Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Layout della memoria di un programma C (reminder)

Un programma C in esecuzione utilizza 4 regioni di memoria logicamente distinte:

1. regione dedicata a contenere il codice in esecuzione (**text/code area**)
2. regione dedicata a variabili `extern`, `global` e `static`. Si divide a sua volta in:
   - **initialized data segment**: variabili `extern`, `global` e `static` i cui valori **sono inizializzati** in fase di dichiarazione
   - **uninitialized data segment** (*bss, block started by symbol*): variabili `extern`, `global` e `static` non inizializzate in fase di dichiarazione. Il kernel inizializza queste variabili a 0 (o `NULL` nel caso di puntatori) prima che il programma entri in esecuzione
3. l'**execution stack**, o *call stack*, o spesso chiamato comunemente *stack* (anche se fonte di ambiguità), contenete gli indirizzi di ritorno delle funzioni invocanti, argomenti e **variabili locali**. Il termine *stack* si riferisce alla *politica di accesso* (LIFO, Last In First Out).
4. l' **heap**, regione di spazio libero utilizzata per l'allocazione dinamica (NB: nessuna relazione con l'omonima struttura dati)

Il C supporta due tipi di allocazione di memoria:

- *__static allocation__*: destinata alle variabili `global` o `static`. Per Ogni variabile viene definito un blocco di spazio di dimensione fissata. Lo spazio viene allocato* all'avvio del programma*.
- *__automatic allocation__*: destinata alle variabili `automatic`, come gli argomenti di funzione o le varibili locali. Tali variabili vengono allocate quando il programma entra materialmente nel blocco in cui tali variabili sono definite, e deallocate quando termina tale blocco.

Una terza tipologia di allocazione, la *__dinamic allocation__*, *non è supportata "in maniera diretta" dal C, ma è disponibile attraverso funzioni di libreria apposite. L'allocazione*

dinamica è necessaria quando non si conosce a priori* quanta memoria è necessaria.

Tutto ciò che viene allocato dinamicamente finisce nell'heap.

Principali funzioni C per l'allocazione dinamica

- `malloc(...)` -> diminutivo di `memory allocation`
- `calloc(...)` -> diminutivo di `contigous allocation`
- `realloc(...)` -> rialloca memoria precedentemente allocata
- `free(...)` -> libera la memoria *allocata attraverso* `malloc(...)` / `calloc(...)` / `realloc(...)`

`malloc(...)`

prototipo:

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

input:

- `size`: dimensione complessiva del blocco da allocare

output:

- indirizzo di start della memoria allocata se l'operazione è andata a buon fine, `NULL` altrimenti.

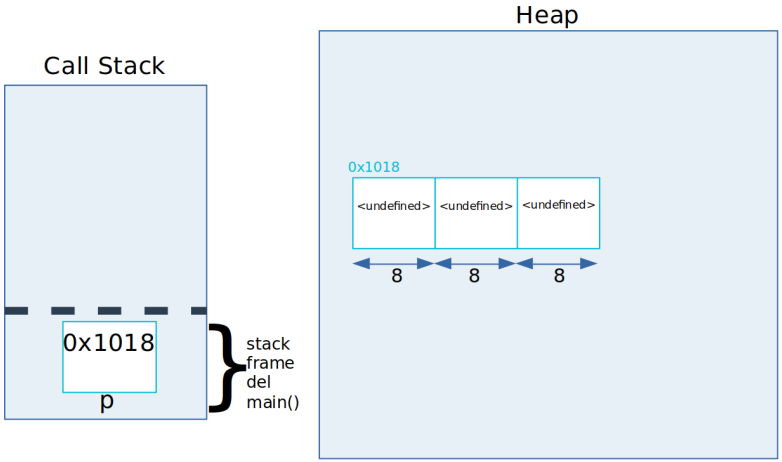
Casi particolari:

- If the size of the space requested is 0, the behavior is implementation-defined.
- If the space cannot be allocated, a null pointer shall be returned

Esempio:

```c
int main()
{
    void* p = malloc(3 * sizeof(int)); /* sizeof(int) restituisce il
                                          numero di byte che compongono
                                          un object di tipo int
                                       */
    return 0;
}
```

`p` quindi punterà ad un'area di memoria sufficientemente grande per contenere 3 `int` (quindi di 34 Byte se sto in un sistema in cui ogni `int` è rappresentato da 8 Byte).



`calloc(...)`

prototipo:

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

input:

- `size` : *dimensione* di ogni object da allocare
- `nitems` : *quanti* object allocare

output:

- indirizzo di start della memoria allocata se l'operazione è andata a buon fine, `NULL` altrimenti. Tale memoria è inizializzata di default a 0.

