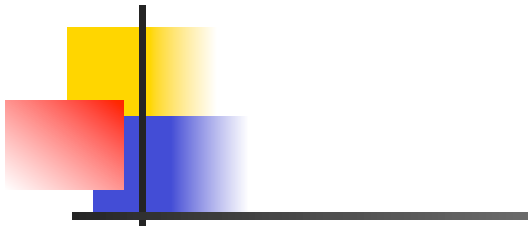


Vantaggi:

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

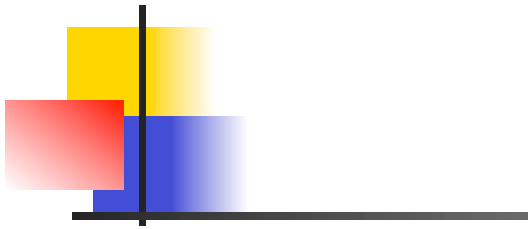
Svantaggi:

- minore modularità, leggibilità, affidabilità



Variabili **locali:**

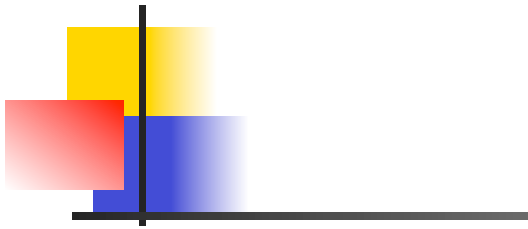
- variabili definite all'interno delle funzioni (main incluso)
- parametri alle funzioni
- temporanee (iniziano ad esistere quando è chiamata la funzione e cessano quando se ne esce)
- visibili solo nella funzione in cui sono dichiarate.



Compilatore:

- programma che esegue un'analisi
  - lessicale
  - sintattica
  - semantica

  del codice sorgente
- e genera un codice oggetto (in linguaggio macchina)



Il codice oggetto contiene riferimenti a funzioni di libreria

Linker:

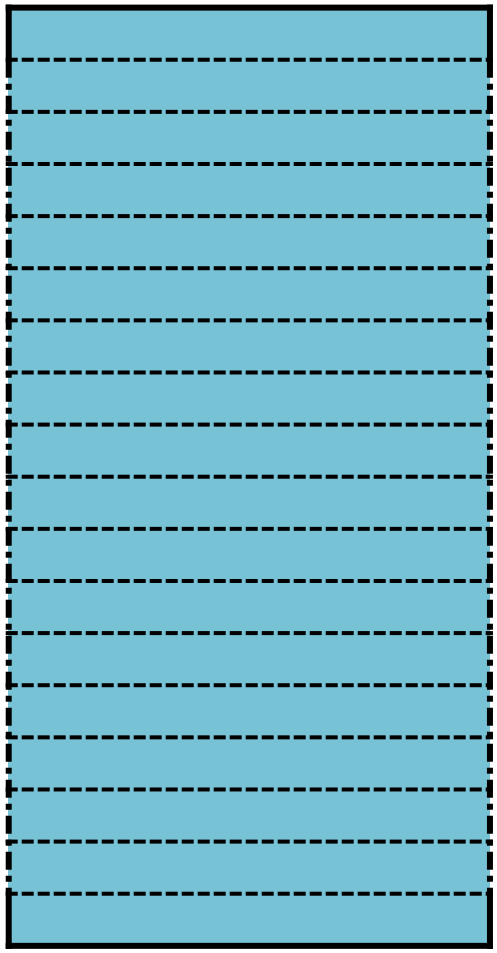
- programma che risolve:
  - i riferimenti a funzioni di libreria
  - I riferimenti reciproci tra più file oggetto.
- e genera un codice eseguibile

Loader:

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

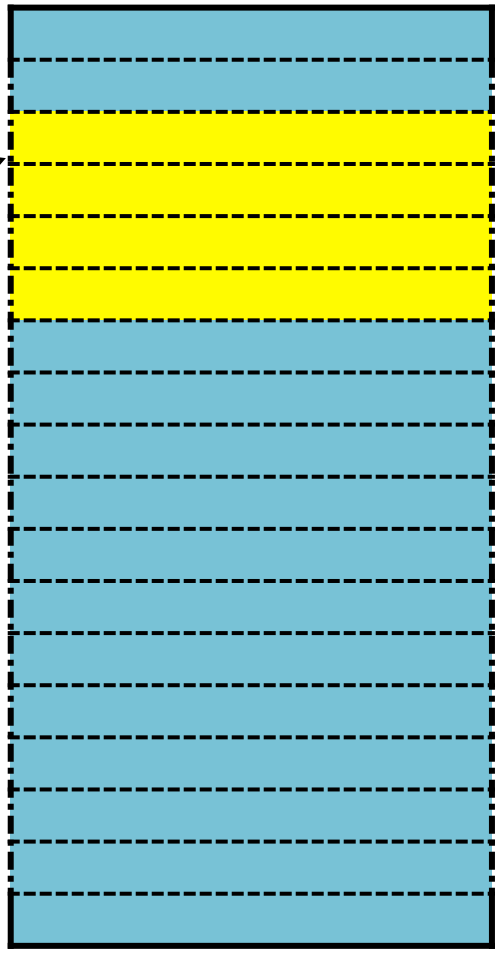
Programma in memoria

```
#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;
   ...
};
```



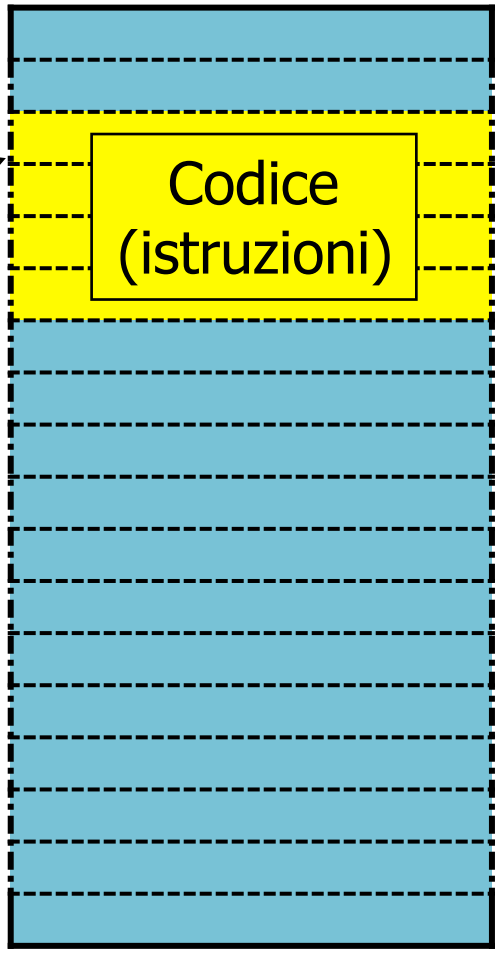
RAM (1 GB)

```
#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;
   ...
};
```