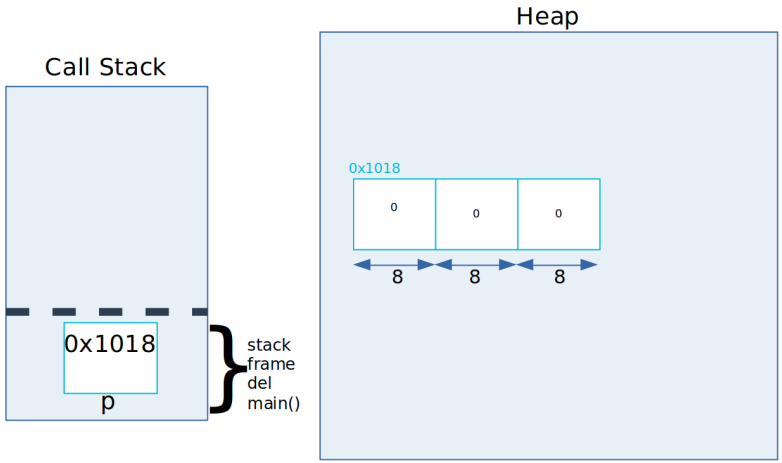
Casi particolari:

- If the size of the space requested is 0, the behavior is implementation-defined.
- If the space cannot be allocated, a null pointer shall be returned

Esempio:

```
int main()
{
    void* p = calloc(3, sizeof(int)); /* sizeof(int) restituisce
il
                                         numero di byte che
compongono
                                         un object di tipo int
                                      */
    return 0;
}
```


`p` quindi punterà ad un'area di memoria sufficientemente grande per contenere 3 `int` (quindi di 34 Byte se sto in un sistema in cui ogni `int` è rappresentato da 8 Byte), tutti inizializzati a 0.



Nota:

Sia `malloc(...)` che `calloc(...)` allocano memoria *__contigua__* (almeno a livello *logico*). Le differenze sostanziali sono due:

1. calloc offre un'interfaccia diversa (2 parametri invece di 1)
2. calloc inizializza l'area allocata (malloc no).

Ad ogni modo, non c'è garanzia che la memoria **fisica** allocata (con cui noi non interagiamo) sia davvero contigua, ma dato che tutti gli accessi vengono effettuati attraverso indirizzi virtuali **contigui** forniti dal Sistema Operativo (che sono ciò con cui noi interagiamo), poco importa.

```
void* calloc_logic_equivalent(size_t nmemb, size_t size)
{
  const size_t bytes = nmemb * size;
  void *p            = malloc(bytes);
  if(p != NULL)
   memset(p, 0, bytes);
  return p;
}
```

Utilizzo dell'indirizzo di ritorno di `malloc/calloc/realloc`

Indirizzo di ritorno di tipo `void` $\Rightarrow$ può essere inserito in un puntatore di qualsiasi tipo.

```
int* p1    = malloc(sizeof(int));

char* p1   = malloc(sizeof(char));

double* p3 = malloc(sizeof(double));
```

In generale, il tipo del puntatore specifica *come* utilizzare la memoria allocata. Esempio:

```
int* p1    = malloc(sizeof(int));
*p1        = 42; // il tipo di p1 permette al 42 di essere
inserito in memoria come int
```

In teoria, sono "valide" anche le seguenti...

```
char* c    = malloc(sizeof(int));

int*  i    = malloc(sizeof(char));
```

queste ultime (la seconda specialmente) sono molto pericolose!!! Ad esempio, si immagini cosa succederebbe se facessi

```
*i = 1042;
```

....

`free(...)`

Prototipo:

```
void free( void* ptr );
```

La funzione `free(...)` dealloca lo spazio allocato attraverso `malloc(...)` / `calloc(...)` / `realloc(...)`.

- Il valore del parametro `ptr` **deve** essere un indirizzo restituito da una di queste funzioni. In caso contrario, il comportamento è indefinito.
- Se `ptr` contiene `NULL` la funzione non fa nulla.
- Se l'area di memoria puntata da `ptr` è stata già deallocata in precedenza (ad esempio da una precedente invocazione di `free(...)`) il comportamento è indefinito
- Se dopo una invocazione di `free(ptr)` si prova ad accedere a `*ptr` il comportamento è indefinito

REGOLA: TUTTO CIO' CHE E' ALLOCATO DINAMICAMENTE ATTRAVERSO `malloc(...)` / `calloc(...)` / `realloc(...)` DEVE ESSERE DEALLOCATO CON `free(...)` PRIMA DEL TERMINE DEL PROGRAMMA!

`realloc(...)`

Prototipo:

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


The `realloc(...)` function shall change the size of the memory object pointed to by `ptr` to the size specified by size. The contents of the object shall remain unchanged up to the lesser of the new and old sizes.

Se `size` è maggiore della dimensione allocata puntata da `ptr`, allora `realloc(ptr, size);` è logicamente equivalente alla seguente sequenza di operazioni:

1. alloca un'area di memoria di dimensione `size`. Se non c'è abbastanza spazio, restituisce `NULL`
2. copia il contenuto dell'area di memoria puntata da `ptr` nella nuova zona
3. invoca `free(ptr);`
4. restituisce l'indirizzo della nuova zona di memoria

Casi particolari:

- se `size=0`, il valore di ritorno dipende dall'implementazione (EVITARE!!!)
- se `size` è minore della vecchia dimensione, allora solo i primi `size` elementi saranno copiati
- se `size` è maggiore della vecchia dimensione, gli elementi in eccesso avranno valore non definito
- se `ptr==NULL`, `realloc(ptr, size)` ≡ `malloc(size)`
- se `ptr` contiene un indirizzo non restituito da `malloc(...)/calloc(...)` il comportamento è indefinito
- se `ptr` contiene un indirizzo precedentemente deallocato, il comportamento è indefinito