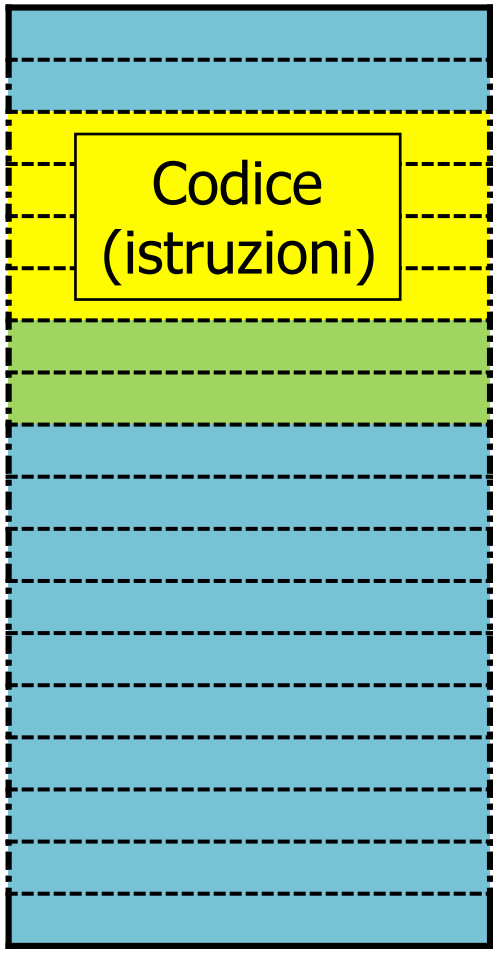
RAM (1 GB)

```
#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;
   ...
};
```



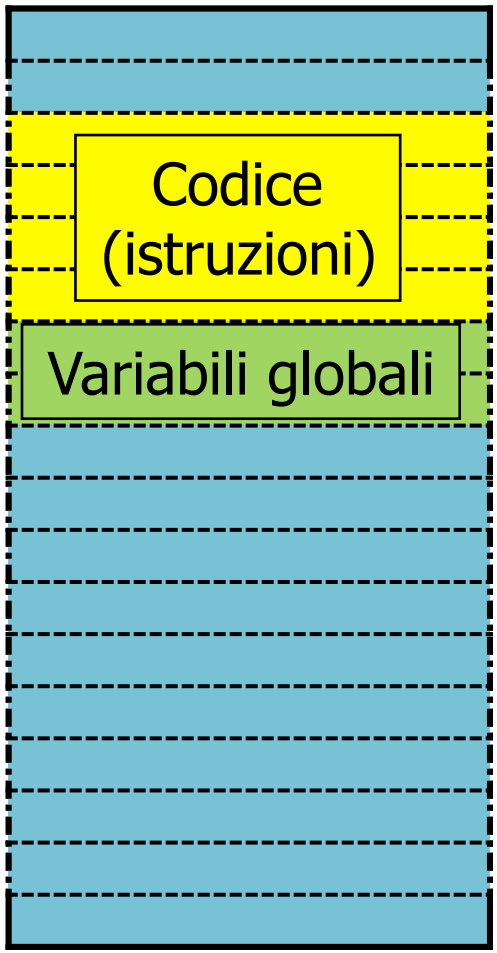
RAM (1 GB)

```
#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;
   ...
};
```



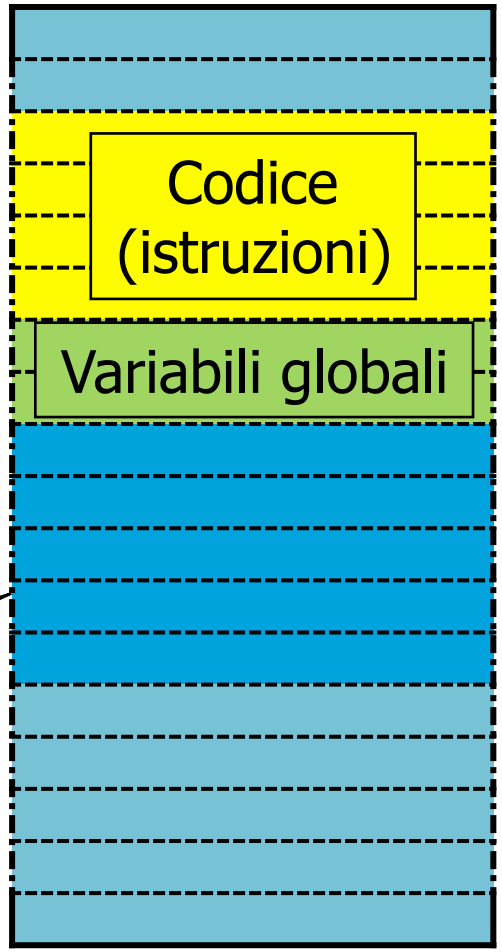
RAM (1 GB)

```
#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;
   ...
};
```



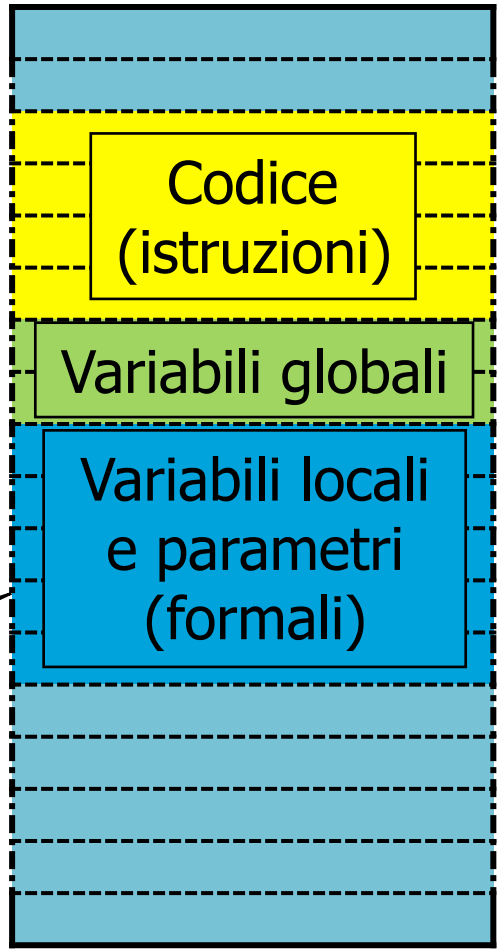
RAM (1 GB)

```
#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 i, j, max;
   ...
};
```



RAM (1 GB)

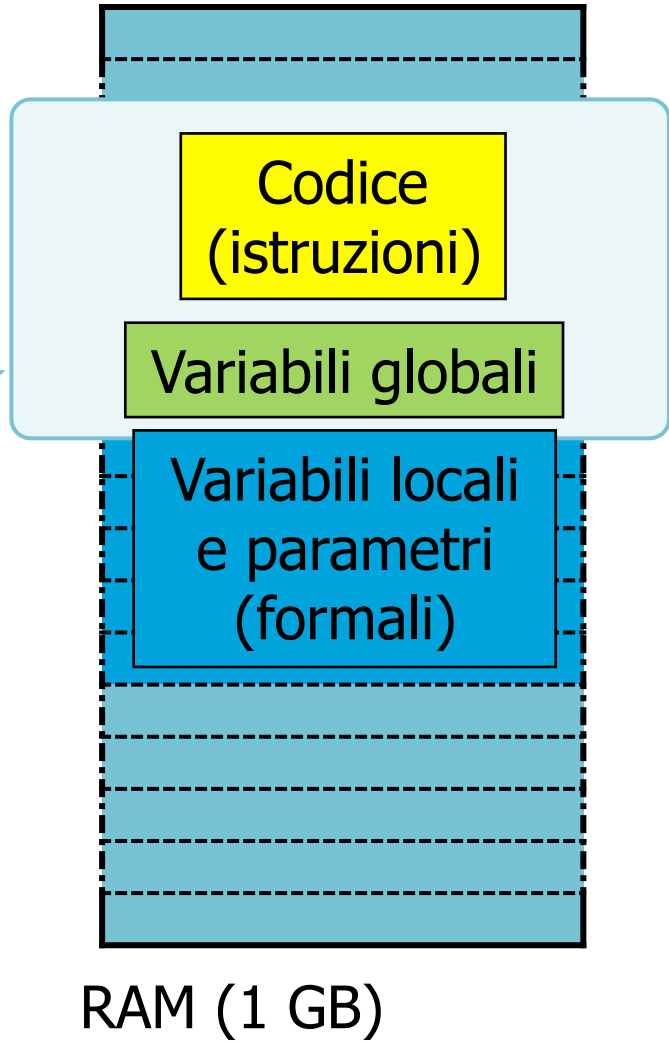
```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
  char nomefile[MAXR];
    FILE *fp;
   ...
}
void ordinaStud(
   struct stud el[] ;          ){
   int i, j, max;
   ...
};
```



RAM (1 GB)

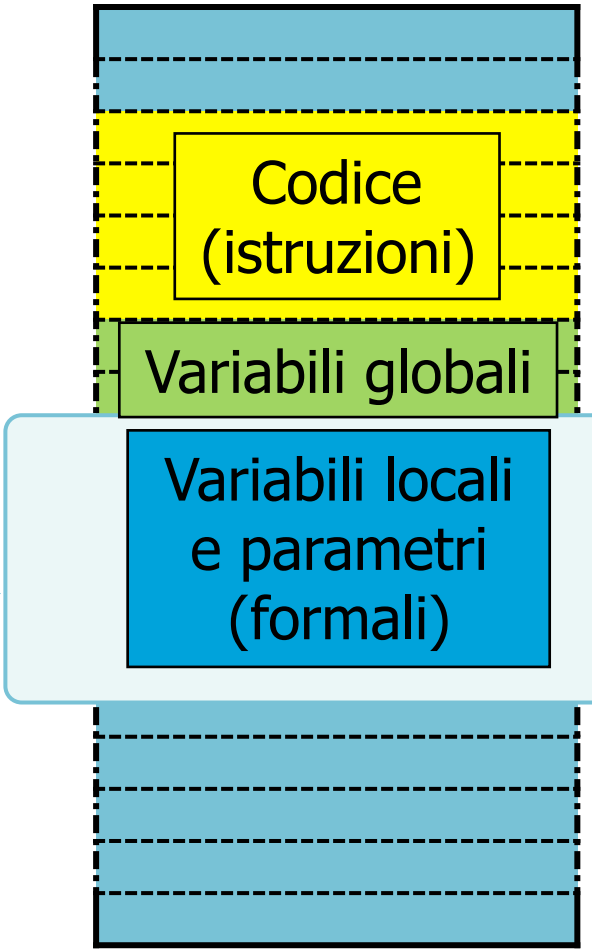
```
#define MAX 100
#define MAXR 20
struct stud {     };
...
struc
int m
  cha
  FILE *fp;
  ...
}
void ordinaStud(
  struct stud el[],int n){
  int i, j, max;
  ...
};
```

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



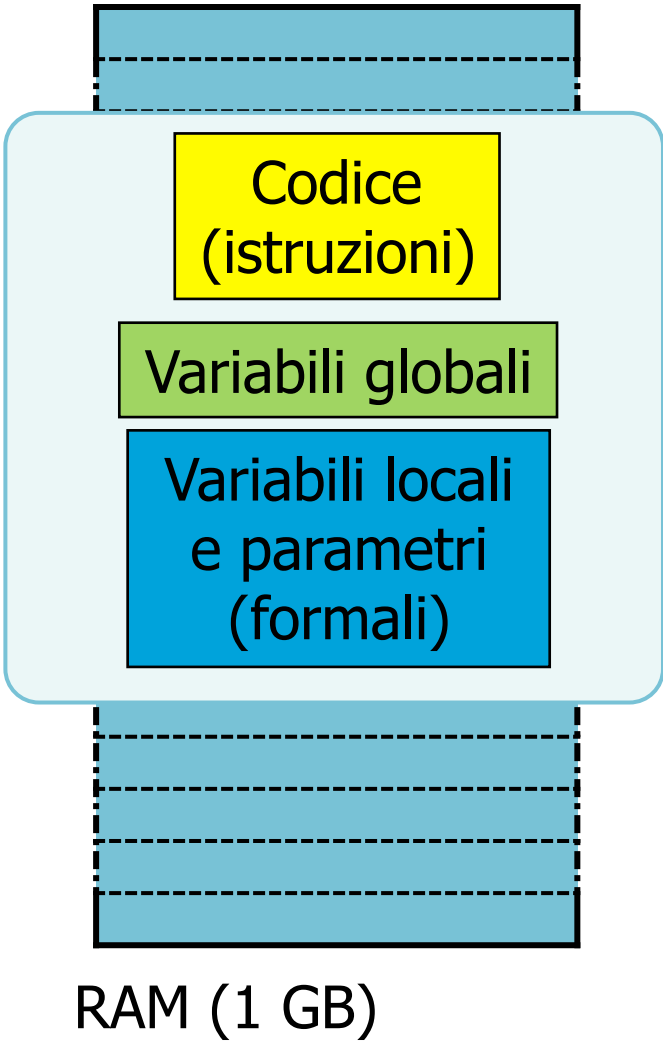
```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
   char nomefile[MAXR];
   FIL
   ...
}
void
   str
   int
   ...
};
```

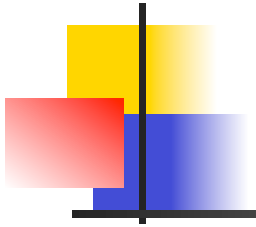
- in memoria (virtualmente) durante l'esecuzione della relativa funzione: allocate e de-allocate automaticamente
- stack frame nello stack



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

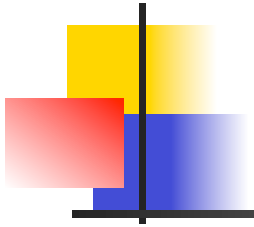
la quantità di memoria da allocare è determinata (IMPLCITAMENTE) dal programmatore:
- istruzioni
- tipo e numero delle variabili
- dimensione dei vettori



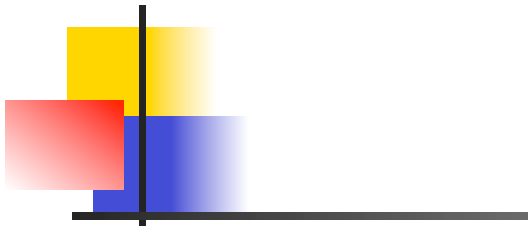


Regole di allocazione automatica:

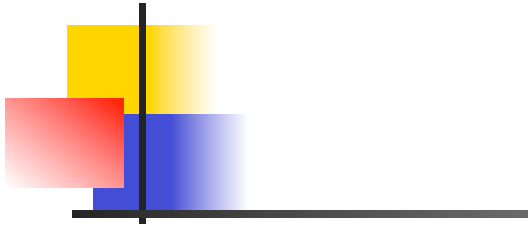
- dimensioni:
  - variabili globali e locali hanno dimensione nota
  - vettori e matrici devono avere dimensione calcolabile
  - i vettori come parametri formali sono puntatori



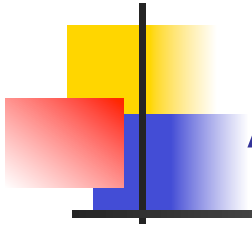
- variabili globali:
  - allocate all'avvio del programma
  - restano in vita per tutto il programma
  - ricordano I valori assegnati da funzioni
  - l'attributo `static` limita la loro visibilità al file in cui compaiono



- 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

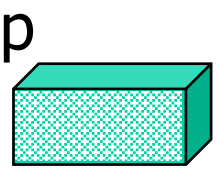


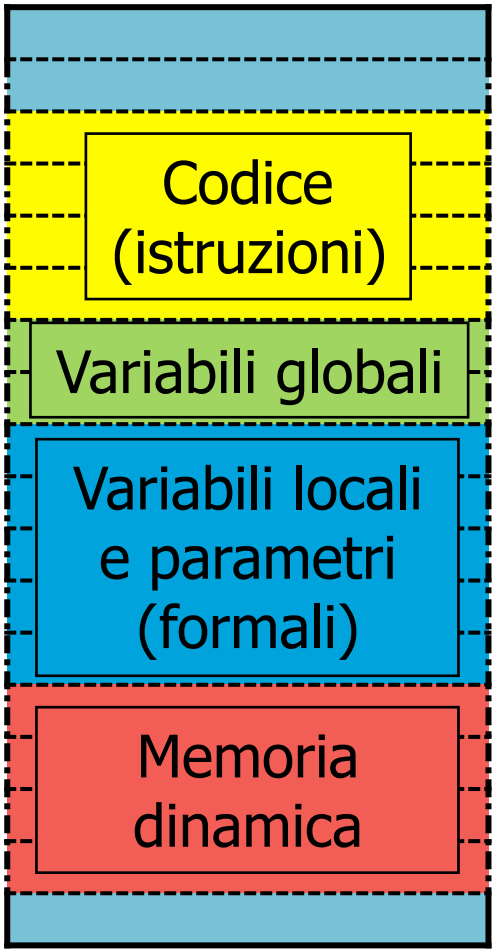
Allocazione/rilascio espliciti

- 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

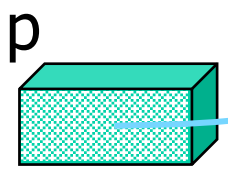
```
int main(void){
   int *p = malloc(...);
   ...
   /*
    p usato come vettore
   */
   free(p);
}
```

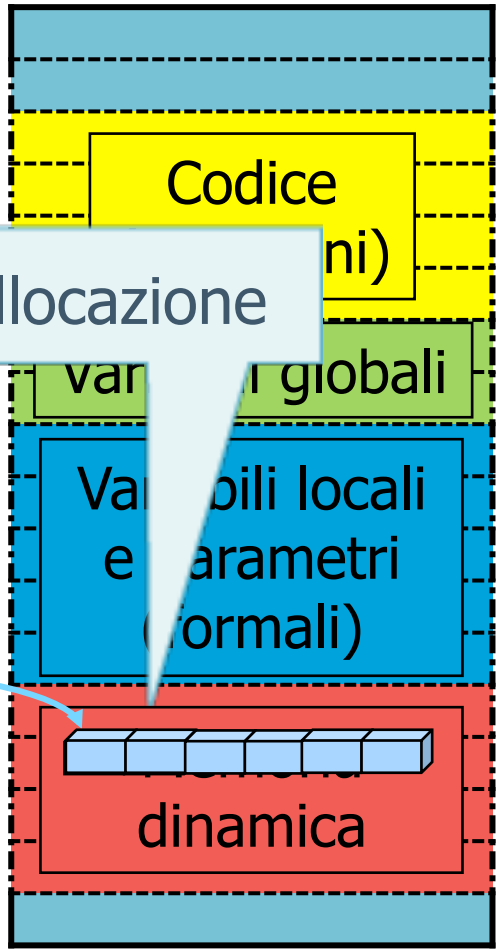




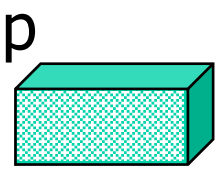
RAM (1 GB)

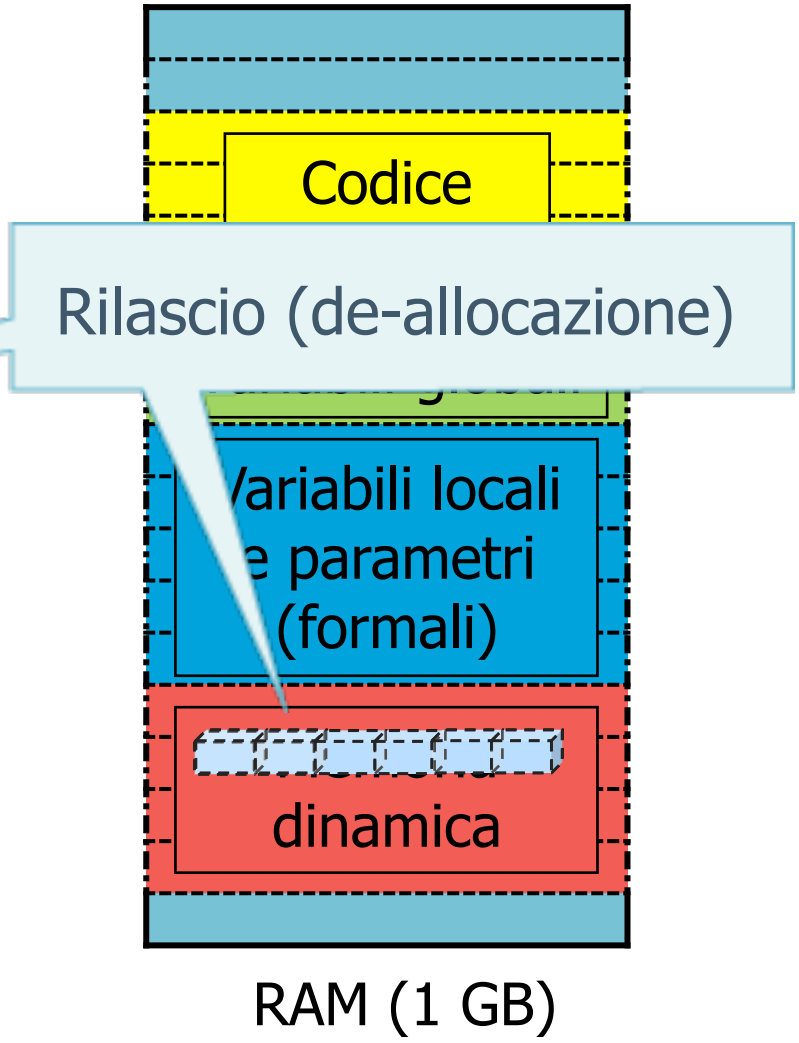
```
int main(void){
  int *p = malloc(...);
  ...
  /*
    p usato come vettore
  */
  free(p);
}
```