In [29]:

```
#include <stdio.h>
int main()
{
    int* p = calloc(3, sizeof(int));

    printf("p: %p\n",p);
    *p     = 1;
    *(p+1) = 5;
    *(p+2) = 7;
    printf("*p: %d, *(p+1): %d, *(p+2): %d\n", *p, *(p+1), *(p+2));

    p = realloc(p, 3*sizeof(int));
    printf("p: %p\n",p);
    printf("*p: %d, *(p+1): %d, *(p+2): %d\n", *p, *(p+1), *(p+2));

    p = realloc(p, 15*sizeof(int));
    printf("p: %p\n",p);
    printf("*p: %d, *(p+1): %d, *(p+2): %d, *(p+5): %d, *(p+10): %d\n", *

    p = realloc(p, 1*sizeof(int));
    printf("p: %p\n",p);
    printf("*p: %d, *(p+1): %d, *(p+2): %d, *(p+5): %d, *(p+10): %d\n", *


    free(p);
```

```
    return 0;
}
```

```
p: 0x555e8f370880
*p: 1, *(p+1): 5, *(p+2): 7
p: 0x555e8f370880
*p: 1, *(p+1): 5, *(p+2): 7
p: 0x555e8f3708f0
*p: 1, *(p+1): 5, *(p+2): 7, *(p+5): 0, *(p+10): 0
p: 0x555e8f3708f0
*p: 1, *(p+1): 5, *(p+2): 7, *(p+5): 0, *(p+10): -1892220912
```

Allocazione dinamica di un array

- Possono essere allocati con `calloc(...)` / `malloc(...)`
- l'accesso agli elementi può essere fatto attraverso l'aritmetica dei puntatori, oppure attraverso l'operatore `[]`

In [33]:
```
#include <stdio.h>
void stampa_vett(int vett[], int n)
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%d,", vett[i]);
    printf("\b)\n");
}

void acquisisci_vett(int vett[], int n)
{
    for(int i=0; i<n; i++)
        scanf("%d", &vett[i]);
}


int main()
{
    int n;
    printf("quanti elementi vuoi inserire?\n");
    scanf("%d", &n);

    int* v = calloc(n, sizeof(int));

    printf("inserisci gli elementi:\n");
    acquisisci_vett(v,n);
    printf("gli elementi inseriti sono: ");
    stampa_vett(v,n);

    free(v); // mai dimenticarla
    return 0;
}
```

```
quanti elementi vuoi inserire?

inserisci gli elementi:

gli elementi inseriti sono: (1,2,3)
```

In [36]:
```
#include <stdio.h>
void stampa_vett(int vett[], int n)
```



```
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%d,", vett[i]);
    printf("\b)\n");
}

int main()
{

    int n  = 0;

    int* v = NULL;

    int inserito = -1;
    printf("inserisci gli elementi (0 per terminare):\n");

    while(inserito != 0)
    {
        scanf("%d", &inserito);
        if(inserito != 0)
        {
            n++;
            if (n == 1)
                v = calloc(n, sizeof(int)); // NB: è davvero necessaria??
            else
                v  = realloc(v, n*sizeof(int));
            v[n-1] = inserito;
        }
    }

    printf("gli elementi inseriti sono: ");
    stampa_vett(v,n);

    free(v); // mai dimenticarla
    return 0;
}
```

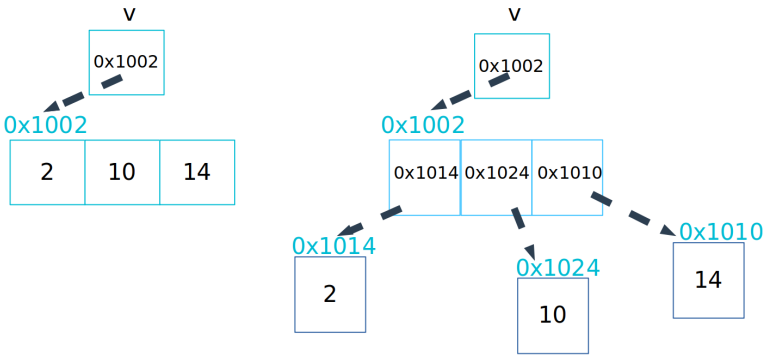
```
inserisci gli elementi (0 per terminare):

gli elementi inseriti sono: (1,2,3,4)
```

Alternativa: Posso vedere un vettore di un tipo base T come un *array* di puntatori a T

In [1]:
```
#include <stdio.h>
void stampa_vett(int* vett[], int n)
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%d,", *vett[i]);
    printf("\b)\n");
}

void acquisisci_vett(int* vett[], int n)
{
    for(int i=0; i<n; i++)
        scanf("%d", vett[i]);
}
```

```
int main()
{

    int n;
    printf("quanti elementi vuoi inserire?\n");
    scanf("%d", &n);

    int** v = calloc(n, sizeof(int*));
    for(int i = 0; i < n; i++)
        v[i] = malloc(sizeof(int));

    printf("inserisci gli elementi:\n");
    acquisisci_vett(v,n);
    printf("gli elementi inseriti sono: ");
    stampa_vett(v,n);


    for(int i = 0; i < n; i++)
        free(v[i]);
    free(v);
    return 0;
}
```

```
quanti elementi vuoi inserire?

inserisci gli elementi:

gli elementi inseriti sono: (1,2,3,4)
```



tale alternativa in generale è **SCONSIGLIATA!**

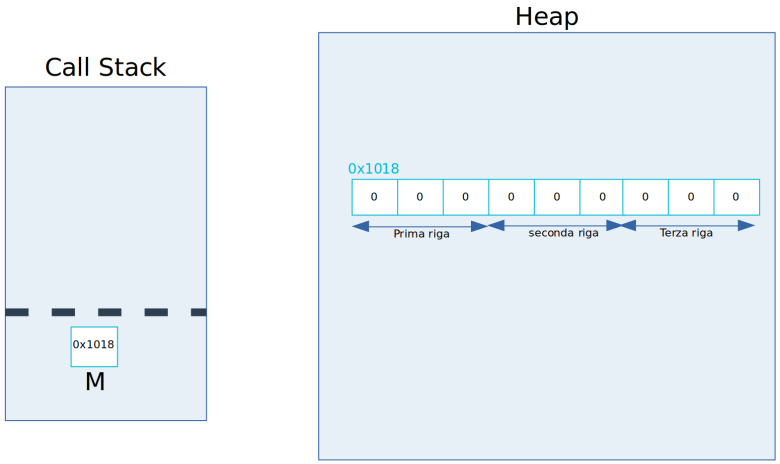
- nessun vantaggio rispetto all'utilizzo di un semplice array di elementi tipo T