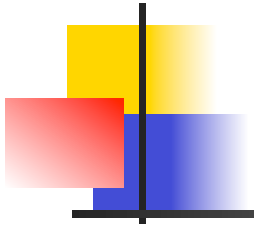




```
int main(void){
  int *p = malloc(...);
  ...
  /*
   p usato come vettore
  */
  free(p);
}
```

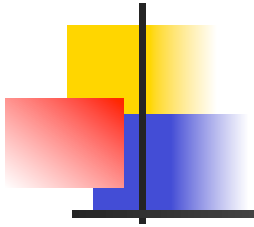






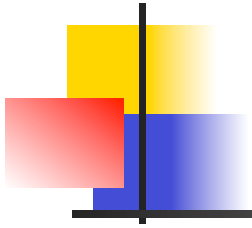
Tipologie di creazione/utilizzo di strutture dati:

- di dimensione fissa determinata in fase di esecuzione
- di dimensione modificabile (aumentabile o diminuibile) mediante riallocazione
- «contenitore» allocate a pezzi (aggiunta o eliminazione di elementi).



Fasi di una struttura dati dinamica:

- creazione (allocazione) esplicita
- utilizzo con possibilità di:
  - riallocazione
  - Inserimenti
  - cancellazioni
- distruzione (deallocazione) 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

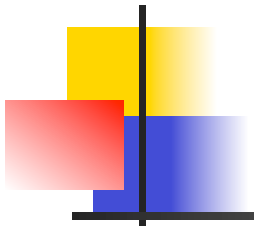
La funzione `malloc`

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

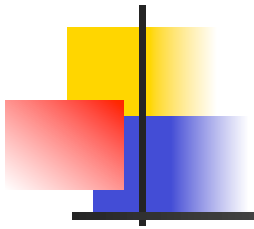
```
void* malloc (size_t size);
```

  - `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 è stato possibile allocare la memoria)
    - è tipizzato a `void*` (puntatore a qualsiasi tipo)

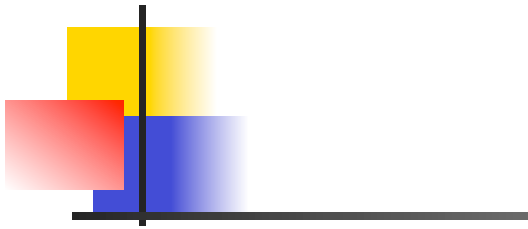
`size_t` è un intero senza segno (si può usare come parametro attuale un `int`)



- Al dato allocato si accede **unicamente** tramite puntatore
- La dimensione del dato è responsabilità del programmatore
- Il puntatore ritornato è **opaco**, tocca al programmatore passare al tipo desiderato mediante assegnazione a opportuna variabile puntatore



- Per usare `malloc` occorre includere `<stdlib.h>`
- Solitamente si ricorre all'operatore `sizeof` per determinare la dimensione (in byte) di un dato:
  - `sizeof`(<tipo>)
  - `sizeof` <espressione riconducibile a tipo>



Forma generale di chiamata a `malloc`:

- 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>);
```

Esempio: 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`

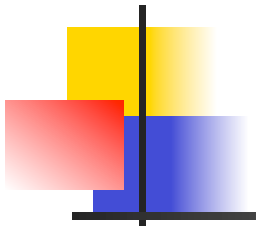
Esempio: data una `struct stud` ed una variabile `s` puntatore a `struct stud`

```
#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)
```



Allocazione di una variabile `s` puntatore a `struct stud:`

- Con cast implicito:

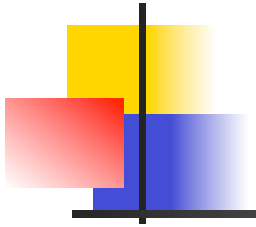
```
s =  malloc (sizeof(struct stud));
```

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

- Con cast esplicito:

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

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



Allocazione di un vettore `v` di `n` dati di tipo `struct stud:`

- Con cast implicito:

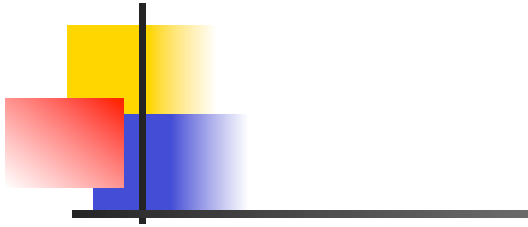
```
v =  malloc (n*sizeof(struct stud));
```

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

- Con cast esplicito:

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

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



Errori comuni:

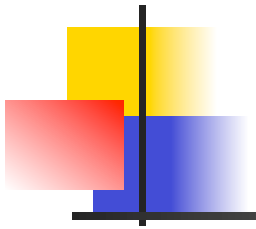
- dimensione richiesta inferiore alla necessaria:
  - uso del tipo errato in `sizeof`

```
double *pd;
struct stud *ps, *v;
int n;
...
pd = malloc (sizeof (int));
```

`sizeof (double)`

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

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



```
sizeof(struct  stud)
```

- uso del tipo puntatore a dato osto del tipo del dato puntato

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

- dimensione n omessa o errata

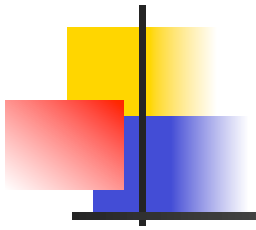
```
v = malloc (sizeof *v);
```

```
n * sizeof (*v)
```

- omissione di `sizeof`

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