Allocazione dinamica di una matrice

- Possono essere allocati con `calloc(...)` / `malloc(...)` in vari modi (almeno 3)

Metodo 1: Matrix as array

- tutte le righe di una matrice $M \in \mathbb{R}^{R \times C}$ sono viste come un unico grande array monodimensionale $\mathbf{v}_R = (m_{11}, m_{12}, \ldots, m_{1C}, m_{21}, m_{22}, \ldots, m_{2C}, \ldots m_{RC})$ con m_{ij} elemento della matrice in riga i e colonna j
- la matrice è quindi vista come un array monodimensionale $\mathbf{v}_R$, e gestito come tale



In [43]:
```
# include <stdio.h>
void stampa_mat(int vett[], int n_r, int n_c)
{
    printf("(\n");
    for(int i=0; i<n_r; i++)
    {
        for(int j=0; j<n_c; j++)
        {
            printf(" %d,", vett[i*n_c + j]);
        }
        printf("\b\n");
    }
    printf(")\n");
}

void acquisisci_mat(int vett[], int n_r, int n_c)
{
    for(int i=0; i<n_r; i++)
        for(int j=0; j<n_c; j++)
            scanf("%d", &vett[i*n_c + j]);
}


int main()
{
    int *m = NULL;
    int n_r = 0, n_c = 0;

    printf("Quante righe?\n");
    scanf("%d", &n_r);

    printf("Quante colonne?\n");
```

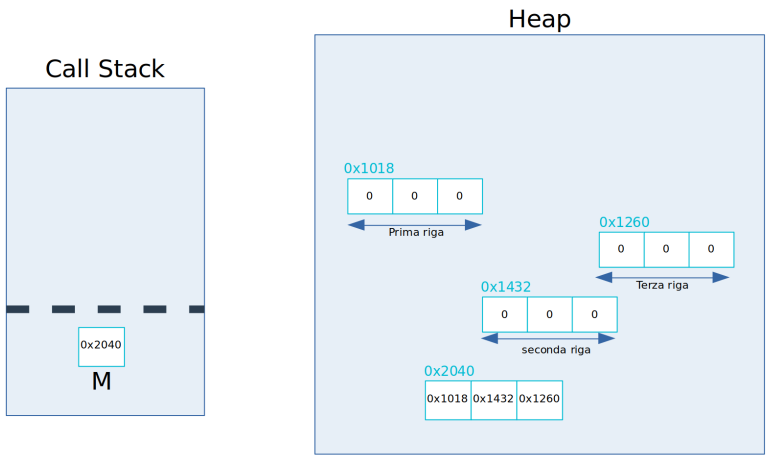
```
    scanf("%d", &n_c);

    // alloco un vettore n_r*n_c
    m = calloc(n_r*n_c, sizeof(int));

    printf("inserisci gli elementi:\n");
    acquisisci_mat(m, n_r, n_c);

    printf("gli elementi inseriti sono:\n");
    stampa_mat(m, n_r, n_c);

    free(m); // mai dimenticarla
    return 0;
}
```

```
Quante righe?

Quante colonne?

inserisci gli elementi:

gli elementi inseriti sono:
(
 1, 2
 3, 4
 5, 6
)
```

- pro: elementi (logicamente) contigui in memoria
- contro: indicizzazione con singolo operatore `[]` come se fosse un array monodimensionale (e relativa gestione degli indici di riga e colonna)

Metodo 2: Matrix as array of pointers to arrays

- ogni riga della matrice viene rappresentata con un diverso array monodimensionale
- la matrice è rappresentata come un array di *puntatori alle righe*



In [46]:

```
# include <stdio.h>
void stampa_mat(int** vett, int n_r, int n_c)
{
    printf("(\n");
    for(int i=0; i<n_r; i++)
    {
        for(int j=0; j<n_c; j++)
        {
            printf(" %d,", vett[i][j]);
        }
        printf("\b\n");
    }
    printf(")\n");
}

void acquisisci_mat(int** vett, int n_r, int n_c)
{
    for(int i=0; i<n_r; i++)
        for(int j=0; j<n_c; j++)
            scanf("%d", &vett[i][j]);
}


int main()
{
    int **m = NULL; // m è un puntatore a puntatore ad intero
    int n_r = 0, n_c = 0;

    printf("Quante righe?\n");
    scanf("%d", &n_r);

    printf("Quante colonne?\n");
    scanf("%d", &n_c);

    // alloco un vettore di n_r puntatori
    m = calloc(n_r, sizeof(int*));

    // ogni elemento del vettore puntato da m punta a sua volta
    // ad un vettore di n_c elementi
    for(int i = 0; i < n_r; i++)
        m[i] = calloc(n_c, sizeof(int));

    printf("inserisci gli elementi:\n");
    acquisisci_mat(m, n_r, n_c);

    printf("gli elementi inseriti sono:\n");
    stampa_mat(m, n_r, n_c);

    // deallocazione
    for(int i = 0; i < n_r; i++)
        free(m[i]);
    free(m);
    return 0;
}
```

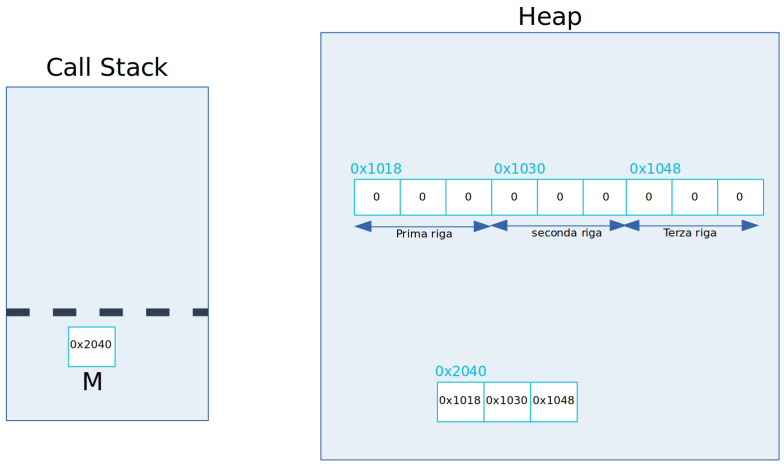
```
Quante righe?

Quante colonne?

inserisci gli elementi:
```

```
gli elementi inseriti sono:
(
 1, 2
 3, 4
 5, 6
)
```

- pro: posso indicizzare utilizzando più operatori `[]`
- contro: la memoria allocata non è contigua
- contro: necessario deallocare manualmente ogni riga

Metodo 3: Matrix as array of pointers to subarrays

- tutte le righe sono rappresentate attraverso un unico grande array monodimensionale $\mathbf{v}_R = (m_{11}, m_{12}, \ldots, m_{1n}, m_{21}, m_{22}, \ldots, m_{2n}, \ldots m_{nn})$ con m_{ij} elemento della matrice in riga i e colonna j
- la matrice è vista come un vettore di puntatori in cui ogni elemento i-esimo punterà all'elemento in $\mathbf{v}_R$ corrispondente al primo elemento della riga i-esima



In [48]:
```
# include <stdio.h>
void stampa_mat(int** vett, int n_r, int n_c)
{
    printf("(\n");
    for(int i=0; i<n_r; i++)
    {
        for(int j=0; j<n_c; j++)
        {
            printf(" %d,", vett[i][j]);
        }
        printf("\b\n");
    }
    printf(")\n");
}
```

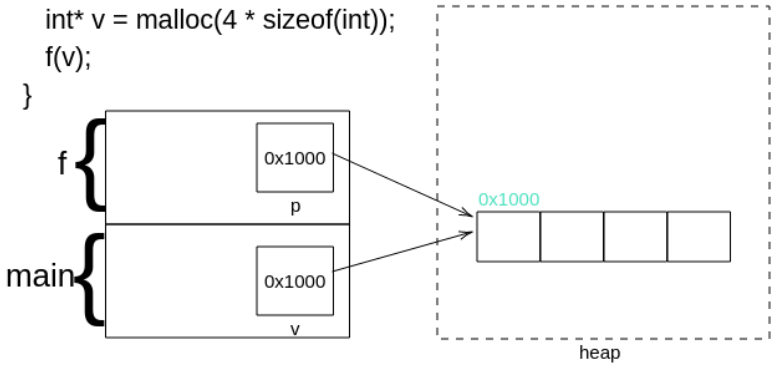


```
void acquisisci_mat(int** vett, int n_r, int n_c)
{
    for(int i=0; i<n_r; i++)
        for(int j=0; j<n_c; j++)
            scanf("%d", &vett[i][j]);
}


int main()
{
    int **m = NULL; // m è un puntatore a puntatore ad intero
    int n_r = 0, n_c = 0;

    printf("Quante righe?\n");
    scanf("%d", &n_r);

    printf("Quante colonne?\n");
    scanf("%d", &n_c);

    m    = calloc(n_r, sizeof(int*));

    m[0] = calloc(n_r*n_c, sizeof(int));

    for(int i = 1; i < n_r; i++)
        m[i] = m[0] + i*n_c;  // sfrutto l'aritmetica dei puntatori

    printf("inserisci gli elementi:\n");
    acquisisci_mat(m, n_r, n_c);

    printf("gli elementi inseriti sono:\n");
    stampa_mat(m, n_r, n_c);

    free(m[0]);
    free(m);

    return 0;
}
```

```
Quante righe?

Quante colonne?

inserisci gli elementi:

gli elementi inseriti sono:
(
 1, 2
 3, 4
 5, 6
)
```

L'importanza di poter puntare

```
void f(int* p)
{
....

}

int main()
{
  int* v = malloc(4 * sizeof(int));
  f(v);
}
```