```
sizeof(struct  stud)
```



```
struct stud *ps;
```

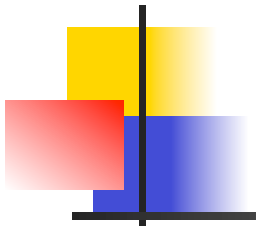
Errori comuni:

- assegnazione di puntatore incompatibile con il dato:

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

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

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



Errori comuni:

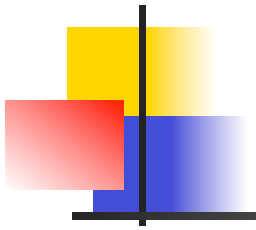
- assegnazione di puntato ompatibile con il dato:

pi e sizeof(struct stud) non sono compatibili

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

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

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

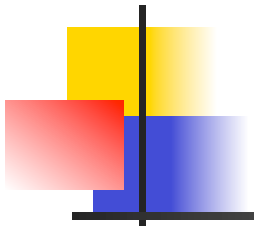


pi e sizeof(*ps) non sono compatibili

Errori comuni:

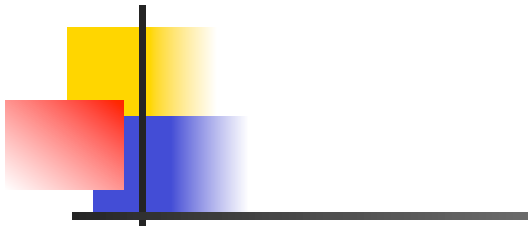
- assegnazione di puntatore incompatibile con il dato:

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



Conseguenze degli errori sui puntatori:

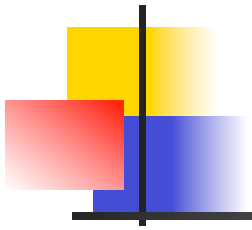
- crash del programma per accesso a indirizzo non ammesso (esito auspicabile)
- accesso ad indirizzo legale, ma al di fuori della struttura dati allocata (errore subdolo) .



Memoria dinamica insufficiente:

- `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

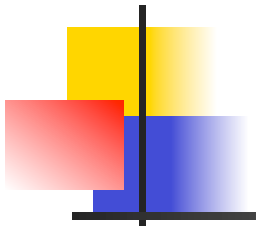
- Equivale a:

  ```
  malloc(n*size);
  ```

  con memoria ritornata azzerata

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

La calloc ha costo O(n).

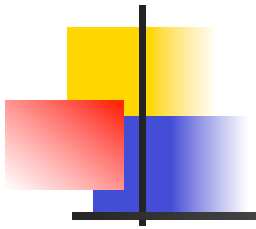


La funzione `free`

- Tutta la memoria allocata dinamicamente com `malloc/calloc` viene restituita tramite la funzione `free (<stdlib.h>)`

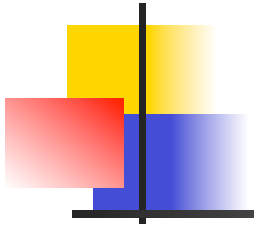
```
void free (void* p);
```

  - p punta alla memoria (precedentemente allocata) da liberare
- Viene di solito chiamata quando è terminato il lavoro sulla struttura dinamica, affinchè la memoria possa essere riutilizzata



- Uso di `free` consigliato, ma non obbligatorio:
  - al termine dell'esecuzione di un programma la memoria viene comunque liberata
  - opportuno per evitare memory leak:
    - la mancata deallocazione di una porzione di memoria
    - e il riutilizzo del puntatore ad essa per un nuovo dato allocato

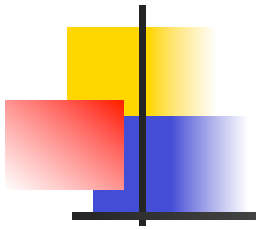
  comportano che sia irraggiungibile ma allocata la porzione precedente.



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



La funzione `realloc`

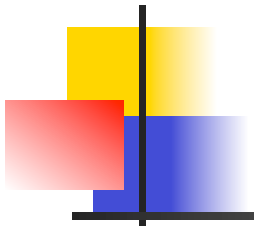
- In C la dimensione della memoria allocata può essere modificata aggiungendo o togliendo una porzione in fondo tramite `realloc`:

```
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

ATTENZIONE: la `realloc` nasconde un costo lineare

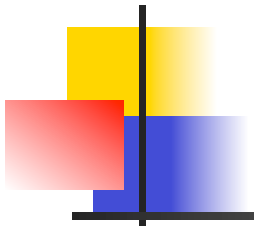
- La riduzione della dimen è sempre possibile
- L'aumento della dimens può:
  - essere impossibile (nc e memoria extra disponibile) e si ritorn ULL
  - essere possibile: esis memoria disponibile contigua al blocco gi allocato (espandibile). Si ritorna il puntatore p nvariato
  - essere possibile, m a altrove:
    - si ricopia con costo lineare nella dimensione (O(n)) il contenuto della vecchia porzione di memoria nella nuova
    - si ritorna un puntatore p aggiornato.



- Uso tipico:

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

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



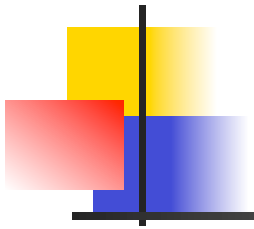
Implementazione ad alto livello della `realloc`:

```
void* realloc (void* p, size_t size) {
  size_t oldSize = cercaDimensione (p, …);
  if (/*si può espandere o ridurre*/) {
    cambiaDimensione (p, …);
    return p;
  }
  else {
    void *newp = malloc(size);
    copiaMemoria(newp,p,min(size,oldSize));
    free(p);
    return newp;
  }
}
```

Implementazione ad alto livello della `realloc`:

```
void* realloc (void* p, size_t size) {
  size_t oldSize = cercaDimensione (p, …);
  if (                                    ) {
    ca
    re
  }
  else {
    void *newp = malloc(size);
    copiaMemoria(newp,p,min(size,oldSize));
    free(p);
    return newp;
  }
}
```

copiaMemoria ha complessità O(min(size,oldSize))

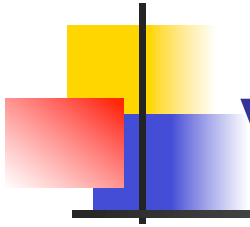


Strutture dati dinamiche

La dimensione delle strutture dati dinamiche:

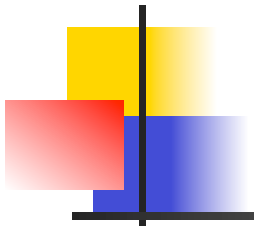
- è nota solo in fase di esecuzione
- può variare nel tempo.

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

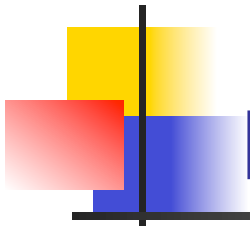


Vettori dinamici

- 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.

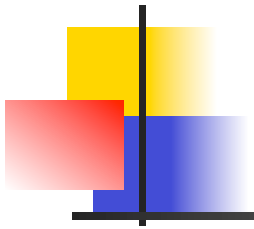


- Soluzione:
  - uso di puntatore, sfruttando la dualità puntatore-vettore, con entrambi le notazioni
  - allocazione mediante `malloc/calloc`
  - rilascio mediante `free`
- Il resto è identico al vettore sovradimensionato in modo statico.



Esempio

- Acquisire da tastiera una serie 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



- 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: la prima per calcolare il numero di dati, la seconda per memorizzarli
  - uso di `realloc`, tenendo presente il suo costo lineare nascosto:

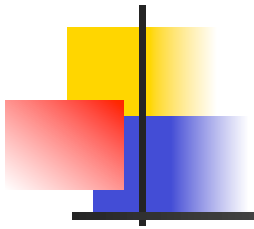
allocazione vettore dinamico

controllo

InvertiOrdine.c

```
#include <    o.h>
#include <    lib.h>
int main()
  float *v   int N, i;
  printf("N? "); scanf("%d",&N);
  v = 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]);
  }
```

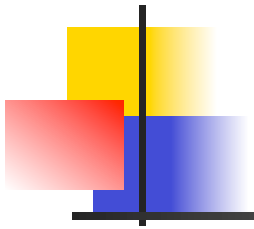
input



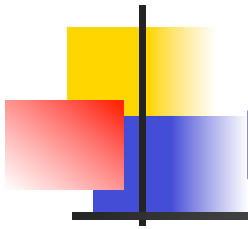
output

```
   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;
};
```

liberazione memoria dinamica

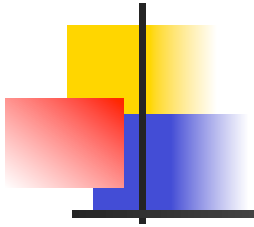


- ATTENZIONE: bisogna conoscere il numero di dati per creare e usare il vettore dinamico!
- Se il numero di dati (ignoto) fosse segnalato da un terminatore (es. input del valore 0):
  - 2 letture da file: la prima per calcolare il numero di dati, la seconda per memorizzarli
  - uso di `realloc`, tenendo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

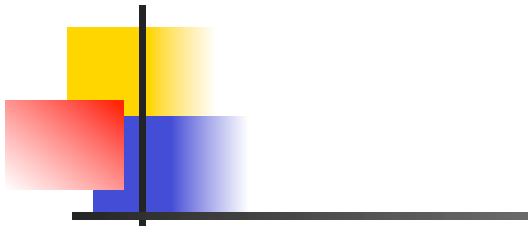


Esempio

- 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 numeri non è nota al programmatore, né sovradimensionabile, ~~ma è acquisita come primo dato da tastiera~~



- Acquisire da tastiera una serie 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.



Il vettore viene ri-allocato:

Soluzione A:

- vettore dinamico di dimensione iniziale N=1
- riallocazione ad ogni nuovo dato con N incrementato di 1
- $O(N^2)$

SCONSIGLIATA !

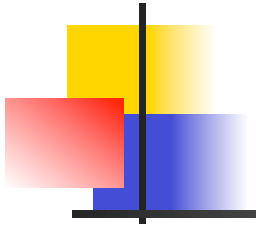
allocazione iniziale

Omessi i controlli di errore

```
...
float *v, d
int N=1, i=0;

v = malloc(N*(sizeof (float)));
printf("Inserisci elementi\n");
printf("Elemento 0: ");
while (scanf("%f", &d)>0) {
  if (i==N) {
    N = N+1; v = realloc(v,N*sizeof(float));
  }
  v[i++] = d;
  printf("Elemento %d: ", i) ;
}
```

input

riallocazione



Il vettore viene ri-allocato:

Soluzione B:

- vettore dinamico di dimensione iniziale `N`=1
- controllo se vettore pieno
- riallocazione se pieno con `N` raddoppiato (sovradimensionamento)
- O(NlogN)

CONSIGLIATA !

allocazione iniziale

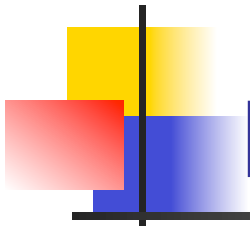
Omessi i controlli di errore

input

controllo se pieno

riallocazione con raddoppio

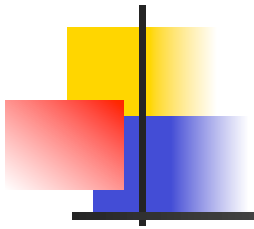
```
float *v, d;
int N, MAXN=1, i=0;
v = malloc (MAXN*(sizeof (float)));
printf("Inserisci elementi\n");
printf("Elemento 0: ");
while (scanf("%f", &d)>0) {
  if (i==MAXN) {
    MAXN=2*MAXN;
    v =realloc(v,MAXN*sizeof(float));
  }
  v[i++] = d;
  printf("Elemento %d: ", i) ;
}
N = i;
```



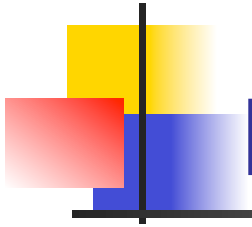
Matrici dinamiche

Due possibilità:

- Soluzione 1 (meno flessibile):
  - vettore dinamico di `nr x nc` elementi
  - organizzazione manuale di righe e colonne su vettore: l'elemento `(i,j)` si trova in posizione `nc*i + j`.



- Soluzione 2:
  - vettore dinamico di `nr` puntatori a righe (o dualmente vettore di `nc` puntatori a colonne)
  - 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:

matriceTraspostaVettDin.c

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

printf("nr nc: "); scanf("%d%d", &nr, &nc);
v = 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

controllo di errore

Con vettore dinamico:

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

printf("nr nc: "); scanf("%d%d", &nr, &nc);
v = 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]);
}
```

float

input

output

```
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);
...
```

liberazione memoria dinamica

Con matrice dinamica

vettore di vettori di `float`

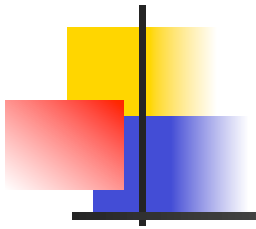
matriceTraspostaMatDin.c

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

printf("nr nc: ");
scanf("%d%d", &nr, &nc);

v = malloc (nr*sizeof (float *));

if (v==NULL)
  exit(1);
```

allocazione di vettore di `nr` puntatori a riga

vettore di `float`

controllo di errore



allocazione di vettore di nc dati `float`

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

controllo di errore

input

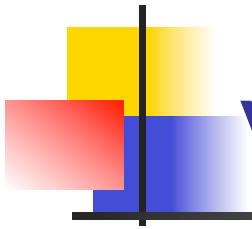
output

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

liberazione memoria dinamica

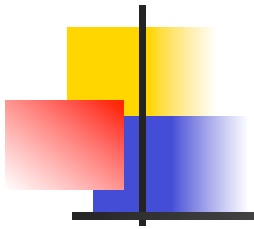
singola riga

vettore delle righe

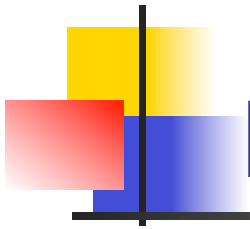


Vettori e matrici creati da funzioni

- Un vettore o matrice dinamici sono accessibili a partire da un puntatore
- Esistono fintanto che è in essere la funzione dove sono dichiarati e sono visibili in essa
- Può essere necessario che siano accessibili da altre funzioni

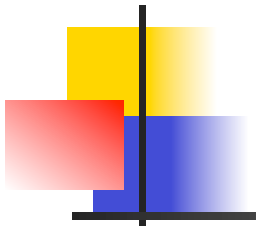


- Per renderli accessibili al programma chiamante:
  - si dichiara il puntatore come variabile globale (sconsigliato!)
  - si inserisce il puntatore tra i parametri della funzione e lo si modifica (passaggio by reference, quindi, in C “by value” di un puntatore a puntatore)
  - si ritorna il puntatore (modificato) come valore di ritorno della funzione.



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:
  - `malloc2dP` dove il puntatore alla matrice è restituito tra i parametri della funzione
  - `malloc2dR` dove il puntatore alla matrice è restituito come valore di ritorno della funzione



Puntatore come valore di ritorno della funzione

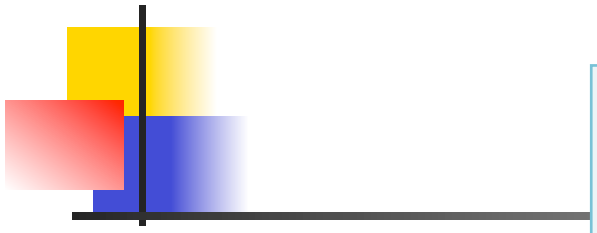
funzione di tipo `void`

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

vettore di puntatori ad elementi di tipo `Item`

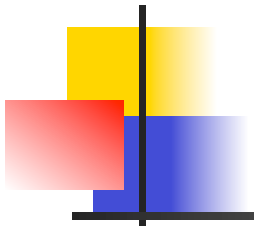
mp: puntatore (triplo) ad un puntatore ad un vettore di righe

```
void malloc2dP(Item ***mp, 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));
   *mp = m;
}
```



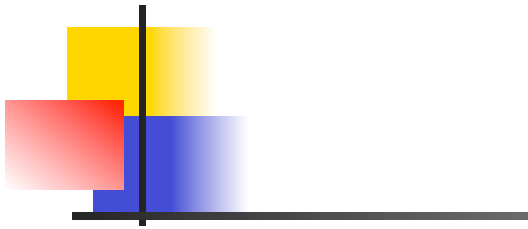
punta a variabile del programma chiamante (puntatore doppio) in cui occorre trasferire il risultato

```
void malloc2dP(Item ***mp, 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));
   *mp = m;
}
```



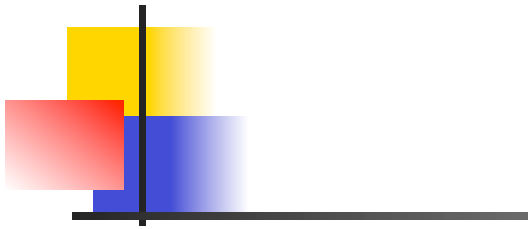
```
void malloc2dP(Item ***mp, int nr, int nc) {
   Item **m;
   int i;
   m = mall    (nr*sizeof(Item *));
   for (i=0;     r; i++)
     m[i] = m          (nc*sizeof(Item));
   *mp = m;
}
```

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



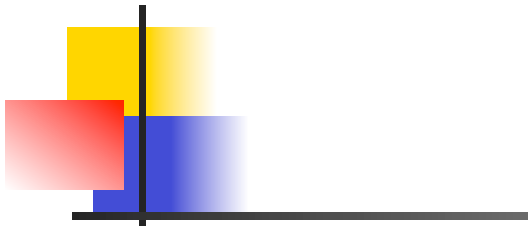
```
void malloc2dP(Item ***mp, int nr, int nc) {
   Item **m;
   int i;
   m = malloc (nr*sizeof(Item *));
   for (i=0; i<nr; i++)
     m[i] = malloc (nc*sizeof(Item));
   *mp = m;
}
```

creazione del vettore di puntatori



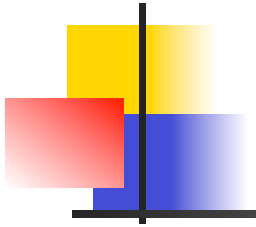
```
void malloc2dP(Item ***mp, int nr, int nc) {
   Item **m;
   int i;
   m = malloc (nr*sizeof(Item *));
   for (i=0; i<nr; i++)
     m[i] = malloc (nc*sizeof(Item));
   *mp = m;
}
```

creazione della singola riga



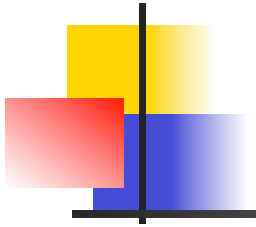
```
void malloc2dP(Item ***mp, int nr, int nc) {
   Item **m;
   int i;
   m = malloc (nr*sizeof(Item *));
   for (i=0; i<nr; i++)
     m[i] = malloc (nc*sizeof(Item));
   *mp = m;
}
```

m (NON ritornato come risultato) assegnato alla variabile (del programma chiamante) puntata da mp



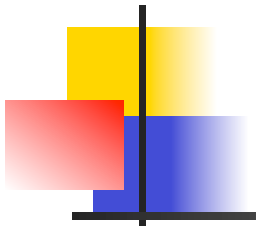
```
void malloc2dP(Item ***mp, int nr, int nc) {
   int i;
   *mp = malloc (nr*sizeof (Item *));
   for (i=0; i<nr; i++) {
      (*mp)[i] = malloc (nc*sizeof (Item));
   }
}
```

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



funzione di tipo puntatore a vettore di `Item`

Puntatore come valore di ritorno della funzione

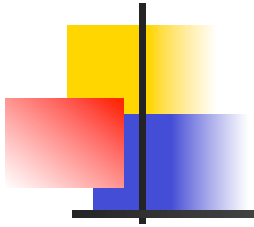
```
typedef ... Item;
...
Item **malloc2dR(int nr, int nc);
void free2d(Item **m, int nr);
...
Item **matr;
int nr, nc;
...
matr = malloc2dR(nr, nc);
...
free2d(matr,nr);
```

vettore di `Item`

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

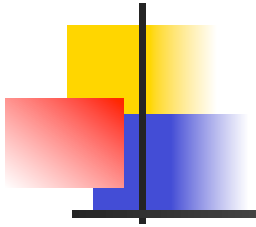
```
Item **malloc2d(int nr, int nc) {
  Item **m;
  int i;
  m = malloc (nr*sizeof (Item *));
  for (i=0; i<nr; i++) {
    m[i] = malloc (nc*sizeof (Item));
  }
  return m;
}
```

m ritornato come risultato



- `free` unica, i parametri vengono solo ricevuti
- Occorre liberare le righe una alla volta
- Infine liberare il vettore di puntatori

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



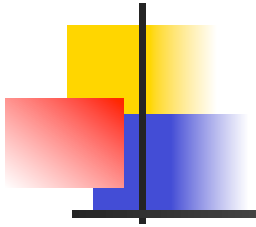
Vettori a dimensione variabile

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

vettori a lunghezza variabile

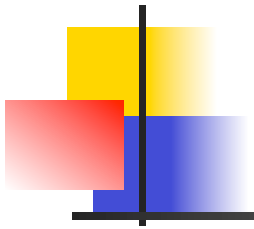
(variable length arrays)

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



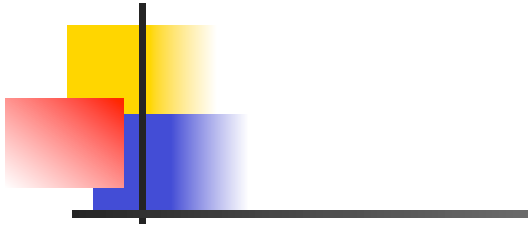
L’uso di vettori a lunghezza variabile è scoraggiato in quanto:

- 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
  - 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]);
}
```



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.