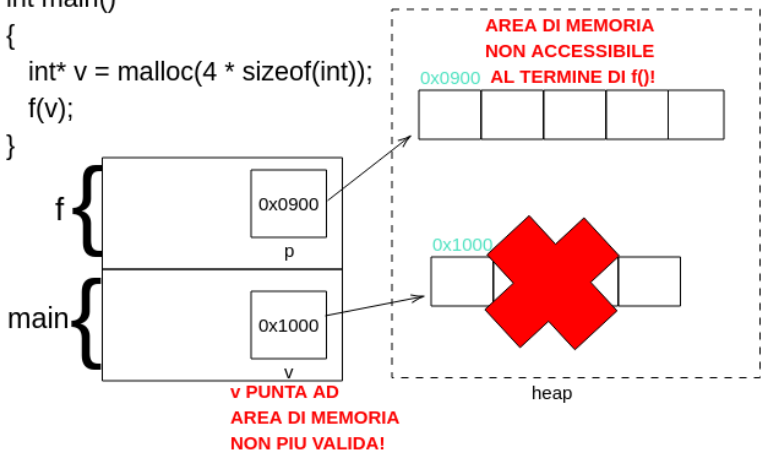


- la funzione `f()`, attraverso il puntatore `p`, può accedere in lettura/scrittura alla stessa area di memoria puntata da `v`. **Non può però modificare in contenuto di `v`**.

Esempio: si supponga che `f(...)` necessiti di aggiungere un elemento all'area di memoria (quindi di modificarne non solo il contenuto, ma anche la forma).

Strategia ingenua (ed errata):


```
void f(int* p)
{
p = realloc (p, 5*sizeof(int));
....
}

int main()
{
  int* v = malloc(4 * sizeof(int));
  f(v);
}
```



Si ricorda che la `realloc(p,...)`, in caso di esecuzione riuscita, dealloca la precedente area di memoria a cui punta `p`.

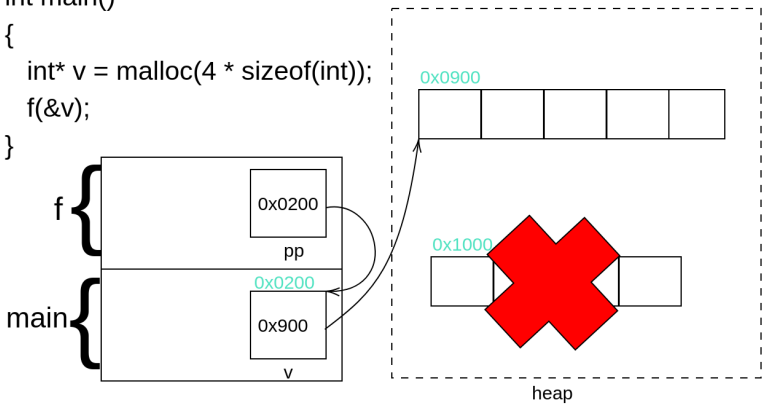
Quindi:

1. al termine di `f()`, `v` punterà ad un'area di memoria deallocata
2. la nuova area di memoria, frutto della `realloc(p,...)`, non sarà accessibile fuori da `f()`, in quanto il puntatore `p` (contenente l'indirizzo della nuova area) viene deallocato al termine della funzione

NECESSITA' DI CAMBIARE IL VALORE DI `v`

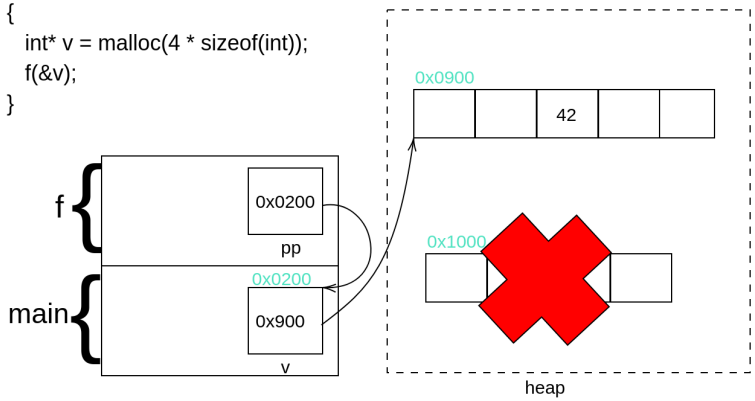
Prima strategia: utilizzare un puntatore a puntatore

```
void f(int** pp)
{
  *pp = realloc (*pp, 5*sizeof(int)); //pp punta a v
   ....
}

int main()
{
  int* v = malloc(4 * sizeof(int));
  f(&v);
}
```



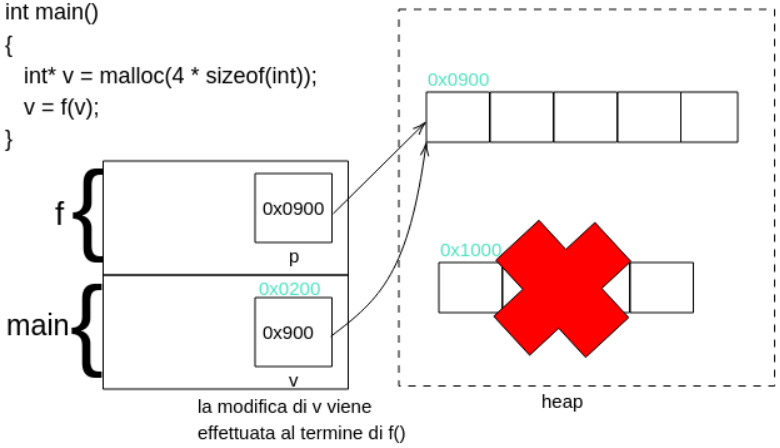
L'accesso agli elementi può essere effettuato attraverso il doppio puntamento

```
void f(int** pp)
{
  *pp = realloc (*pp, 5*sizeof(int)); //pp punta a v
   (*pp)[2] = 42;  // <=> *((*pp)+2) = 42;
}

int main()
{
  int* v = malloc(4 * sizeof(int));
  f(&v);
}
```



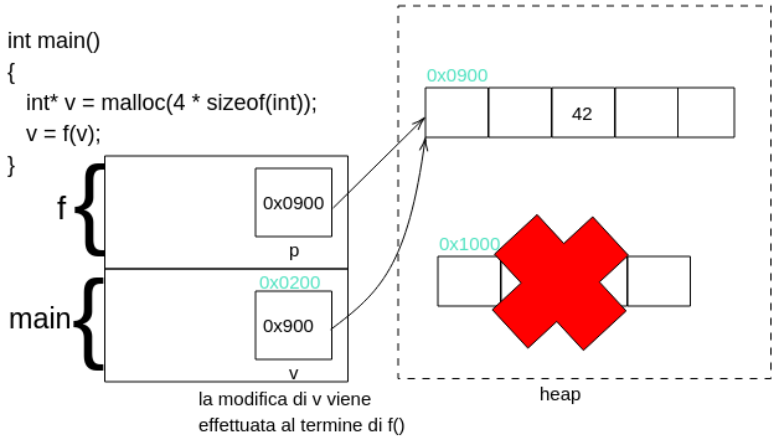
Seconda strategia: sfruttare il valore di ritorno


```
int* f(int* p)
{
  p = realloc (p, 5*sizeof(int)); //pp punta a v
  ...
  return p;
}

int main()
{
  int* v = malloc(4 * sizeof(int));
  v = f(v);
}
```



L'accesso agli elementi può essere effettuato attraverso il puntamento classico

```
int* f(int* p)
{
  p = realloc (p, 5*sizeof(int)); //pp punta a v
  p[2] = 42; // <=> *(p+2) = 42;
  ...
  return p;
}

int main()
{
  int* v = malloc(4 * sizeof(int));
  v = f(v);
}
```



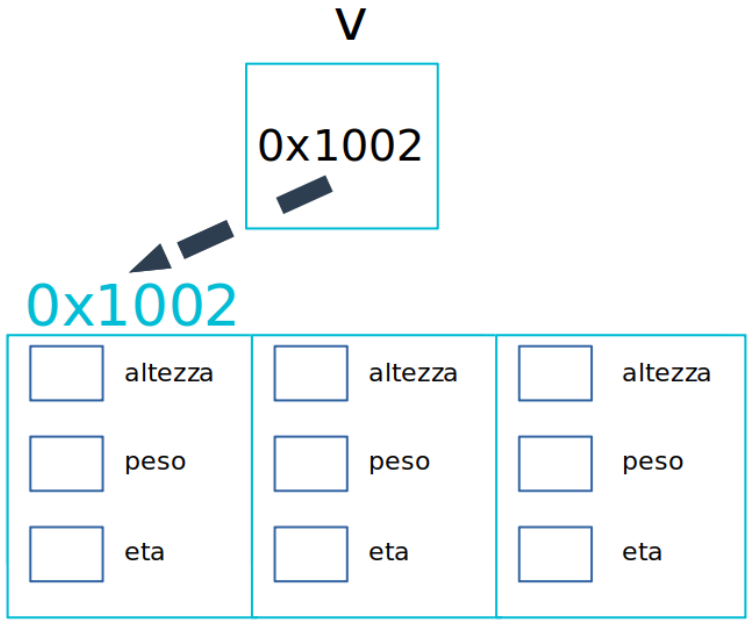
- pro: memoria contigua
- pro: posso indicizzare utilizzando più operatori `[]`

Allocazione dinamica di una Struct

In [7]:
```
#include <stdio.h>
#define N 3

struct Paziente
{
    float altezza;
    float peso;
    int   eta;
};

int main()
{
    struct Paziente* p = malloc(sizeof(struct Paziente));

    printf("inserisci l'altezza del paziente: ");
    scanf("%f", &(p->altezza));
    printf("inserisci il peso del paziente: ");
    scanf("%f", &(p->peso));
    printf("inserisci l'età del paziente: ");
    scanf("%d", &(p->eta));

    printf("Altezza inserita: %.2f\n", p->altezza);
    printf("Peso inserito: %.2f\n", p->peso);
    printf("età inserita: %d\n", p->eta);

    return 0;
}
```

```
inserisci l'altezza del paziente:

inserisci il peso del paziente:

inserisci l'età del paziente:

Altezza inserita: 1.70
Peso inserito: 60.50
età inserita: 20
```

Array di Struct allocate dinamicamente

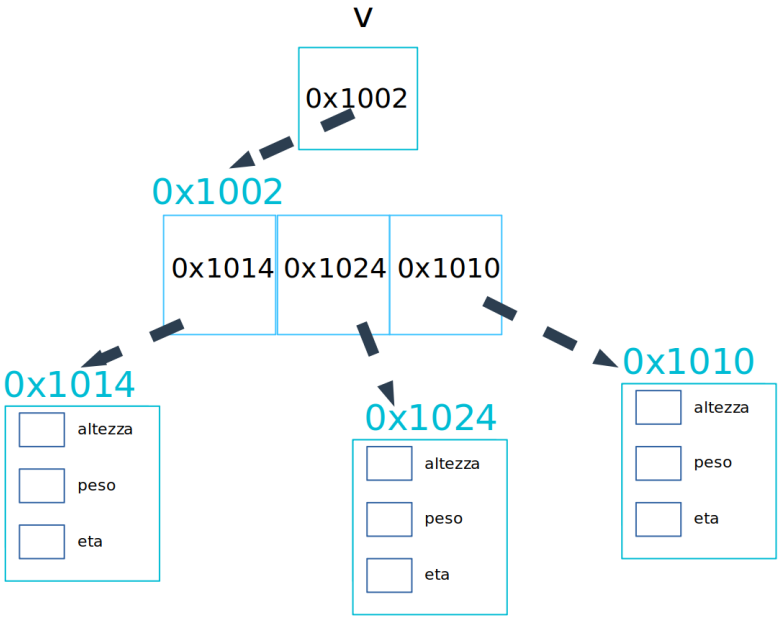


In [10]:
```
#include <stdio.h>
#define N 3

struct Paziente
{
    float altezza;
    float peso;
    int   eta;
};

int main()
{


    printf("Quanti pazienti vuoi inserire? ");
    int n;
    scanf("%d", &n);

    struct Paziente* v = calloc(n, sizeof(struct Paziente));

    for (int i = 0; i < n; i++)
    {
        printf("inserisci l'altezza del paziente %d: ", i);
        scanf("%f", &(v[i].altezza));
        printf("inserisci il peso del paziente %d: ", i);
        scanf("%f", &(v[i].peso));
        printf("inserisci l'età del paziente %d: ", i);
        scanf("%d", &(v[i].eta));
    }
```

```
    for (int i = 0; i < n; i++)
    {
        printf("Altezza inserita del paziente %i: %.2f\n", i, v[i].altezz
        printf("Peso inserito del paziente %i: %.2f\n", i, v[i].peso);
        printf("età inserita del paziente %i: %d\n", i, v[i].eta);
    }

    free(v);
    return 0;
}
```

```
Quanti pazienti vuoi inserire?

inserisci l'altezza del paziente 0:

inserisci il peso del paziente 0:

inserisci l'età del paziente 0:

inserisci l'altezza del paziente 1:

inserisci il peso del paziente 1:

inserisci l'età del paziente 1:

Altezza inserita del paziente 0: 1.70
Peso inserito del paziente 0: 60.00
età inserita del paziente 0: 34
Altezza inserita del paziente 1: 1.80
Peso inserito del paziente 1: 50.00
età inserita del paziente 1: 322
```

oppure...



In [ ]:

```
#include <stdio.h>
#define N 3

struct Paziente
{
    float altezza;
    float peso;
    int   eta;
};

int main()
{


    printf("Quanti pazienti vuoi inserire? ");
    int n;
    scanf("%d", &n);

    struct Paziente** v = calloc(n, sizeof(struct Paziente*));
    for (int i = 0; i < n; i++)
    {
        v[i] = malloc(sizeof(struct Paziente));
    }


    for (int i = 0; i < n; i++)
    {

        printf("inserisci l'altezza del paziente %d: ", i);
        scanf("%f", &(v[i]->altezza));
        printf("inserisci il peso del paziente %d: ", i);
        scanf("%f", &(v[i]->peso));
        printf("inserisci l'età del paziente %d: ", i);
        scanf("%d", &(v[i]->eta));
    }


    for (int i = 0; i < n; i++)
    {
        printf("Altezza inserita del paziente %i: %.2f\n", i, v[i]->altez
        printf("Peso inserito del paziente %i: %.2f\n", i, v[i]->peso);
        printf("età inserita del paziente %i: %d\n", i, v[i]->eta);
    }


    for (int i = 0; i < n; i++)
    {
            free(v[i]);
    }
    free(v);
    return 0;
}
```

```
Quanti pazienti vuoi inserire?

inserisci l'altezza del paziente 0:

inserisci il peso del paziente 0:

inserisci l'età del paziente 0:
```

```
inserisci l'altezza del paziente 1:

inserisci il peso del paziente 1:

inserisci l'età del paziente 1:

Altezza inserita del paziente 0: 1.00
Peso inserito del paziente 0: 30.00
età inserita del paziente 0: 32
Altezza inserita del paziente 1: 1.60
Peso inserito del paziente 1: 60.00
età inserita del paziente 1: 34
```

Array di Struct allocate dinamicamente - Ricapitolando...

Ho quindi almeno due modi per allocare un array di struct:

1. allocazione di un array di elementi struct
2. allocazione di un array di puntatori, ognuno di questi che punta ad una struct diversa

Attenzione! Potrei avere differenze di comportamento in caso di accesso!

Esempio:

v: array di struct Paziente

```
struct Paziente* v = ...
v[0].eta = 15;
struct Paziente paz;
paz      = v[0];
paz.eta  = 34;
printf("%d\n", v[0].eta);
```

v: array di puntatori

```
struct Paziente** v = ...
v[0]->eta = 15;
struct Paziente* paz;
paz      = v[0];
paz->eta = 34;
printf("%d\n", v[0]->eta);
```

- Nel caso in cui `v` sia un array di `struct Paziente` (caso a sinistra), `paz` conterrà una *copia* dell'elemento `v[0]`
- Nel caso in cui `v` sia un array di puntatori a `struct Paziente` (caso a destra), `paz` conterrà l'indirizzo dello stesso elemento a cui punta `v[0]`

Ovviamente è possibile allocare anche array multidimensionali di struct, seguendo le diverse strategie illustrate in precedenza.

In [ ]: