

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Layout della memoria di un programma C

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 variabili 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 `contiguous 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:

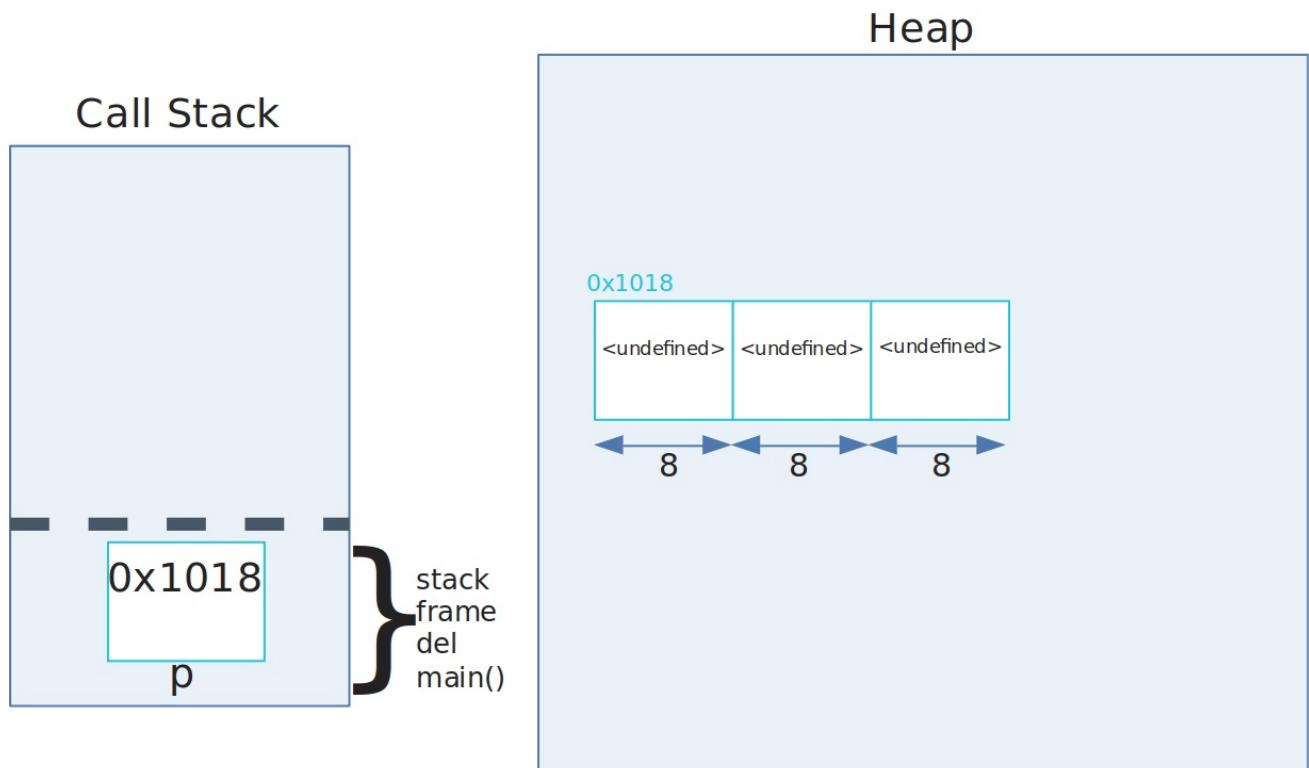
```
int main()
```

```

    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:

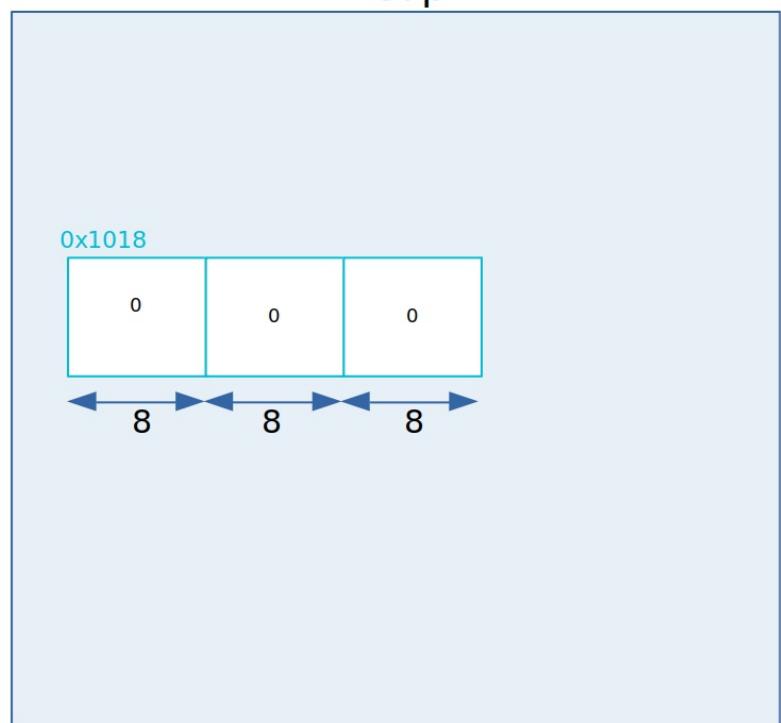
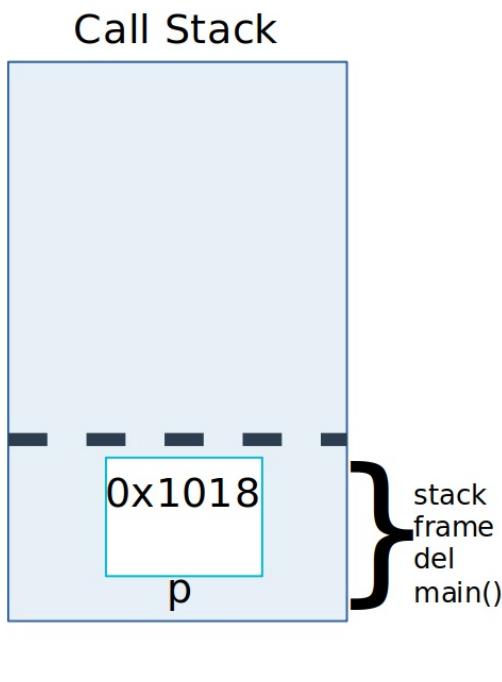
- If the size of the space requested is 0, the behavior is implementation-defined.
- If the space cannot be allocated, a null pointer shall be returned

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.

Heap



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 sia davvero contigua, ma dato che tutti gli accessi vengono effettuati attraverso indirizzi virtuali forniti dal SO, 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` → 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` **dove** 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 deallocated, 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, *(p+1), *(p+2), *(p+5), *(p+10));

    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", *p, *(p+1), *(p+2), *(p+5), *(p+10));

    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));

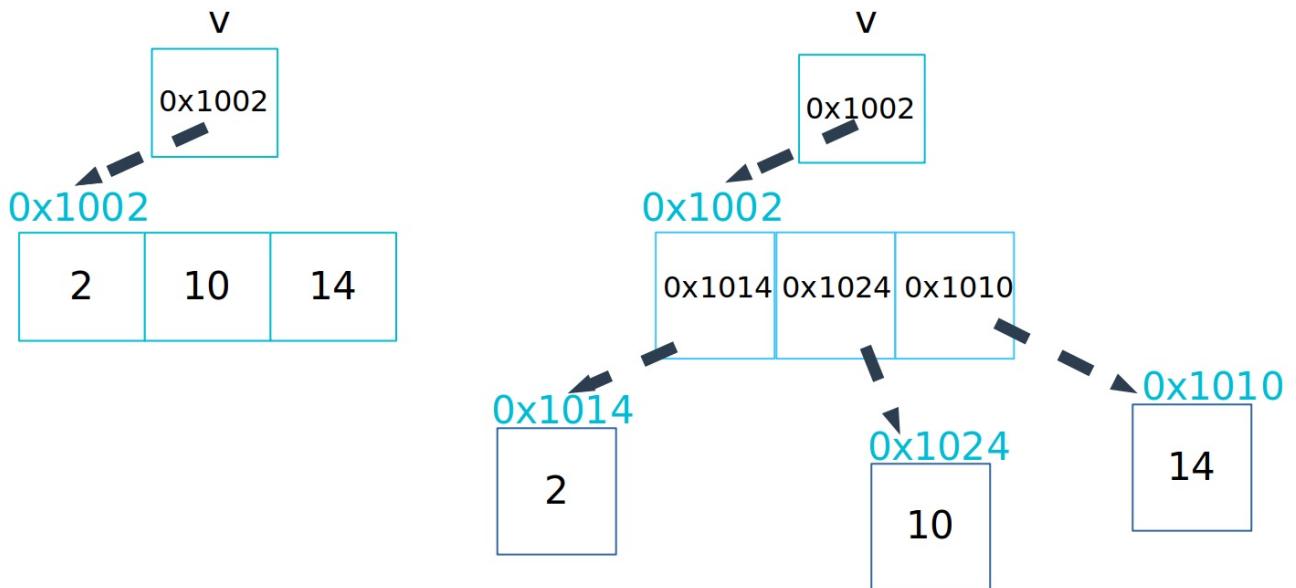
    printf("inserisci gli elementi:\n");
    acquisisci_vett(v,n);
    printf("gli elementi inseriti sono: ");
    stampa_vett(v,n);

    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)



tal alternativa in generale è **SCONSIGLIATA!**

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

Allocazione dinamica di una matrice

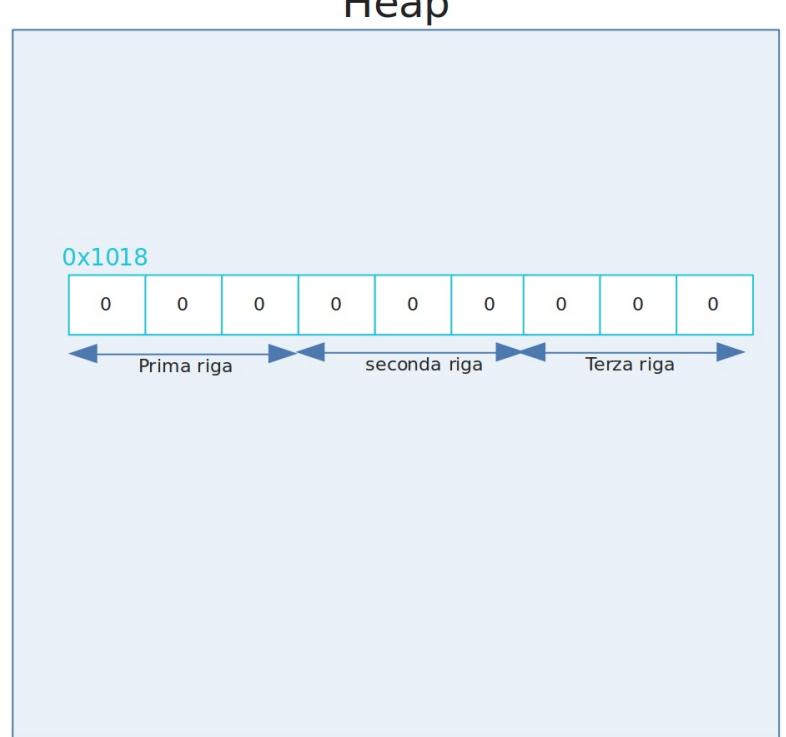
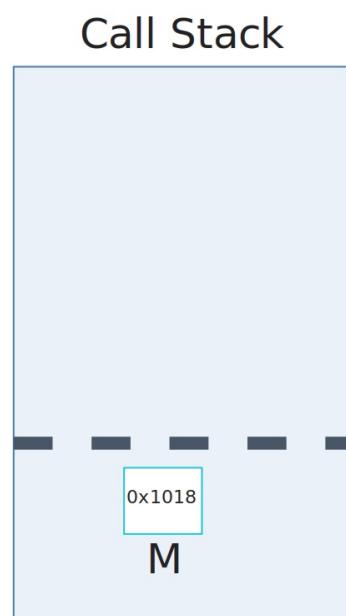
- Possono essere allocati con `calloc(...)` / `malloc(...)` in vari modi (almeno 3)

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

$\mathbf{v}_R = (m_{11}, m_{12}, \dots, m_{1C}, m_{21}, m_{22}, \dots, m_{2C}, \dots m_{RC})$ 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:
```

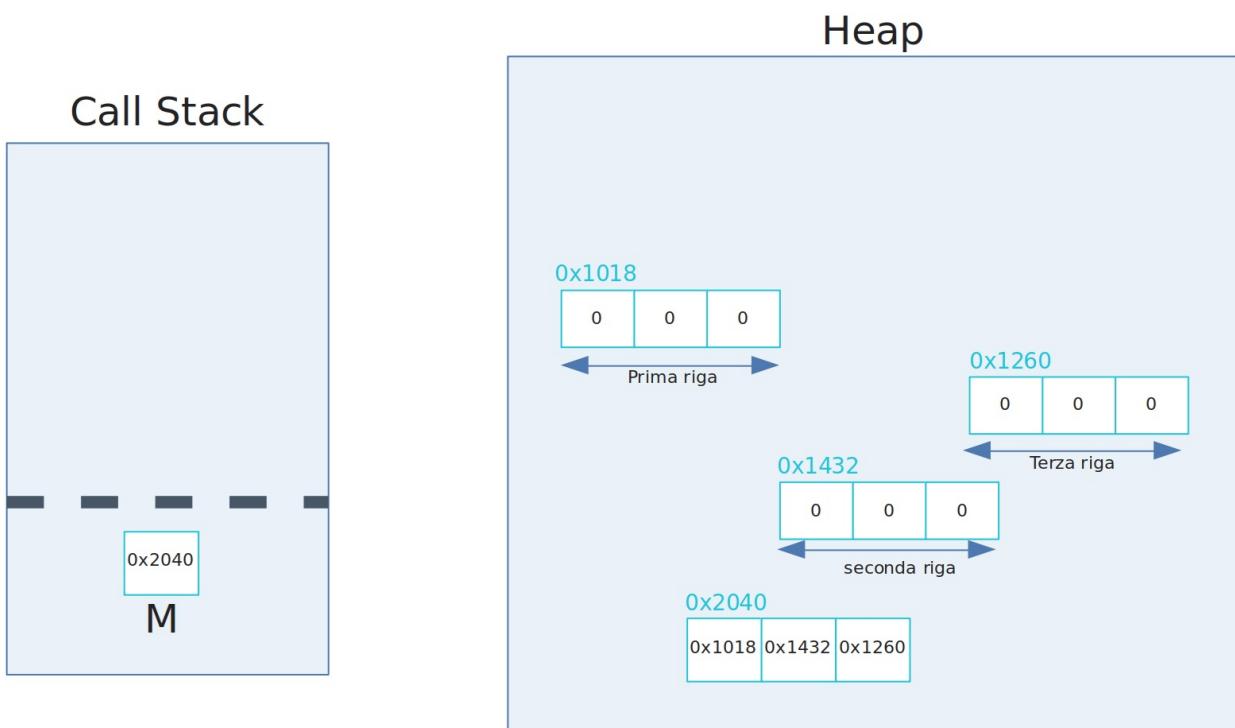
```
gli elementi inseriti sono:
```

```
(  
    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 n_r di 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?

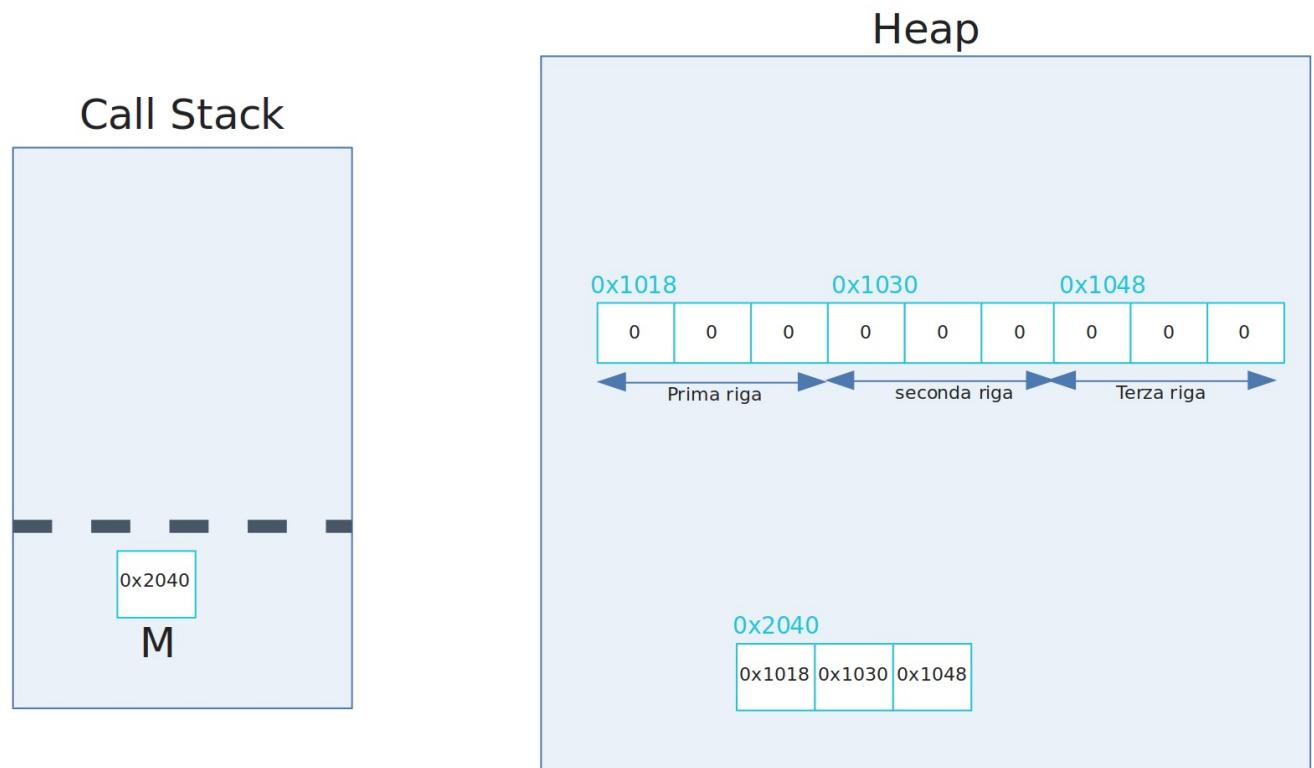
inserisci gli elementi:

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 $\mathbf{v}_R = (m_{11}, m_{12}, \dots, m_{1n}, m_{21}, m_{22}, \dots, m_{2n}, \dots, m_{nn})$ 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
)

- 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;
}

```

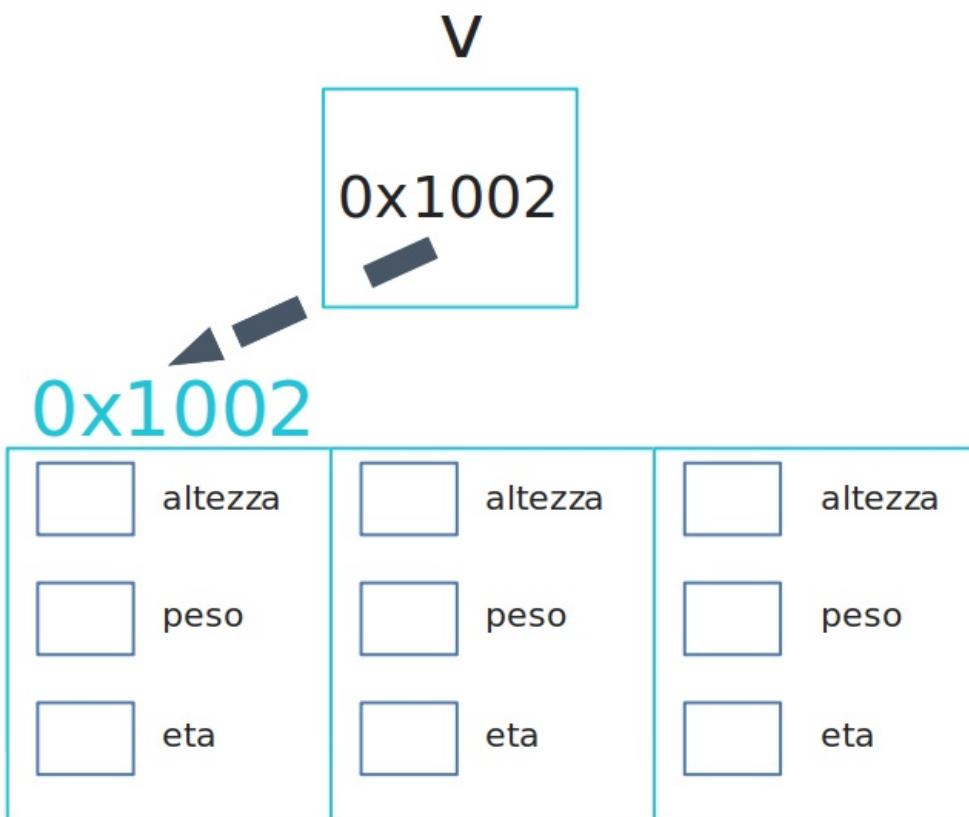
inserisci l'altezza del paziente:

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 = malloc(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].altezza);
        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

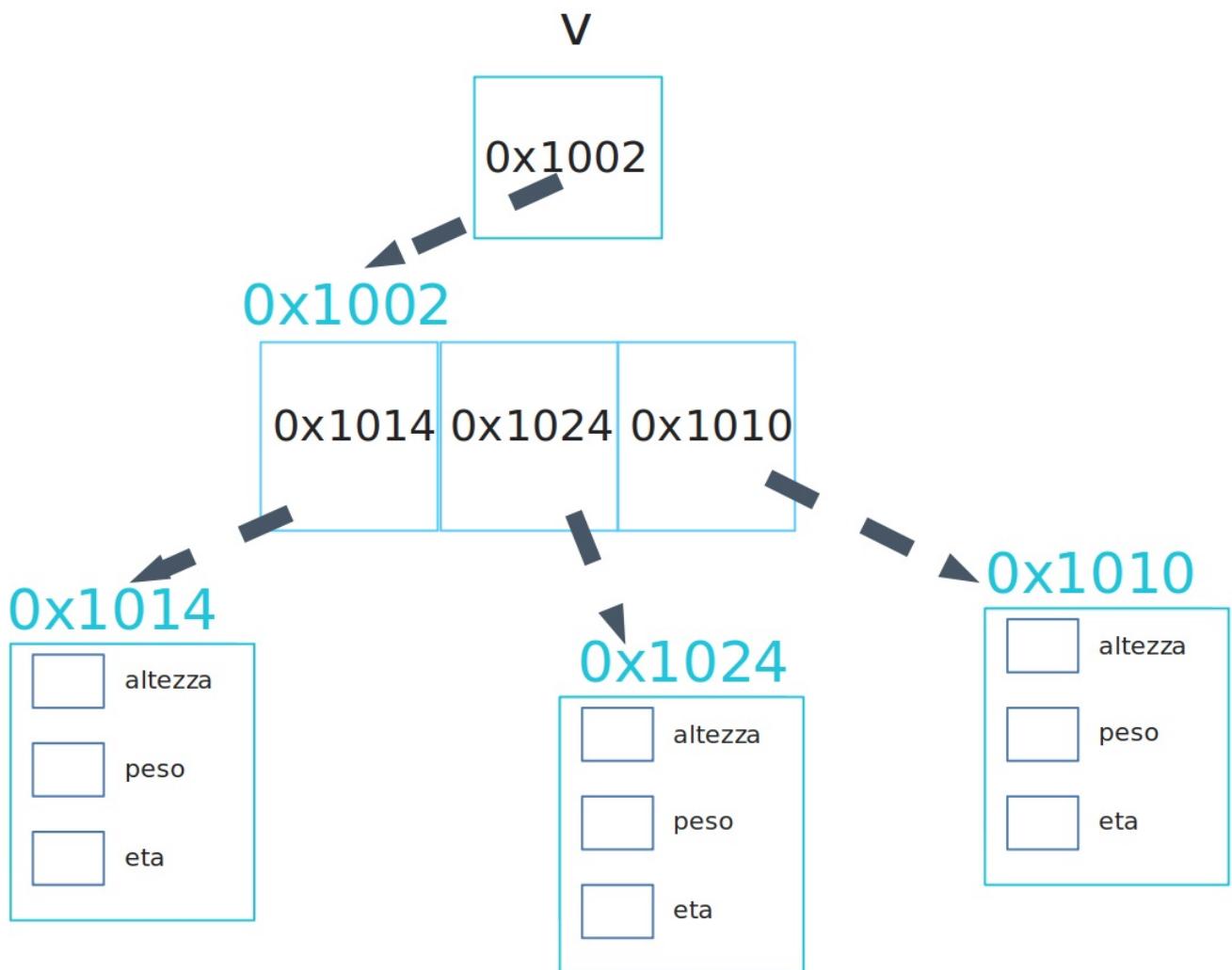
età inserita del paziente 0: 34

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]->altezza);
    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:

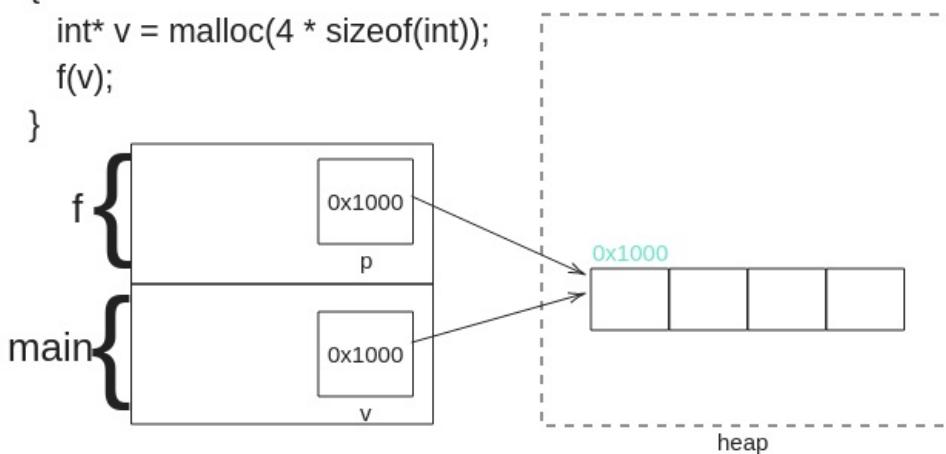
v: array di struct Paziente	v: array di puntatori
<pre> struct Paziente* v = ...; v[0].eta = 15; struct Paziente paz; paz = v[0]; paz.eta = 34; printf("%d\n", v[0].eta); </pre>	<pre> struct Paziente** v = ... v[0]->eta = 15; struct Paziente* paz; paz = v[0]; paz->eta = 34; printf("%d\n", v[0]->eta); </pre>

- 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]`

L'importanza di poter puntare

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

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



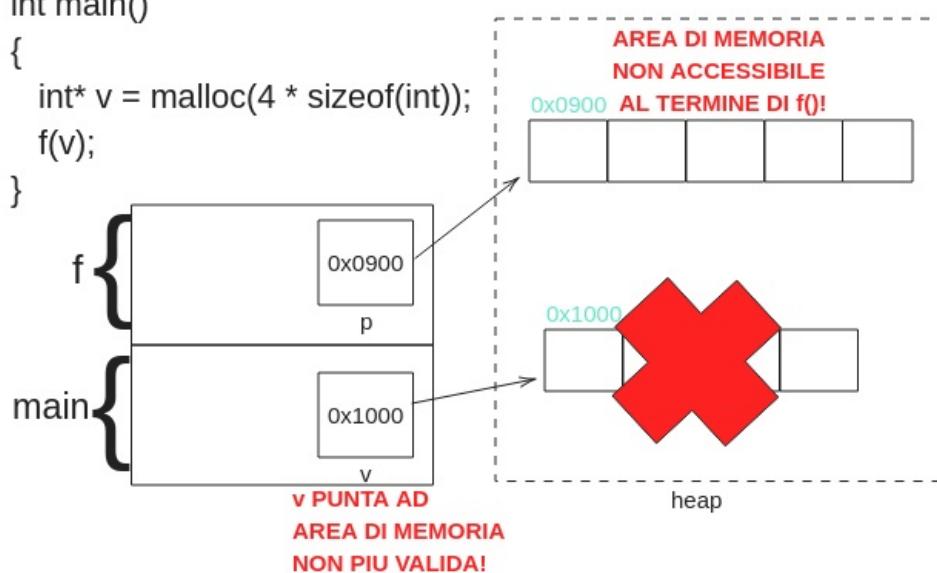
- la funzione `f()`, attraverso il puntatore `p`, può accedere in lettura/scrittura alla stessa area di memoria puntata da `v`. **Non può però modificare in contenuto di `v`**.

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:

```
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:

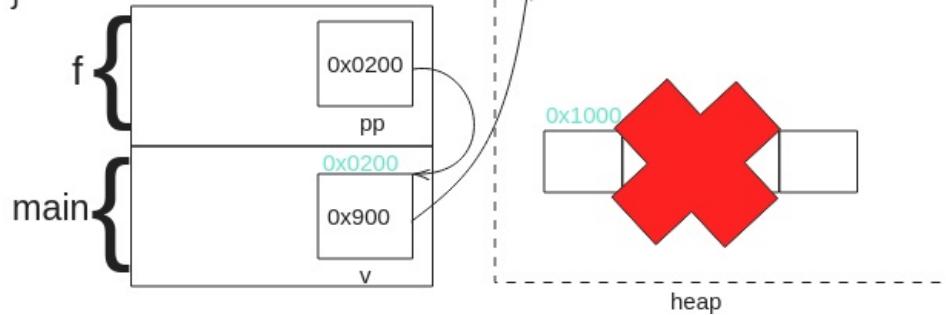
1. al termine di `f()`, `v` punterà ad un'area di memoria deallocata
2. la nuova area di memoria, frutto della `realloc(p, ...)`, non sarà accessibile fuori da `f()`, in quanto il puntatore `p` (contenente l'indirizzo della nuova area) viene deallocated al termine della funzione

NECESSITA' DI CAMBIARE IL VALORE DI `v`

Prima strategia: utilizzare un puntatore a puntatore

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

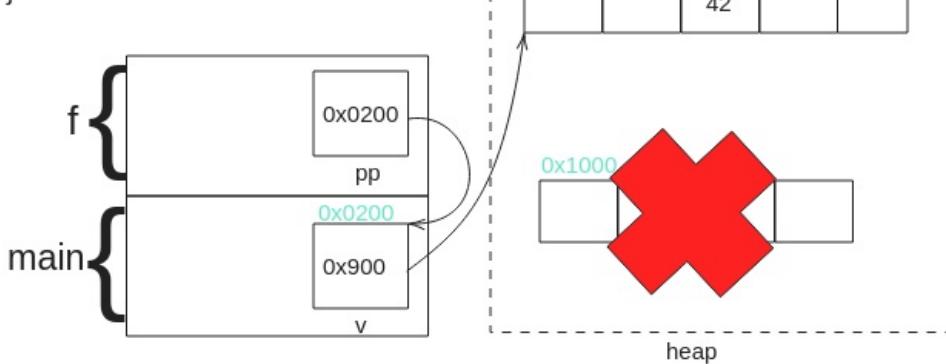
```
int main()
{
    int* v = malloc(4 * sizeof(int));
    f(&v);
}
```



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

```
void f(int** pp)
{
    *pp = realloc (p, 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

```

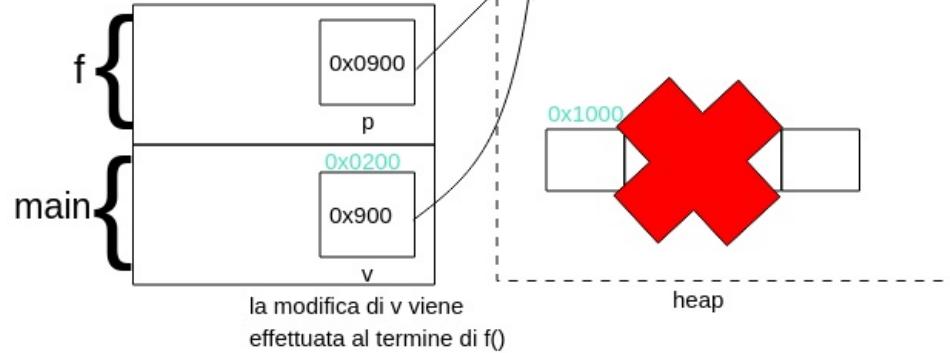
int* f(int* p)
{
    p = realloc (p, 5*sizeof(int)); //pp punta a v
    ...
    return p;
}

```

```

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

```



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

```

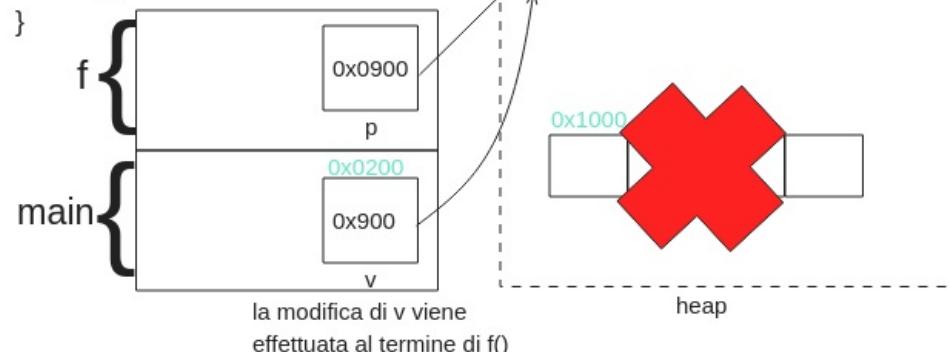
int* f(int* p)
{
    p = realloc (p, 5*sizeof(int)); //pp punta a v
    p[2] = 42; // <=> *(p+2) = 42;
    ...
    return p;
}

```

```

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

```



In []:

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Aritmetica dei puntatori

Sui (valori contenuti nelle variabili) puntatori sono definite le operazioni di *somma* e *differenza*, ma diverse da quelle canoniche.

dato un puntatore di nome *p* di tipo *T ** ed un intero *n* l'operazione:

- *p + n* punta all'indirizzo $p + n \cdot \text{sizeof}(T)$
- *p - n* punta all'indirizzo $p - n \cdot \text{sizeof}(T)$

dati due puntatori rispettivamente con nomi *p* e *q* ed entrambi di tipo *T **, l'operazione

- *p - q* punta all'indirizzo $\frac{p - q}{\text{sizeof}(T)}$

Queste operazioni possono essere utili nel caso di accesso ad aree di memoria continua (esempio, array)

```
In [1]: #include <stdio.h>
int main()
{
    int v[4] = {42, 101, 106, 108};
    int* punt = &v[0];
    int* punt2 = punt;
    printf("*punt: %d\n", *punt);
    printf("incremento punt di 1...\n");
    punt = punt + 1;
    printf("*punt: %d\n", *punt);
    printf("incremento punt di 1...\n");
    punt = punt + 1;
    printf("*punt: %d\n", *punt);
    printf("decremento punt di 1...\n");
    punt--;
    printf("*punt: %d\n", *punt);
    printf("incremento punt di 2...\n");
    punt += 2;
    printf("*punt: %d\n", *punt);
    printf("decremento punt di 1 ed aggiungo 200 al valore a cui punta...\n");
    *(--punt) += 200;
    printf("*punt: %d\n", *punt);
    printf("*punt+1: %d\n", *(punt+1));
    printf("*(punt+1): %d\n", *(punt+1));

    printf("punt punta a %d, l'elemento iniziale del vettore è %d.\n", *punt, *punt2);
    printf("punt: %llu (hex: %p), punt2: %llu (hex: %p)\n", punt, punt, punt2, punt2);
    printf("tra %d e %d ci sono %d elementi di dimensione %d Byte\n",
           *punt, *punt2, punt-punt2, sizeof(*punt));

    return 0;
}
```

```

*punt:      42
incremento punt di 1...
*punt:      101
incremento punt di 1...
*punt:      106
decremento punt di 1...
*punt:      101
incremento punt di 2...
*punt:      108
decremento punt di 1 ed aggiungo 200 al valore a cui punta...
*punt:      306
*punt+1:    307
*(punt+1): 108
punt punta a 306, l'elemento iniziale del vettore è 42.
punt: 140721125449960 (hex: 0x7ffc30b1e0e8), punt2: 140721125449952 (hex: 0x7ffc30b1e0e0)
tra 306 e 42 ci sono 2 elementi di dimensione 4 Byte

```

L'operatore []

dato un puntatore di nome `p` ed un intero contenuto in una variabile `n`, allora:

```
p[n] = *(p+n)
```

ad esempio, le istruzioni:

```
*(p+1) = 5; , *p = 100; , *(p-3) = *(p+1)+3;
```

possono essere riscritte come:

```
p[1] = 5; , p[0] = 100; , p[-3] = p[1]+3;
```

```
In [ ]: #include <stdio.h>
int main()
{
    int v[4] = {42, 101, 106, 108};
    int* punt = &v[0];
    printf("punt[0]: %d\n", punt[0]);
    punt += 3;
    printf("punt[0]: %d\n", punt[0]);
    printf("punt[-2]: %d\n", punt[-2]);

    return 0;
}
```

```
In [ ]: // Aritmetica dei puntatori
#include <stdio.h>
int main()
{
    int answer = 42;
    int* punt = &answer;
    printf("answer: %d\n", answer);
    printf("punt: %llu\n", punt);
    printf("*punt: %d\n", *punt);

    punt = punt + 1;

    printf("=====dopo incremento====\n");
    printf("punt: %llu\n", punt);
    printf("*punt: %d\n", *punt);

    return 0;
}
```

```
In [2]: #include <stdio.h>
int main()
{
    int answer = 42;
    int* punt = &answer;
    printf("answer: %d\n", answer);
    printf("&answer(hex): %p\n", &answer);
    printf("&answer(dec): %llu\n", &answer);
    printf("===== \n");
    printf("punt: %llu\n", punt);
    printf("*punt: %d\n", *punt);
    printf("&punt(hex): %p\n", &punt);
    printf("&punt(dec): %llu\n", &punt);
    printf("===== ma anche... =====\n");
    printf("*(&answer): %d\n", *(&answer));

    *punt = *punt + 10;

    printf("== modificando *punt ==\n");
    printf("answer: %d\n", answer);
    printf("*punt: %d\n", *punt);
```

```

    return 0;
}

answer:      42
&answer(hex): 0x7ffe546f9eec
&answer(dec): 140730315022060
=====
punt:      140730315022060
*punt:      42
&punt(hex): 0x7ffe546f9ef0
&punt(dec): 140730315022064
===== ma anche... =====
*(&answer): 42
== modificando *punt ==
answer:      52
*punt:      52

```

In [1]: #include <stdio.h>

```

void f(int f_v[]) // equivalente a void f(int* f_v)
{
    printf("sizeof(f_v): %d Byte\n", sizeof(f_v));
    printf("f_v+1: %p\n", f_v+1);
}

int main()
{
    int v[] = {3, 10, 42};
    int *p_v = v;
    printf("sizeof(v): %d Byte\n", sizeof(v));
    printf("v: %p; v+1: %p\n", v, v+1);
    printf("sizeof(p_v): %d Byte\n", sizeof(p_v));
    printf("p_v: %p; p_v+1: %p\n", p_v, p_v+1);

    f(v);

    return 0;
}

```

```

/tmp/tmpzdnmvey4.c: In function 'f':
/tmp/tmpzdnmvey4.c:4:44: warning: 'sizeof' on array function parameter 'f_v' will return size of 'int *' [-Wsize-of-array-argument]
  4 |     printf("sizeof(f_v): %d Byte\n", sizeof(f_v));
   |
/tmp/tmpzdnmvey4.c:2:12: note: declared here
  2 | void f(int f_v[]) // equivalente a void f(int* f_v)
   | ~~~~~^~~~~~
sizeof(v): 12 Byte
v: 0xffff510af2ec; v+1: 0xffff510af2f0
sizeof(p_v): 8 Byte
p_v: 0xffff510af2ec; p_v+1: 0xffff510af2f0
sizeof(f_v): 8 Byte
f_v+1: 0xffff510af2f0

```

Un esempio di utilizzo dell'aritmetica dei puntatori:

In [5]: #include <stdio.h>

```

void stampa_vett(int v[], int n) // equivalente a void f(int* v, int n)
{
    printf(" ");
    for(int i = 0; i < n; i++)
        printf("%d, ", v[i]);
    printf("\b\b ");
}

int main()
{
    int v[] = {3, 10, 42, 8, 4, 2};
    int n = 6;

    //stampo l'array v a partire dalla terza posizione
    stampa_vett(v+2, n-2); // passo l'indirizzo della terza posizion

    return 0;
}

( 42, 8, 4, 2 )

```

Aritmetica dei puntatori su matrici allocate con operatore []

```
#include <stdio.h>
#define MAX_COLS 10
void stampa_mat(int mat[][MAX_COLS], 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(" %2d, ", mat[i][j]);
        printf("\\b\\b\\n");
    }
    printf(")\\n");
}
int main()
{
    int M[] [MAX_COLS] = {{3, 10, 42},
                          {8, 7, 64},
                          {11, 22, 33}
                        };

    stampa_mat(M, 3, 3);
    printf("M: %p, *M: %p, **M: %d\\n", M, *M, **M);
    /* NB: M non è da considerarsi come un puntatore a puntatore ad int,
       ma è un puntatore ad un array di int, che è un tipo distinto */
    printf("M+1: %p, *(M+1): %p, **(M+1): %d\\n", M+1, *(M+1), **(M+1));
    printf("M+2: %p, *(M+2): %p, **(M+2): %d\\n", M+2, *(M+2), **(M+2));

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

    printf("M[2]: %p, *M[2]: %d\\n", M[2], *M[2]);
    return 0;
}

(
  3, 10, 42
  8, 7, 64
  11, 22, 33
)
M: 0x7ffc06b36a50, *M: 0x7ffc06b36a50, **M: 3
M+1: 0x7ffc06b36a78, *(M+1): 0x7ffc06b36a78, **(M+1): 8
M+2: 0x7ffc06b36aa0, *(M+2): 0x7ffc06b36aa0, **(M+2): 11
M+2: 0x7ffc06b36aa0, *M+2: 0x7ffc06b36a58, *(*M+2): 42
M+2+1:0x7ffc06b36ac8, *(M+2)+1:0x7ffc06b36aa4, *(*(M+2)+1): 22
M[2]: 0x7ffc06b36aa0, *M[2]: 11
```

Una matrice `M` di tipo `T` è vista come un puntatore ad array di `MAX_COLS` elementi di tipo `T`, ossia: `int (*) [MAX_COLS]` ← puntatore `(*)` ad array `[]` di tipo `int`. Le parentesi tonde servono proprio a specificare che si tratta di un puntatore ad array, e non di un array di puntatori (ossia `int* [MAX_COLS]`).

Questa tipologia di puntatori non saranno trattati in questo corso, basti sapere che:

```
int (*) [MAX_COLS] ≠ int**
```

In sostanza, `M+n` con `n`, contenente un intero positivo, punta alla riga di indice `n`.

Si noti quindi che vale ancora:

```
M[i] \equiv *(M+i)
```

Quindi `M[i][j] \equiv *(*(M+i) + j)`

In quanto:

1. attraverso il primo livello di *dereferencing* (ossia `*(M+i)`) si accederà alla riga di indice `i`,
2. essendo tale riga un array di `[MAX_COLS]` elementi, posso accedere ai suoi elementi utilizzando l'aritmetica dei puntatori
3. In particolare, per accedere all'elemento di indice `j`, sfruttando sempre l'aritmetica dei puntatori è sapendo che `*(M+i)` è un array, basta sommarci `j`. `*(M+i)+j` sarà quindi l'indirizzo dell'elemento in riga di indice `i` e in colonna di indice `j`. Essendo un indirizzo, per avere il valore effettivo utilizzo un secondo livello di *dereferencing*, ossia `(*(*(M+i) + j))`.

```
In [14]: #include <stdio.h>
#define MAX_COLS 3

int main()
{
    //char v[] = {3, 8, 10};
    int M[] [MAX_COLS] = {{3,10,20},
                          {8,4,2},
                          {10,14,16},
                          {1,2,3}
                        };
    //printf("sizeof(v): %lld\\n", sizeof(v));
    long int M_più_2 = (long int)(M + 2);
```

```

long int M_piu_1 = (long int)(M + 1);
long int M_0      = (long int) M;
printf("M+1: %llu    M: %llu    diff: %llu\n", M_piu_1, M_0, M_piu_1 - M_0);
printf("M+2: %llu    M: %llu    diff: %llu\n", M_piu_2, M_0, M_piu_2 - M_0);

printf("sizeof(M):           %lld\n", sizeof(M));
printf("sizeof(M[0]):         %lld\n", sizeof(M[0]));
printf("sizeof(M[0][0]):       %lld\n", sizeof(M[0][0]));
printf("n_c * sizeof(M[0][0]): %lld\n", MAX_COLS * sizeof(M[0][0]));

return 0;
}

```

```

M+1: 140721888639772  M: 140721888639760  diff: 12
M+2: 140721888639784  M: 140721888639760  diff: 24
sizeof(M):
48
sizeof(M[0]):
12
sizeof(M[0][0]):
4
n_c * sizeof(M[0][0]): 12

```

sia `M` definita come `int M[n_r][n_c]`, allora:

`M` punta alla *riga 0* della matrice. `M+i` punterà quindi alla $i+1$ -esima riga della matrice. L'indirizzo effettivo puntato da `M+i` sarà quindi dato da `M + i \cdot n_c \cdot \text{sizeof(int)}`.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Laboratorio di Programmazione

Corso di Laurea in Informatica

Gr. 3 (N-Z)

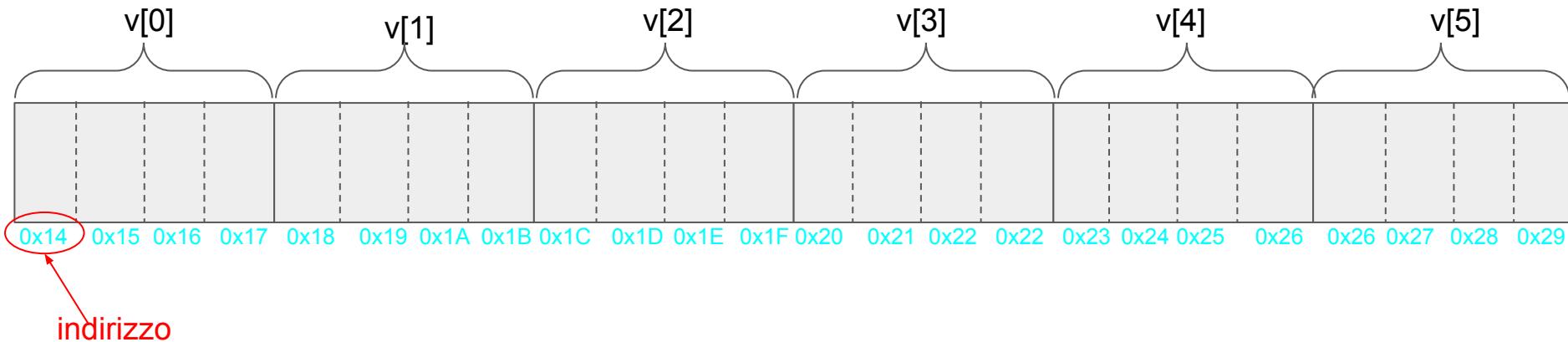
Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Array

- **sequenza** di elementi di tipo **omogeneo**
- identificato da un nome
- ogni elemento è identificato da un indice di posizione
- rappresentato in memoria come una **sequenza** di byte
- l'indirizzo di un array corrisponde all'indirizzo del **primo** elemento che lo compone



Array

```
int v[6];  
v[4] = 2;
```

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
?	?	?	?	2	?

Da notare che l'operatore [] assume due significati differenti in base a dove è usato:

- in fase di **dichiarazione/definizione**: specifica *quanti* elementi compongono l'array
- in fase di **accesso**: specifica *dove* leggere/scrivere il dato. In tale fase è detto anche *subscript operator*.

il valore tra [] deve essere di tipo intero, e può essere sia un valore costante che una variabile.

Esempio:

- int a = 4;
- v[a] = 42; // inserisce nella posizione di indice 4 il valore 42

In fase di accesso i valori di posizione plausibili vanno da 0 a n-1 dove n è il numero di elementi che compongono l'array esplicitato in fase di dichiarazione.

Vettori

Un array può essere utilizzato in maniera naturale per rappresentare un vettore

matematica

$$\mathbf{v} = (45, 2, 7, 1, 52, 66)$$

C

```
int v[6] = {45, 2, 7, 1, 52, 66};
```

memoria

v[0] v[1] v[2] v[3] v[4] v[5]

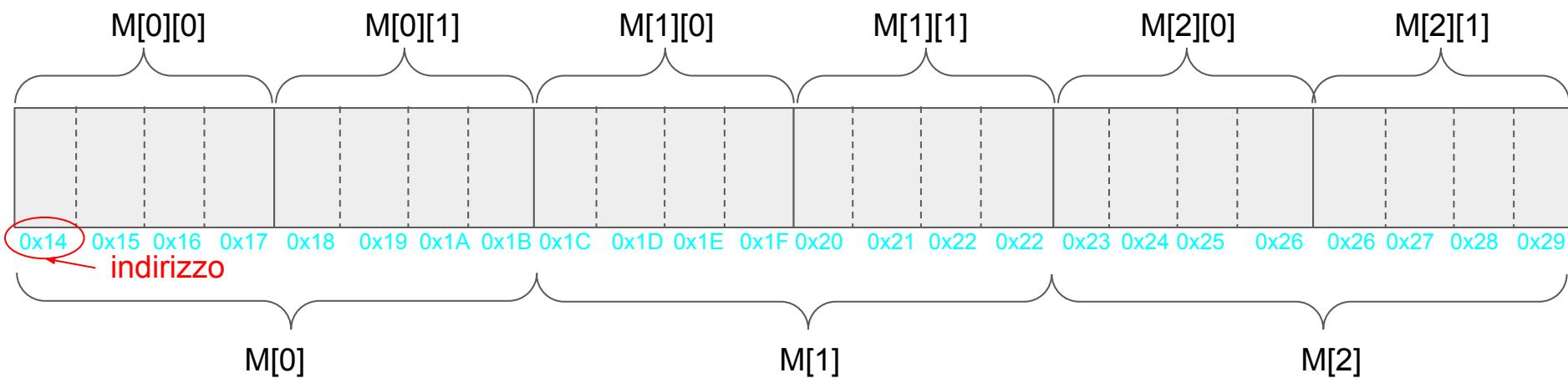
45	2	7	1	52	66
----	---	---	---	----	----

Array bidimensionali

- Il C permette di dichiarare array di array, ossia composto a sua volta di array dello stesso tipo
- ogni elemento di un array multidimensionale è quindi, a sua volta, un array

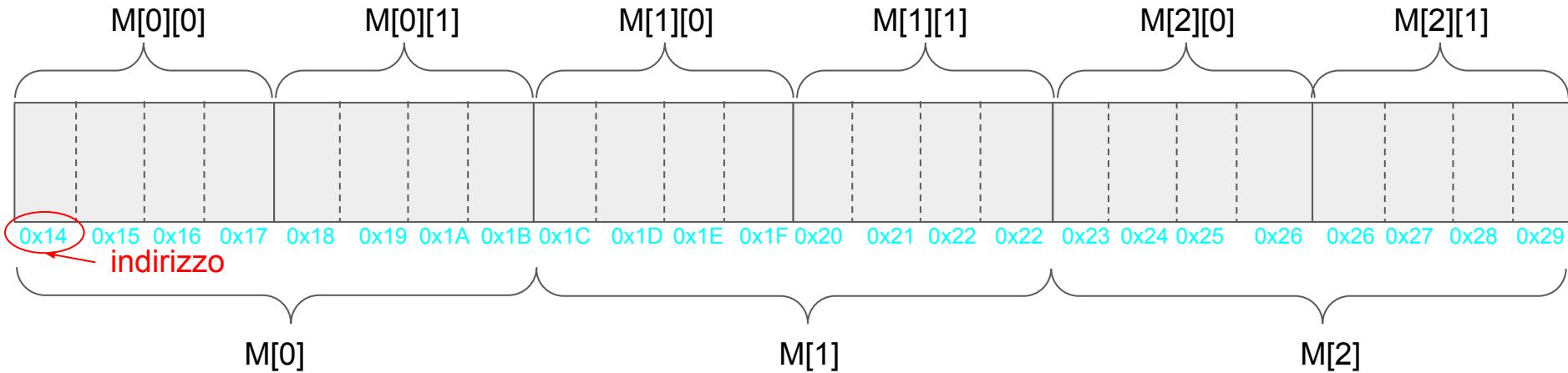
```
int M[3][2]; // dichiara un array composto da 3 array, ognuno dei quali  
              // è a sua volta composto da 2 int
```

```
M[1][0] = 5; // inserisce il 5 nel secondo array (indice 1),  
              // in prima posizione (indice 0)
```



Array bidimensionali

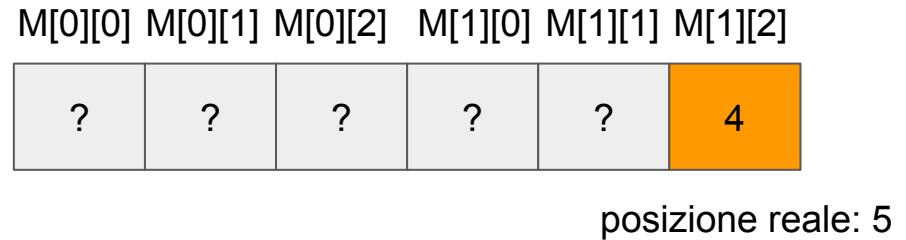
La rappresentazione in memoria di un array multidimensionale è identica a quella di un array monodimensionale



- Quando viene allocato un vettore bidimensionale, viene allocato nella realtà un array monodimensionale
- Cambia però l'indicizzazione logica, ossia gli indici necessari a leggere/scrivere un dato valore
- Internamente, l'array bidimensionale viene gestito come un array monodimensionale

Array bidimensionali

```
int M[2][3];  
M[1][2] = 4;
```



Internamente, gli indici di posizione vengono convertiti nella posizione reale dell'array in memoria

Array bidimensionali

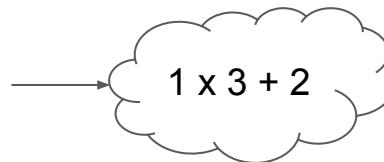
```
int M[2][3];  
M[1][2] = 4;
```

M[0][0]	M[0][1]	M[0][2]	M[1][0]	M[1][1]	M[1][2]
?	?	?	?	?	4

posizione reale: 5

indici

(1,2)



Internamente, gli indici di posizione vengono convertiti nella posizione reale dell'array in memoria

Array bidimensionali

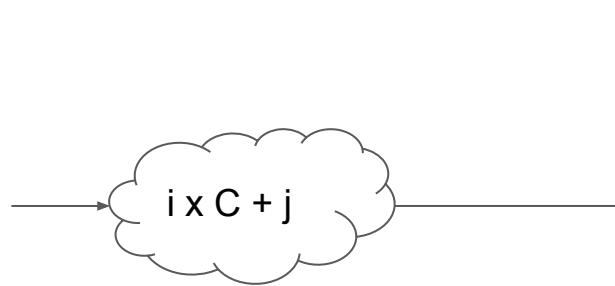
```
int M[3][2];  
M[2][1] = 4;
```

M[0][0]	M[0][1]	M[0][2]	M[1][0]	M[1][1]	M[1][2]
?	?	?	?	?	4

posizione reale

indici

(i,j)



con C numero di elementi che compongono i sottoarray (in questo caso, 2)

Matrici

Un array bidimensionale può essere utilizzato in maniera naturale per rappresentare una matrice

matematica

$$M = \begin{pmatrix} 45 & 2 \\ 7 & 1 \\ 52 & 66 \end{pmatrix}$$

C

```
int M[3][2] = {  
    { 45, 2 },  
    { 7, 1 },  
    { 52, 66 }  
};
```

memoria

M[0][0] M[0][1] M[1][0] M[1][1] M[2][0] M[2][1]

45	2	7	1	52	66
----	---	---	---	----	----

ogni sottovettore rappresenta una **riga** della matrice

- il primo indice rappresenta il numero di riga dell'elemento
- il secondo indice rappresenta il numero di colonna dell'elemento

Dato che **in C** gli elementi sono memorizzati per riga, la strategia di memorizzazione è detta **row-major order**

Row-major order

matematica *memoria*

$$M = \begin{pmatrix} 45 & 2 \\ 7 & 1 \\ 52 & 66 \end{pmatrix}$$

M[0][0] M[0][1] M[1][0] M[1][1] M[2][0] M[2][1]

45	2	7	1	52	66
----	---	---	---	----	----

Andiamo a leggere sequenzialmente gli elementi dell'array in memoria, e annotiamo i corrispondenti indici di matrice:

- | | |
|--------------|--------|
| 1° elemento: | (0, 0) |
| 2° elemento: | (0, 1) |
| 3° elemento: | (1, 0) |
| 4° elemento: | (1, 1) |
| 5° elemento: | (2, 0) |
| 6° elemento: | (2, 1) |

Row-major order

matematica

$$M = \begin{pmatrix} 45 & 2 \\ 7 & 1 \\ 52 & 66 \end{pmatrix}$$

memoria

M[0][0]	M[0][1]	M[1][0]	M[1][1]	M[2][0]	M[2][1]
45	2	7	1	52	66

Andiamo a leggere sequenzialmente gli elementi dell'array in memoria,
e annotiamo i corrispondenti indici di matrice:

- 1° elemento:
- 2° elemento:
- 3° elemento:
- 4° elemento:
- 5° elemento:
- 6° elemento:

(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)

l'indice di colonna
varia più velocemente
dell'indice di riga

il Row-major order è la strategia di memorizzazione degli array multidimensionali
in C

Column-major order

<i>matematica</i>		<i>memoria</i>																		
$M = \begin{pmatrix} 45 & 2 \\ 7 & 1 \\ 52 & 66 \end{pmatrix}$		<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>M[0][0]</td><td>M[1][0]</td><td>M[2][0]</td><td>M[0][1]</td><td>M[1][1]</td><td>M[2][1]</td></tr><tr><td>45</td><td>7</td><td>52</td><td>2</td><td>1</td><td>66</td></tr></table>							M[0][0]	M[1][0]	M[2][0]	M[0][1]	M[1][1]	M[2][1]	45	7	52	2	1	66
M[0][0]	M[1][0]	M[2][0]	M[0][1]	M[1][1]	M[2][1]															
45	7	52	2	1	66															

Andiamo a leggere sequenzialmente gli elementi dell'array in memoria, e annotiamo i corrispondenti indici di matrice:

- 1° elemento:
- 2° elemento:
- 3° elemento:
- 4° elemento:
- 5° elemento:
- 6° elemento:

↓
(0, 0)
↓
(1, 0)
↓
(2, 0)
↓
(0, 1)
↓
(1, 1)
↓
(2, 1)

l'indice di riga
varia più velocemente
dell'indice di colonna

il Column-major order è la strategia di memorizzazione degli array multidimensionali in altri linguaggi, come MATLAB o FORTRAN

Row-major VS Column-major

$$M = \begin{pmatrix} 45 & 2 \\ 7 & 1 \\ 52 & 66 \end{pmatrix}$$

Row-major

45	2	7	1	52	66
----	---	---	---	----	----

M[0][0] M[0][1] M[1][0] M[1][1] M[2][0] M[2][1]

Column-major

45	7	52	2	1	66
----	---	----	---	---	----

M[0][0] M[1][0] M[2][0] M[0][1] M[1][1] M[2][1]

- 1° elemento:
- 2° elemento:
- 3° elemento:
- 4° elemento:
- 5° elemento:
- 6° elemento:

(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)

l'indice di colonna
varia più velocemente
dell'indice di riga

(0, 0)
(1, 0)
(2, 0)
(0, 1)
(1, 1)
(2, 1)

l'indice di riga
varia più velocemente
dell'indice di colonna

- Se un linguaggio utilizza il **row-major order**, allora gli elementi in memoria saranno disposti in modo tale che, leggendo sequenzialmente gli elementi in memoria ed i relativi indici, la "velocità" di cambiamento degli indici aumenti *dalla prima dimensione all'ultima dimensione*.
- Se un linguaggio utilizza il **column-major order** allora gli elementi in memoria saranno disposti in modo tale che, leggendo sequenzialmente gli elementi in memoria ed i relativi indici, la "velocità" di cambiamento degli indici aumenti *dall'ultima dimensione alla prima dimensione*.

Array multidimensionali

E' possibile generalizzare ad un numero qualsiasi di dimensioni

```
int M[3][2]; // dichiara un array composto da 3 array, ognuno dei quali  
// è a sua volta composto da 2 int  
  
float T[4][3][5]; // dichiara un array composto da 4 array,  
// ognuno dei quali è a sua volta composto da 3 array,  
// ognuno dei quali è composto a sua volta da 5 array
```

Array multidimensionali

```
int K[3][4]; // array di 12 elementi distribuiti lungo 2 dimensioni  
int W[2][3][2]; // array di 12 elementi distribuiti lungo 3 dimensioni  
int Q[1][2][2][3]; // array di 12 elementi distribuiti lungo 4 dimensioni
```

Gli **indici** di un array vengono utilizzati per identificare un elemento in fase di lettura/scrittura, esattamente come in un sistema di coordinate.

```
K[2][0] = 5; // assegno il valore 5 alla posizione di indici (2,0)  
W[2][0][0] = 5; // assegno il valore 5 alla posizione di indici (2,0,0)  
Q[0][2][0][0] = 5; // assegno il valore 5 alla posizione di indici (0,2,0,0)
```

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

Come sono disposti in memoria?

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}};
```

Come sono disposti in memoria?

Row-major order: la velocità di cambiamento aumenta dalla prima all'ultima dimensione

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}}  
;
```

Come sono disposti in memoria?

Row-major order: la velocità di cambiamento aumenta dalla prima all'ultima dimensione

[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]	[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]	[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]	[1][0][0]	[1][0][1]	[1][0][2]	[1][0][3]	[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]	[1][2][0]	[1][2][1]	[1][2][2]	[1][2][3]
1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}}  
};
```

Può essere visto come un array composto da 2 matrici 3 x 4

[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]	[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]	[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]	[1][0][0]	[1][0][1]	[1][0][2]	[1][0][3]	[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]	[1][2][0]	[1][2][1]	[1][2][2]	[1][2][3]
1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:

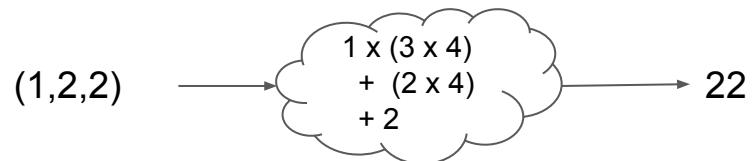


1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:

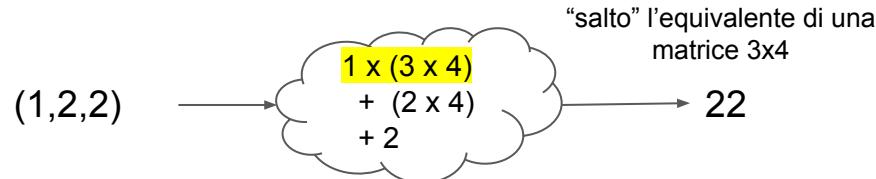


1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:

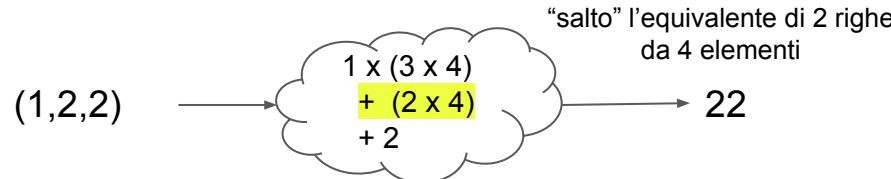


[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]	[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]	[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]	[1][0][0]	[1][0][1]	[1][0][2]	[1][0][3]	[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]	[1][2][0]	[1][2][1]	[1][2][2]	[1][2][3]
1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:

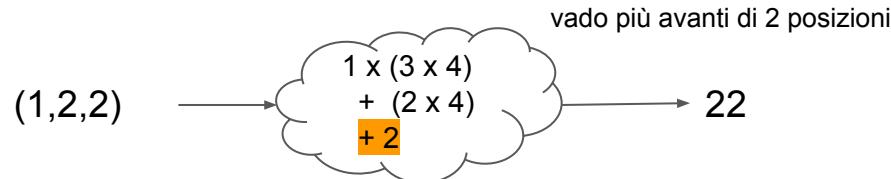


[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]	[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]	[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]	[1][0][0]	[1][0][1]	[1][0][2]	[1][0][3]	[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]	[1][2][0]	[1][2][1]	[1][2][2]	[1][2][3]
1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:

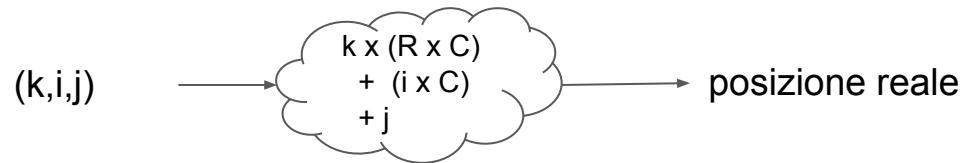


[0][0][0]	[0][0][1]	[0][0][2]	[0][0][3]	[0][1][0]	[0][1][1]	[0][1][2]	[0][1][3]	[0][2][0]	[0][2][1]	[0][2][2]	[0][2][3]	[1][0][0]	[1][0][1]	[1][0][2]	[1][0][3]	[1][1][0]	[1][1][1]	[1][1][2]	[1][1][3]	[1][2][0]	[1][2][1]	[1][2][2]	[1][2][3]
1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1

Array multidimensionali

```
int T[2][3][4] = {  
    {{1,5,7,9},  
     {2,4,6,8},  
     {5,6,2,3}},  
    {{3,1,8,4},  
     {6,2,4,7},  
     {5,9,4,1}}  
};
```

data una configurazione di indici, come calcolare la posizione reale?
Esempio:



[0][0][0] [0][0][1] [0][0][2] [0][0][3] [0][1][0] [0][1][1] [0][1][2] [0][1][3] [0][2][0] [0][2][1] [0][2][2] [0][2][3] [1][0][0] [1][0][1] [1][0][2] [1][0][3] [1][1][0] [1][1][1] [1][1][2] [1][1][3] [1][2][0] [1][2][1] [1][2][2] [1][2][3]

1	5	7	9	2	4	6	8	5	6	2	3	3	1	8	4	6	2	4	7	5	9	4	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Array sovradimensionati

La dimensione dei vettori automatici viene decisa nel codice sorgente, e quindi fissata a *compile time*. Spesso però si vorrebbe che la dimensione effettiva di tali vettori possa essere impostata durante la materiale esecuzione del programma, ossia a *run-time*.

Esempi:

- dato un vettore di 10 elementi, estrarre gli elementi pari ed inserirli in un altro vettore
 - non si può determinare la lunghezza del secondo vettore se non dopo una scansione del primo vettore a *run time*
- acquisire dall'utente un vettore di n interi con n *deciso dall'utente*
 - Impossibile sapere la lunghezza del vettore prima del run time in cui l'utente comunica quanti elementi inserire

Array sovradimensionati

In questi casi, le possibilità sono almeno due:

- utilizzo di allocazione dinamica della memoria (vedremo poi)
- utilizzare una *stima* della lunghezza

Esempi:

- se si vuole produrre un "sottovettore", la lunghezza sarà *al più pari alla lunghezza del vettore di partenza*. Si può quindi dichiarare un nuovo vettore di uguale lunghezza, *tenendo traccia di quanti elementi saranno effettivamente utilizzati*
- se la dimensione è decisa dall'utente, si può fare una stima di quanti elementi un utente potrebbe inserire (e.g., non più di 100), avvisandolo opportunamente di tale limite. Si può quindi dichiarare un vettore di 100 elementi, *tenendo traccia di quanti elementi saranno effettivamente utilizzati*

Array sovrdimensionati

Entrambi questi esempi fanno uso di vettori sovrdimensionati → necessaria quindi una ulteriore variabile che contenga quanti siano gli elementi realmente utilizzati.

```
int main()
{
    int V[6]; // dichiaro un vettore di 6 elementi
    int n;    // variabile di supporto contenente il numero di elementi effettivi
    printf("quanti elementi vuoi inserire nel vettore? (NB: MAX 6)");
    scanf("%d", &n);

    // l'accesso al vettore avviene soltanto sui primi n elementi
    for(int i=0; i < n; i++)
    {
        scanf("%d", &V[i]);
    }
    return 0;
}
```

V

5	1	2	4	?	?
---	---	---	---	---	---

n

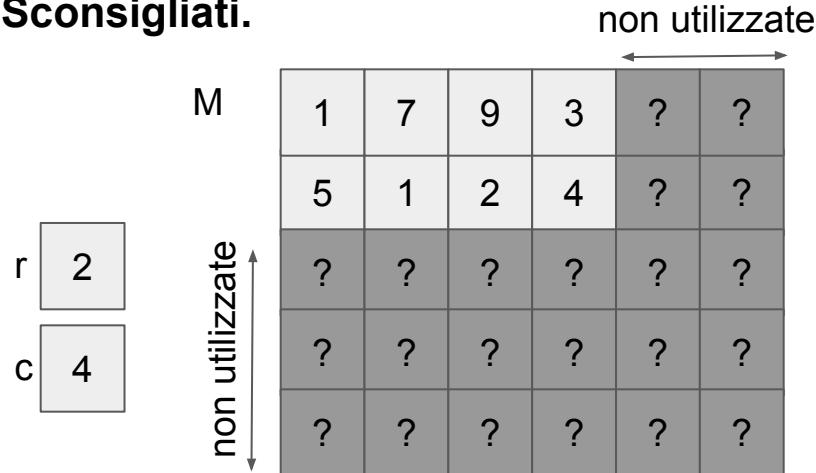
4

 non utilizzate

Array sovradianimensionati

- In alcuni testi, la variabile contenente il numero di elementi effettivi del vettore è detta *riempimento*. Sconsiglio di usare questo termine.
- Lo stesso discorso si può generalizzare al caso degli array multidimensionali. Ad esempio, nel caso delle matrici, si può chiedere all'utente di fornire il numero di righe ed il numero di colonne materialmente utilizzate, a fronte di una matrice allocata sovradianimensionata.
- Potreste aver sentito parlare degli Array di Lunghezza Variabile (*Variable Length Array*, VLA).

Non saranno trattati in questo corso. Sconsigliati.



Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Input e Output su file

Le funzioni per la gestione dei file sono dichiarate `stdio.h`.

2 tipologie di file:

- file **ASCII** (o di testo)
- file **binari**

Esempio di creazione di un file **binario**:

```
In [3]: #include <stdio.h>
int main()
{
    int v[] = {64, 77, 48, 92};
    FILE* fp;
    fp = fopen("vettore.bin", "wb");

    fwrite(v, sizeof(int), 4, fp);

    fclose(fp);

    return 0;
}
```

```
cat vettore.bin
```

```
@M0
```

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	'
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[END OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					

Source: www.LookupTables.com

Da terminale:

```
xxd -b vettore.bin
```

```
00000000: 01000000 00000000 00000000 00000000 01001101 00000000 | @...M.  
00000006: 00000000 00000000 00110000 00000000 00000000 00000000 | ..0...  
000000c: 01011100 00000000 00000000 00000000 | \...
```

- 00000000 00000000 00000000 01000000 è la codifica binaria di 64 su 4 byte
 - 00000000 00000000 00000000 01001101 è la codifica binaria di 77 su 4 byte
 - 00000000 00000000 00000000 00110000 è la codifica binaria di 48 su 4 byte
 - 00000000 00000000 00000000 01011100 è la codifica binaria di 92 su 4 byte

Esempio di creazione di un file di **testo**:

```
In [5]: #include <stdio.h>
int main()
{
    int v[] = {64, 77, 48, 92};
    FILE* fp;
    fp = fopen("vettore.txt", "w");
    for(int i=0; i<4; i++)
        fprintf(fp, "%d, ", v[i]);
    fclose(fp);
    return 0;
}
```

Da terminale:

```
cat vettore.txt
```

64 77 48 92

```
xxd -b vettore.txt
```

```
00000000: 00110110 00110100 00100000 00110111 00110111 00100000 | 64 77  
00000006: 00110100 00111000 00100000 00111001 00110010 00100000 | 48 92
```

- 00110110 è la codifica ASCII del 6
 - 00110100 è la codifica ASCII del 4
 - 00100000 è la codifica ASCII dello spazio
 - In un file di testo, le cifre di ogni numero vengono convertite nei *caratteri* ASCII corrispondenti prima di essere scritte, così da poter essere facilmente lette da un essere umano attraverso un editor di testo

- In un file binario, ogni valore è scritto nella sua codifica originaria, senza che sia effettuata alcuna trasformazione

Apertura e chiusura di un file

Le funzioni per la gestione dei file sono generalmente dichiarate (o comunque incluse) in `stdio.h`.

In generale, per aprire e chiudere un file si usano rispettivamente le funzioni `fopen(...)` ed `fclose(...)`.

`fopen(...)`

La funzione `fopen(...)` ha il seguente prototipo:

```
FILE* fopen ( const char * filename, const char * mode );
```

Tale funzione accetta due parametri: il nome del file (stringa), e la *modalità di accesso* (stringa).

La modalità di accesso permette di stabilire due cose:

1. se vogliamo leggere/scrivere sul file
2. se vogliamo leggere o scrivere un file in modalità binaria o testuale

mode (text)	mode (binary)	descrizione
"r"	"rb"	read : Apre il file in lettura (input). Il file deve esistere, altrimenti restituisce NULL.
"w"	"wb"	write : Crea un file in scrittura (output). Se il file già esiste, il suo contenuto viene eliminato.
"a"	"ab"	append : Crea un file in scrittura (output). Se il file già esiste, la scrittura partirà dalla fine del file. Repositioning operations (<code>fseek</code> , <code>fsetpos</code> , <code>rewind</code>) are ignored.
"r+"	"r+b"	read/update : Apre il file sia in lettura che scrittura. Il file deve esistere, altrimenti restituisce NULL.
"w+"	"w+b"	write/update : Crea un file per lettura/scrittura. Se il file già esiste, il contenuto viene eliminato.
"a+"	"a+b"	append/update : Crea un file in lettura/scrittura. Se il file già esiste, eventuali operazioni di scrittura verranno effettuate a partire dalla fine del file. Repositioning operations (<code>fseek</code> , <code>fsetpos</code> , <code>rewind</code>) affects the next input operations, but output operations move the position back to the end of file.

Tale funzione restituisce un puntatore di tipo `FILE`.

`FILE` : [struct] that identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator and all its state indicators.

Tale struttura è definita (o comunque inclusa) in `stdio.h`.

Generalmente la struttura FILE di un dato file è chiamata anche *file handler*.

```
struct _IO_FILE
{
    int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */

    /* The following pointers correspond to the C++ streambuf protocol. */
    char *_IO_read_ptr;   /* Current read pointer */
    char *_IO_read_end;   /* End of get area. */
    char *_IO_read_base;  /* Start of putback+get area. */
    char *_IO_write_base; /* Start of put area. */
    char *_IO_write_ptr;  /* Current put pointer. */
    char *_IO_write_end;  /* End of put area. */
    char *_IO_buf_base;   /* Start of reserve area. */
    char *_IO_buf_end;    /* End of reserve area. */

    /* The following fields are used to support backing up and undo. */
    char *_IO_save_base; /* Pointer to start of non-current get area. */
    char *_IO_backup_base; /* Pointer to first valid character of backup area */
    char *_IO_save_end;   /* Pointer to end of non-current get area. */

    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;
    int _flags2;
    _off_t _old_offset; /* This used to be _offset but it's too small. */

    /* 1+column number of pbase(); 0 is unknown. */
    unsigned short _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];
}
```

```
_IO_lock_t * _lock;  
#ifdef _IO_USE_OLD_IO_FILE  
};
```

```
typedef struct _IO_FILE FILE;
```

I campi di questa struttura saranno utilizzati dalle funzioni di scrittura/lettura per accedere materialmente al file.

fclose(...)

Una volta che il file non deve più essere utilizzato, è buona norma chiuderlo.

La chiusura di un file può essere effettuata attraverso la funzione `fclose(...)`

- `int fclose(FILE *stream)`

esempio:

```
// apro il file  
FILE* file_handler = fopen(...);  
// effettuo operazione sul file...  
  
// terminate le operazioni sul file, chiudo il file  
fclose(file_handler);
```

è buona norma chiudere **sempre** un file una volta finito di utilizzarlo. Sia per liberare risorse (e.g., la struct `FILE` occupa spazio in memoria) sia per permettere di finalizzare eventuali operazioni sospese (e.g., le operazioni di scrittura sul file non è detto che avvengano all'atto dell'invocazione della funzione di scrittura).

Se l'operazione va a buon fine, `fclose(...)` restituisce 0.

File Binari

Scrittura e lettura in un file binario

- `fwrite (const void* ptr, size_t size, size_t nitems, FILE * stream);`

La funzione `fwrite(...)` scrive, sul file puntato da `stream`, (fino a) `nitems` elementi di dimensione `size` (in Byte), a partire dall'indirizzo specificato in `ptr`. Tale funzione restituisce il numero di elementi correttamente scritti. Se tutto è andato bene, tale valore sarà pari a `nitems`.

Otherwise, if a write error occurs, the error indicator for the stream is set and `errno` is set to indicate the error.

- `fread(void *ptr, size_t size, size_t nitems, FILE *stream);`

La funzione `fread(...)` legge, dal file puntato da `stream`, (fino a) `nitems` di dimensione `size` (in Byte). La funzione restituisce il numero di elementi correttamente letti. Tale valore sarà < `nitems` solo se c'è un errore oppure il file è finito prima di riuscire a leggere `nitems` elementi.

In []:

In [1]:

```
#include <stdio.h>  
#define MAX_LEN 100  
  
void stampa_vett(int v[], int n)  
{  
    printf("(" );  
    for(int i=0; i<4; i++)  
        printf("%d, ", v[i]);  
    printf("\b\b)\n");  
}  
  
int main()  
{  
    int v[] = {64, 77, 48, 92};  
    FILE* fp;  
    fp = fopen("vettore.bin","wb");  
  
    int n_scritti = fwrite(v, sizeof(int), 4, fp);  
  
    fclose(fp);  
  
    printf("scritto: ");  
    stampa_vett(v, n_scritti);  
  
    // lettura
```

```

fp = fopen("vettore.bin","rb");
int v2[MAX_LEN];
int n2 = 0;

int readed = 1;
while(readed >=1)
{
    readed = fread(&v2[n2], sizeof(int),1, fp);
    n2++;
}

fclose(fp);

printf("letto: ");
stampa_vett(v2, n2);

return 0;
}

```

scritto: (64, 77, 48, 92)
letto: (64, 77, 48, 92)

L'esempio precedente non sfrutta appieno la funzione `fread(...)` in quanto:

- necessita di più invocazioni, una per ogni elemento da leggere dal file
- basandosi unicamente sul numero di valori letti dalla funzione `fread(...)` per terminare, non permette di avere nello stesso file più strutture diverse (e.g., più array) memorizzati nello stesso file

Possibile soluzione: Salvare all'interno del file *informazioni addizionali* in testa, ad esempio la lunghezza dell'array

```

In [2]: #include <stdio.h>
#define MAX_LEN 100

void stampa_vett(int v[], int n)
{
    printf("( ");
    for(int i=0; i<4; i++)
        printf("%d, ", v[i]);
    printf("\b\b)\n");
}

int main()
{
    int v[] = {64, 77, 48, 92};
    int n    = 4;
    FILE* fp;
    fp = fopen("vettore.bin","wb");

    // scrivo il numero di elementi dell'array in testa al file
    fwrite(&n, sizeof(int), 1, fp);

    // provo a scrivere n*sizeof(int) Bytes nel file
    int n_scritti = fwrite(v, sizeof(int), n, fp);

    fclose(fp);

    printf("scritto: ");
    stampa_vett(v, n_scritti);

    // lettura
    fp = fopen("vettore.bin","rb");
    int v2[MAX_LEN];
    int n2 = 0;
    // leggo numero di elementi dell'array in testa al file
    fread(&n2, sizeof(int), 1, fp);

    // leggo n2 elementi (ossia n2*sizeof(int) Bytes)
    fread(v2, sizeof(int), n2, fp);

    fclose(fp);

    printf("letto:   ");
    stampa_vett(v2, n2);

    return 0;
}

scritto: ( 64, 77, 48, 92)
letto:   ( 64, 77, 48, 92)

```

File di testo

Scrittura e lettura in un file di testo

Diverse funzioni possibili:

- `int fputc(int c, FILE *fp);` Scrive il carattere con codifica ASCII `c` sul file gestito da `fp`. In caso di successo, restituisce il carattere scritto.
- `int fputs(const char *s, FILE *fp);` The function fputs() writes the string s to the output stream referenced by fp. It returns a non-negative value on success.
- `int fprintf(FILE *restrict stream, const char *restrict format, ...);` come `printf(...)`, ma su file di testo.
- `int fgetc(FILE * fp);` legge un carattere dal file gestito da `fp` e ne restituisce la codifica ASCII.
- `char *fgets(char *buf, int n, FILE *fp);` legge al più $n - 1$ caratteri dal file gestito da `fp`. Copia la stringa letta nell'area di memoria puntata da `buf`, aggiungendo un terminatore di stringa (ossia, dopo l'ultimo carattere letto, viene inserito nel buffer il carattere '0'). Se nella stringa letta c'è un `\n` oppure il file termina prima di leggere $n - 1$ caratteri, allora `buf` conterrà solo i caratteri letti fino a quel punto, incluso il `\n`. Se funzione viene completata con successo, allora il valore di ritorno sarà lo stesso di `buf`.
- `int fscanf(FILE *restrict stream, const char *restrict format, ...);` Come `scanf(...)`, ma legge dal file di testo gestito da `stream`. NB: ricordate però come la `scanf(...)` si comporta in presenza di spazi...

In [22]: // esempio di utilizzo di fputs(...)

```
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    int rit = fputs("ciao pluto", fp);
    printf("fputs: valore di ritorno: %d\n", rit);

    fclose(fp);

    // apertura in lettura
    char str[30];
    fp = fopen("stringa.txt","r");
    char* str_rit = fgets(str,30, fp);
    // lettura
    printf("fgets: *str_rit:%s, *str:%s\n", str_rit, str);
    printf("fgets: str_rit: %p, str:%p\n", str_rit, str);
    printf("fgets: valore di ritorno: %d\n", rit);

    fclose(fp);

    return 0;
}
```

fputs: valore di ritorno: 1
fgets: *str_rit:ciao pluto, *str:ciao pluto
fgets: str_rit: 0x7ffec8771370, str:0x7ffec8771370
fgets: valore di ritorno: 1

Cosa succede nel caso di lettura di un file terminato (ossia in cui già è stato letto tutto e non c'è più nulla da leggere)?

Iniziamo con la funzione `fgets(...)`. Tale funzione, in caso tentativo di lettura da file terminato, restituisce `NULL`. Ma attenzione...

In [2]: // attenzione...

```
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    int rit = fprintf(fp, "ciao Pluto come stai?\nBene grazie");
```

```

printf("fprintf: valore di ritorno: %d\n", rit);
fclose(fp);

// apertura in lettura
fp = fopen("stringa.txt","r");

// lettura
char str[30];
rit = fscanf(fp, "%s", str);

printf("fscanf: *str:%s\n", str);
printf("fscanf: valore di ritorno: %d\n", rit);

// prima invocazione di fgets
char* rit_str = fgets(str,30, fp);
printf("(1) fgets: *str:%s\n", str);
printf("(1) fgets: valore di ritorno: %p\n", rit_str);
// seconda invocazione di fgets
rit_str = fgets(str,30, fp);
printf("(2) fgets: *str:%s\n", str);
printf("(2) fgets: valore di ritorno: %p\n", rit_str);
// terza invocazione di fgets
rit_str = fgets(str,30, fp);
printf("(3) fgets: *str:%s\n", str);
printf("(3) fgets: valore di ritorno: %p\n", rit_str);
fclose(fp);

return 0;
}

```

```

fprintf: valore di ritorno: 33
fscanf: *str:ciao
fscanf: valore di ritorno: 1
(1) fgets: *str: Pluto come stai?

(1) fgets: valore di ritorno: 0x7ffe275df630
(2) fgets: *str:Bene grazie
(2) fgets: valore di ritorno: 0x7ffe275df630
(3) fgets: *str:Bene grazie
(3) fgets: valore di ritorno: (nil)

```

Intuitivamente, dato che il file `stringa.txt` contiene due righe (quindi due `\n`) e dato che ogni invocazione di `fgets(...)` prende una riga (ossia fino al `\n`), mi sarei aspettato che il valore di ritorno della **seconda** invocazione di `fgets(...)` restituisse `NULL`. Invece ciò avviene soltanto dopo la **terza** invocazione. Questo perché questa funzione (e le funzioni di lettura in generale) impostano lo stato di fine-file soltanto quando materialmente viene fatta una lettura "a vuoto". Dato che le prime due letture vanno entrambe a buon fine (la prima per la prima frase, i.e. `Pluto come stai?`, e la seconda per la seconda frase, i.e. `Bene grazie`) non viene rilevato lo stato di fine-file. E' quindi necessaria una lettura ulteriore affinché `fgets(...)` non riesca a leggere nulla (in quanto il file è finito) e restituisca materialmente `NULL`, lasciando il contenuto di `str` inalterato *rispetto alla lettura precedente*.

Non tenere conto di ciò potrebbe portare ad una implementazione ingenua come la seguente:

```

In [45]: // Leggere un file di testo e stampare a video tutte le righe presenti (implementazione errata)

#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt","r");
    char str[30];
    char* rit_str = str;
    while(rit_str != NULL)
    {

        rit_str = fgets(str,30, fp);
        printf("fgets: %s\n", str);
    }
    fclose(fp);
}

```

```
    return 0;
}

fgets: ciao Pluto come stai?

fgets: Bene grazie
fgets: Bene grazie
```

Il codice di sopra stampa due volte l'ultima riga. Questo perché `rit_str` diventerà `NULL` solo quando `fgets(...)` non riuscirà a leggere materialmente nulla dal file in quanto terminato (lasciando quindi `str` inalterata), e ciò avverrà solo alla terza iterazione (e non alla seconda).

Possibile soluzione:

```
In [4]: // Leggere un file di testo e stampare a video tutte le righe presenti (soluzione 1)
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt", "w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt", "r");
    char str[30];
    char* rit_str = str;
    while(rit_str != NULL)
    {

        rit_str = fgets(str, 30, fp);
        if(rit_str != NULL)
            printf("fgets: %s\n", str);
    }
    fclose(fp);

    return 0;
}

fgets: ciao Pluto come stai?
```

```
fgets: Bene grazie
```

Oppure...

```
In [5]: // Leggere un file di testo e stampare a video tutte le righe presenti (soluzione 2)
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt", "w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt", "r");
    char str[30];

    while(fgets(str, 30, fp) != NULL) // controllo il valore di ritorno stesso all'interno del ciclo
    {
        printf("fgets: %s\n", str);
    }
    fclose(fp);

    return 0;
}
```

```
fgets: ciao Pluto come stai?
```

```
fgets: Bene grazie
```

The curse of *End Of File*

Quando una funzione prova a leggere un file *terminato*, viene segnalata **internamente** (ad esempio attraverso un flag nel file handler) una condizione di *end-of-file*. A prescindere da questo, ogni funzione adotta una politica differente nel caso in cui si provi ad accedere ad un file terminato. Le principali sono:

function In caso di end-of-file (o errore) restituisce

fgets(...) NULL

fscanf(...) numero di conversioni effettuate minore di quanto atteso; In the case of an input failure before any data could be successfully read, EOF is returned.

fgetc(...) EOF

fread(...) numero di elementi letti minore di quanto atteso

EOF è una macro definita in `stdio.h` (o in qualche file `.h` da quest'ultimo a sua volta incluso).

NB: EOF ≠ end-of-file ;

- EOF è una macro definita da qualche parte (e.g., `#define EOF -1` in `stdio.h` o in file da questo inclusi). E' un valore che è restituito da *alcune* (e non tutte!) le funzioni della libreria standard quando si prova a leggere da un file terminato. Generalmente il suo valore è `-1`, o comunque un valore `< 0`.
- *end-of-file* è un possibile *stato* del file, generalmente identificato da un apposito flag nel file handler. Tale flag nasce per dire se il file è terminato oppure no. Il valore di tale flag viene generalmente impostato quando le funzioni di accesso al file (e.g., `fread(...)`, `fgets(...)`, ecc.) provano a leggere da un file già *terminato* in precedenza. L'accesso in lettura a tale flag (ossia sapere quale valore contiene in un dato istante) è possibile tramite la funzione `feof(...)`.

int feof(FILE *stream) :

controlla se, a seguito dell'ultima lettura effettuata, il file è terminato, ossia è stato impostato lo *stato end-of-file*.

In altri termini, `feof(fp)` restituisce un valore $\neq 0$ se l'indicatore *end-of-file* associato al file-handler `*fp` è impostato, altrimenti restituisce 0.

SCONSIGLIATA!

Esempio:

contenuto file `testo.txt`:

```
topolino
paperino
```

In [70]: // lettura di un file di testo e stampa a video di ogni riga (versione errata!)

```
#include <stdio.h>
int main()
{
    // apertura in lettura

    FILE* fp = fopen("testo.txt","r");
    char str[30];
    while(feof(fp) == 0)
    {

        fgets(str,sizeof(str), fp);
        printf("fgets: %s\n", str);
    }
    fclose(fp);

    return 0;
}
```

```
fgets: topolino
```

```
fgets: paperino
```

```
fgets: paperino
```

- 1° invocazione di `fgets(...)` : legge `topolino` (ma non imposta lo stato *end-of-file*)
- 2° invocazione di `fgets(...)` : legge `paperino` (ma non imposta lo stato *end-of-file*)

- 3° invocazione di `fgets(...)` : il file è terminato quindi non legge nulla, lascia `str` inalterato ed imposta lo stato di *end-of-file*

In [72]: // lettura di un file di testo e stampa a video di ogni riga (versione corretta)

```
#include <stdio.h>
int main()
{
    // apertura in lettura

    FILE* fp = fopen("testo.txt", "r");
    char str[30];
    while(fgets(str, sizeof(str), fp) != NULL)
    {

        printf("fgets: %s\n", str);
    }
    fclose(fp);

    return 0;
}
```

fgets: topolino

fgets: paperino

In [6]: // lettura di un file di testo e stampa a video di ogni parola attraverso funzione fscanf // (versione corretta)

```
#include <stdio.h>
int main()
{
    // apertura in lettura

    FILE* fp = fopen("testo.txt", "r");
    char str[30];
    while(fscanf(fp, "%s", str) > 0)
    {

        printf("fscanf: %s\n", str);
    }
    fclose(fp);

    return 0;
}
```

fscanf: topolino
fscanf: paperino

Consiglio: non utilizzare la funzione `feof(...)`, utilizzare invece in maniera corretta i valori di ritorno delle diverse funzioni di accesso ai file.

STDIN/STDOUT/STDERR

definite in `<stdio.h>`

- `FILE* stdin;` ogni lettura da tastiera viene vista come una lettura dal file definito dalla struct (puntata da) `stdin`
- `FILE* stdout;` ogni scrittura sul file definito dalla struct (puntata da) `stdout` viene vista come una scrittura sul monitor
- `FILE *stderr;` associato tipicamente con il monitor ed è utilizzato per visualizzare messaggi di errore.

Alcune funzioni, ad esempio `getchar(...)` e `putchar(...)`, usano automaticamente `stdin` e `stdout`

```
getchar(...) = getc(stdin,...)
printf(...) = fprintf(stdout,...)
scanf(...) = fscanf(stdin,...)
```

per stampare un messaggio su `stderr`, si può usare la funzione

```
void perror(const char *s);
```

In [3]: // Esempio di modifica dell' stdout (sconsigliato)...

```
#include <stdio.h>
int main()
{

    FILE* original_stdout = stdout;
    stdout = fopen("new_stdout.txt", "w");
    printf("ciao\n");
    fclose(stdout);
```

```
    stdout = original_stdout;
    printf("a tutti\n");
    return 0;
}
```

a tutti

Avendo cambiato lo `stdout`, la parte iniziale del messaggio si troverà nel file `new_stdout.txt`.

Altre funzioni per l'accesso ai file

- `void rewind (FILE * stream);`

Riavvolge il file all'inizio

Funzioni per muoversi lungo il file:

- `long int ftell (FILE * stream);`
- `int fseek (FILE * stream, long int offset, int origin);`
- `int fsetpos (FILE * stream, const fpos_t * pos);`
- `int fgetpos (FILE * stream, fpos_t * pos);`

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

I puntatori

Il concetto di **object**

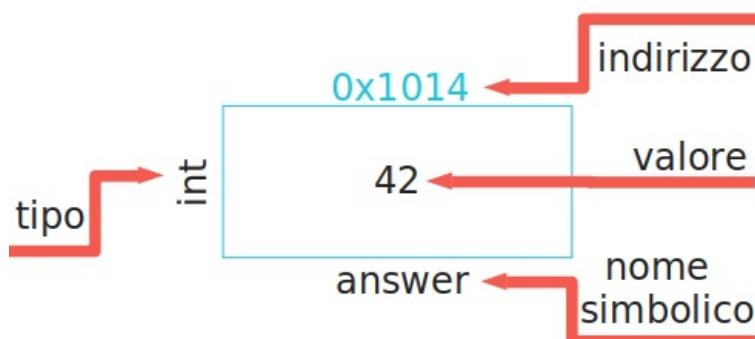
object: region of data storage in the execution environment, the contents of which can represent values

(From: C committee draft www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

NESSUNA relazione con la *Programmazione ad Oggetti*

Quindi in C, una regione di memoria contenente dati è chiamata genericamente **object** (variabili, array, etc.)

```
int answer = 42;
```



I puntatori

Una variabile puntatore è una variabile in grado di contenere l' *indirizzo* di un object

```
int* punt; → dichiarazione di una variabile puntatore di nome punt di tipo puntatore ( ) a int`*
```



Tale variabile puntatore può contenere al suo interno un indirizzo di un object di tipo `int` ;

```
punt = &answer; → inserisce in punt l'indirizzo dell'object answer
```

l'operatore `&` significa *indirizzo di ...*



ovviamente potevo anche fare direttamente `int* punt = &answer;`

è possibile adesso riferirsi al *contenuto* dell'object `answer` attraverso `punt` utilizzando l'operatore `*`

```
In [1]: #include <stdio.h>
int main()
{
    int answer = 42;
    int* punt = &answer;

    printf("answer: %ld\n", answer);
    printf("punt (hex): %p dec: %ld\n", punt, punt);
    printf("*punt: %ld\n", *punt);

    answer = answer + 10;

    printf("*punt: %ld\n", *punt);

    *punt = *punt + 20;

    printf("*punt: %ld\n", *punt);

    *punt = punt + 2;
    printf("*punt: %ld\n", *punt);
    printf("answer: %ld\n", answer);
    return 0;
}
```

```
/tmp/tmp8i0l7x4t.c: In function 'main':
/tmp/tmp8i0l7x4t.c:19:11: warning: assignment to 'int' from 'int *' makes integer from pointer without a cast [-Wint-conversion]
  19 |     *punt = punt + 2;
     |           ^
answer: 42
punt (hex): 0x7fff74e7749c dec: 140735154713756
*punt: 42
*punt: 52
*punt: 72
*punt: 1961325732
answer: 1961325732
```

Attenzione all'operatore `*` (1)

Notiamo che l'operatore `*` può acquisire due significati differenti in base alla fase in cui si trova:

1. in fase di *dichiarazione* (`int* punt;`): dichiara la variabile `punt` di tipo *puntatore a int*. In questa fase, l'operatore `*` è **sempre** preceduto dal tipo dell'object di cui il puntatore può contenere l'indirizzo (in questo caso `int`)
2. in fase di *deferenziavione* (e.g., `printf("%d", *punt)` o `int copy=*punt`): con `*` si desidera *accedere* al valore dell'object il cui indirizzo è contenuto nel puntatore `punt` (quindi già dichiarato in precedenza)

La posizione dell'operatore `*` in fase di dichiarazione è ininfluente, ossia:

```
int* punt;   = int * punt;   = int *punt;

int* punt=&answer; = int * punt=&answer; = int *punt=&answer;
```

NB: `int *punt=&answer;` significa *dichiara una variabile punt di tipo 'int ed inserisci al suo interno l'indirizzo dell'object answer'.*

Un altro modo di leggere `int *punt;` è il seguente: *dichiara una variabile di nome punt tale che la combinazione 'punt sia un int'.*

```
int *punt=&answer;   ≠ *punt=&answer;
```

Non fatevi ingannare dal `*` attaccato al nome dell'object! Ricordatevi che in questo caso siete in fase di **dichiarazione** (semplificando, quando alla sua sinistra c'è un tipo).

Attenzione all'operatore `*` (2)

Nei testi la sintassi più utilizzata per la dichiarazione di un puntatore è `tipo *nome`, e.g. `int *punt;` invece che `int* punt;`. L'utilizzo della sintassi `tipo *nome` permette di evitare ambiguità in caso di dichiarazioni multiple.

Esempio:

`int* p, q;` dichiara la variabile `p` di tipo *puntatore a int*, mentre la variabile `q` sarà di tipo `int` ma **non** puntatore (in sostanza, l'operatore `*` viene applicato solo al primo object, e non agli altri).

Per avere entrambe le variabili di tipo puntatore a int: `int* p,*q;`, oppure `int * p, * q;`, oppure...

```
int* p;  
int* q;
```

Attenzione all'operatore `&`

Potreste trovare del codice con istruzioni del tipo:

```
int& r = q; oppure int & r = q; oppure int& r = q;
```

Se trovate una istruzione del genere, allora non state avendo a che fare con il linguaggio C, ma con in linguaggio **C++**.

In C++ l'operatore `&`, se posto di fianco ad un tipo in fase di dichiarazione non è più operatore *indirizzo*, ma di **aliasing**.

L' *aliasing* non esiste in C! E' una caratteristica del C++, e come tale non oggetto di questo corso!

Buone norme di puntamento

In C, la dichiarazione di un puntatore non gli assegna automaticamente un valore!

→ Evitare di dichiarare puntatori senza inizializzarli!

```
int* p;
```

In tal caso `p`, se allocato nel call stack, potrebbe contenere al suo interno qualsiasi valore. Una sua eventuale deferenziazione `*p = ...` per assegnargli un valore può essere pericolosa (si pensi se casualmente `p` contenesse l'indirizzo di memoria di una object `const ...`). Generalmente, molti bugs nascono da una cattiva/mancata inizializzazione dei puntatori.

Se non si conosce ancora quale sarà l'object a cui dovrà puntare `p` o non è stato ancora dichiarato, meglio inizializzarlo con `NULL`. `NULL` è una macro definita nell' header `stdio.h` corrispondente all'indirizzo `0`.

```
int* p = NULL;
```

In C è garantito che nessun object esiste né esisterà all'indirizzo `NULL`. Attraverso questa convenzione, è possibile controllare se un puntatore punta o meno ad un oggetto (e.g., `if(p == NULL){...}`).

Ciò non toglie che deferenziare un puntatore che (non) punta a null(a) è concettualmente sbagliato, anche se sintatticamente corretto

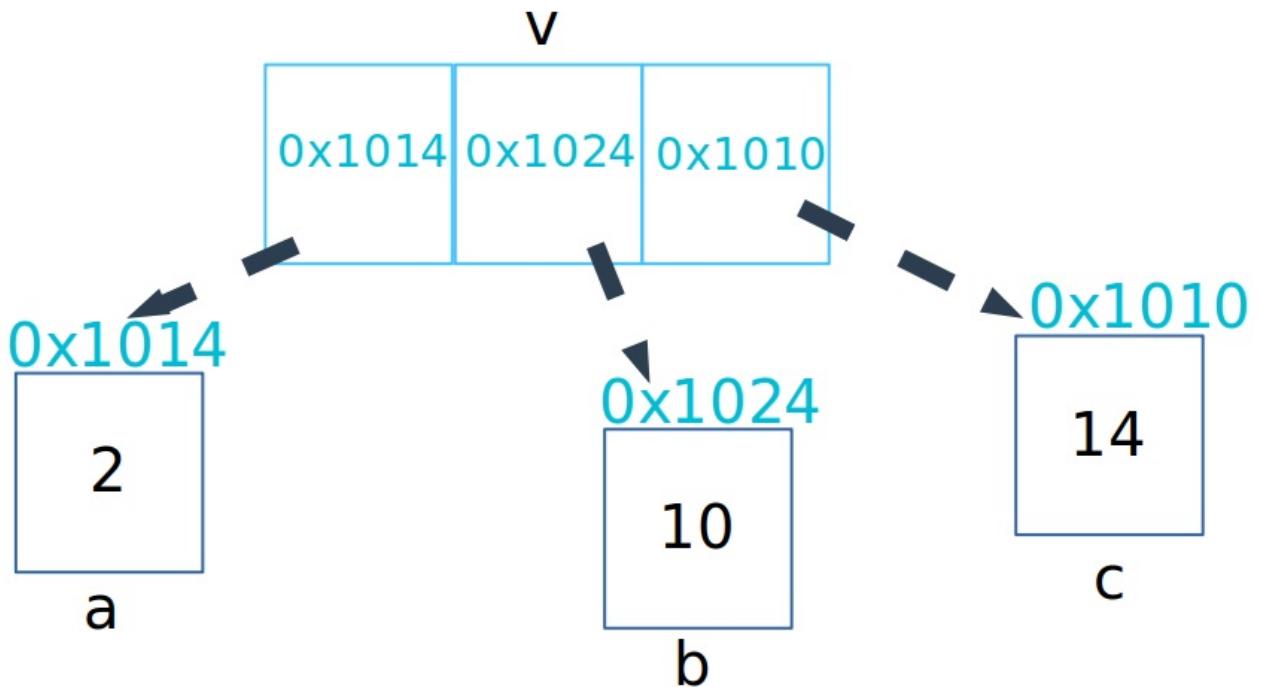
```
int* p = 0;  
*p = 10; //sbagliato! ma nessun errore da parte del compilatore
```

```
In [ ]:  
#include <stdio.h>  
int main()  
{  
    int *p;  
    printf("puntatore p non inizializzato. Valore:%p\n", p);  
  
    int *q = NULL;  
    printf("puntatore q inizializzato. Valore:%p, corrispondente al decimale: %lld \n", q, q);  
  
    return 0;  
}
```

puntatore p non inizializzato. Valore:0x7ffe4419bf40

puntatore q inizializzato. Valore:(nil), corrispondente al decimale: 0

C arrays OF pointers



```
In [2]: #include <stdio.h>
int main()
{
    int* v[3];
    int a = 2;
    int b = 10;
    int c = 14;
    v[0] = &a;
    v[1] = &b;
    v[2] = &c;
    printf("Prima:\n");
    printf("a: %d, b: %d, c: %d\n", a,b,c);
    printf("*v[0]: %d, *v[1]: %d, *v[2]:%d\n", *v[0],*v[1],*v[2]);

    a++;
    b = b + 2;
    c = a + b;
    printf("dopo:\n");

    printf("*v[0]: %d, *v[1]: %d, *v[2]:%d\n", *v[0],*v[1],*v[2]);
}
```

```
Prima:
a: 2, b: 10, c: 14
*v[0]: 2, *v[1]: 10, *v[2]:14
dopo:
*v[0]: 3, *v[1]: 12, *v[2]:15
```

C arrays AS pointers (?)

Except when it is the operand of the sizeof operator or the unary & operator, or is a string literal used to initialize an array, an expression that has type “array of type” is converted to an expression with type “pointer to type” that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined. (C standard, <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>)

Il nome di un array può essere utilizzato, nella maggioranza delle volte, come se fosse un puntatore costante (ossia il cui indirizzo memorizzato non può essere modificato). Tale “puntatore” punta al primo elemento di un array

```
In [41]: #include <stdio.h>
int main()
{
    int v[3] = {3, 10, 42};
    printf("primo valore del vettore v visto come array: %d\n", v[0]);
    printf("primo valore del vettore v visto come puntatore: %d\n", *v);
    printf("indirizzo usando v: %p; indirizzo usando &v[0]: %p\n", v, &v[0]);
    return 0;
}
```

```
primo valore del vettore v visto come array: 3
primo valore del vettore v visto come puntatore: 3
indirizzo usando v: 0x7ffe03a79e2c; indirizzo usando &v[0]: 0x7ffe03a79e2c
```

Una delle eccezioni è l'operatore `sizeof` che ha comportamenti differenti

```
In [42]: #include <stdio.h>
int main()
{
    int v[] = {3, 10, 42};
    int *p_v = v;

    printf("v: %p; p_v: %p; &v: %p; p_v: %p\n", v, p_v, &v, &p_v);
    printf("sizeof(v): %d Byte\n", sizeof(v));
    printf("sizeof(p_v): %d Byte\n", sizeof(p_v));
    return 0;
}
```

```
v: 0x7ffed829286c; p_v: 0x7ffed829286c; &v: 0x7ffed829286c; p_v: 0x7ffed8292860
sizeof(v): 12 Byte
sizeof(p_v): 8 Byte
```

pur puntando allo stesso vettore, l'operatore `sizeof` applicato al *nome* di un array restituisce la dimensione effettiva dell'array, mentre se applicato ad un puntatore che punta ad un array restituisce la dimensione del puntatore.

Questo implica che un array non è mai copiato quando passato come argomento di una funzione, mentre ciò che viene materialmente passato è il solo indirizzo al suo primo elemento.

Se però un array viene passato come parametro ad una funzione, l'array viene trattato esattamente come un puntatore, *indipendentemente se questo è passato come array o come puntatore*.

There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so `pa=a` and `pa++` are legal. But an array name is not a variable; constructions like `a=pa` and `a++` are illegal. (from: Expert C Programming)

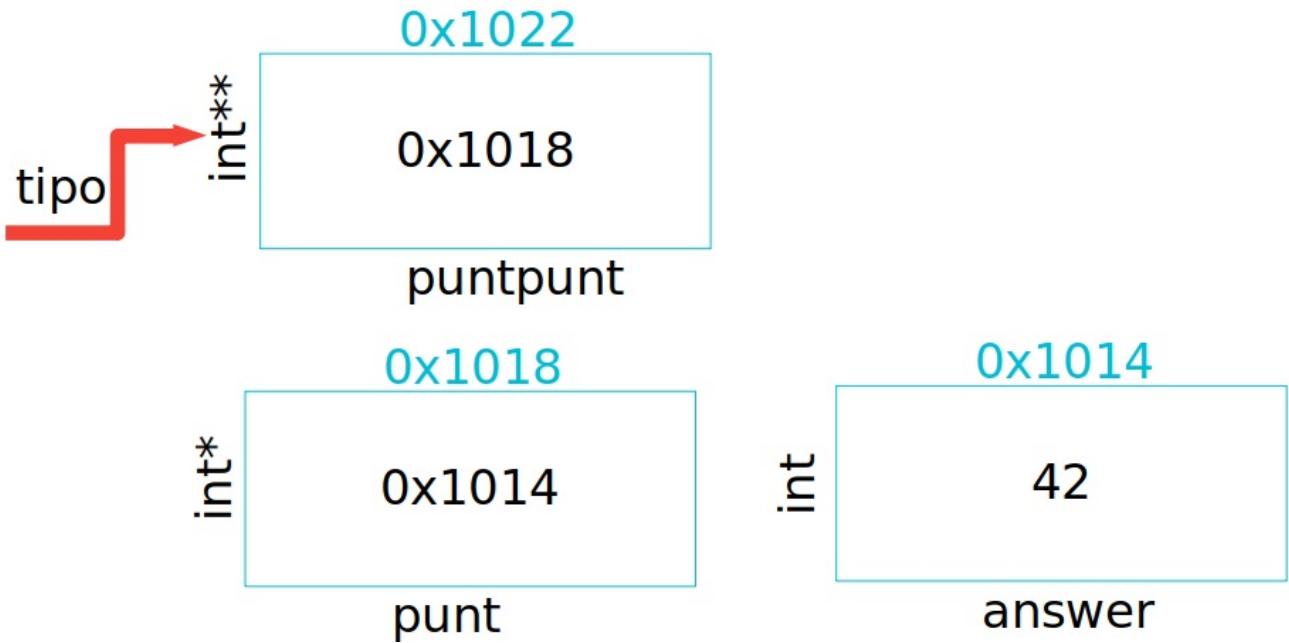
```
In [43]: #include <stdio.h>
void f(int f_v[]) // equivalente a void f(int* f_v)
{
    printf("sizeof(f_v): %d Byte\n", sizeof(f_v));
}

int main()
{
    int v[] = {3, 10, 42};
    int *p_v = v;
    printf("sizeof(v): %d Byte\n", sizeof(v));
    printf("sizeof(p_v): %d Byte\n", sizeof(p_v));
    f(v);
    return 0;
}
```

```
/tmp/tmpexfqqp0b.c: In function 'f':
/tmp/tmpexfqqp0b.c:4:44: warning: 'sizeof' on array function parameter 'f_v' will return size of 'int *' [-Wsize-of-array-argument]
  4 |         printf("sizeof(f_v): %d Byte\n", sizeof(f_v));
   |
/tmp/tmpexfqqp0b.c:2:12: note: declared here
  2 | void f(int f_v[]) // equivalente a void f(int* f_v)
   |         ~~~~^~~~~~
sizeof(v): 12 Byte
sizeof(p_v): 8 Byte
sizeof(f_v): 8 Byte
```

Pointers to pointers

```
int answer      = 42;
int* punt      = &answer;
int** punt_punt = &punt;
```



```
In [29]: #include <stdio.h>
int main()
{
    int answer      = 42;
    int* punt      = &answer;
    int** puntpunt = &punt;
    printf("answer:      %d\n", answer);
    printf("&answer(hex): %p\n", &answer);
    printf("=====\n");
    printf("punt:        %p\n", punt);
    printf("*punt:       %d\n", *punt);
    printf("&punt(hex): %p\n", &punt);
    printf("=====\n");
    printf("puntpunt:    %p\n", puntpunt);
    printf("**puntpunt: %p\n", *puntpunt);
    printf("**puntpunt: %d\n", **puntpunt);
    printf("&puntpunt(hex):%p\n", &puntpunt);

    return 0;
}

answer:      42
&answer(hex): 0x7fffc79a7424
=====
punt:        0x7fffc79a7424
*punt:       42
&punt(hex): 0x7fffc79a7428
=====
puntpunt:    0x7fffc79a7428
**puntpunt:  0x7fffc79a7424
***puntpunt: 42
&puntpunt(hex):0x7fffc79a7430
```

I nomi delle matrici **non** allocate dinamicamente possono essere visti come puntatore a puntatore?

NO!

In generale, un array multidimensionale è visto come un puntatore a *elementi di tipo array*.

`int (*) [] ≠ int**`

i puntatori a elementi di tipo array non saranno argomento di questo corso.

```
In [30]: #include <stdio.h>
int main()
{
    int M[2][3] = {
        {3, 10, 42},
        {4, 11, 94}
    };
    printf("primo valore della matrice M vista come array: %d\n", M[0][0]);
    printf("primo valore della matrice M vista come puntatore: %d\n", (*M)[0]);
    printf("indirizzo usando M: %p; indirizzo usando &M[0]: %p\n", M, &M[0]);
    int** p_M = M;
    printf("primo valore della matrice M usando un puntatore a puntatore: %d\n", **p_M);
    return 0;
}
```

```

}

/tmp/tmp11spqkk8.c: In function 'main':
/tmp/tmp11spqkk8.c:11:17: warning: initialization of 'int **' from incompatible pointer type 'int (*)[3]' [-Wincompatible-pointer-types]
  11 |     int** p_M = M;
      |
primo valore della matrice M vista come array: 3
primo valore della matrice M vista come puntatore: 3
indirizzo usando M: 0x7ffc2f49bfc0; indirizzo usando &M[0]: 0x7ffc2f49bfc0
[C kernel] Executable exited with code -11

```

Puntatori di tipo void

Un puntatore di tipo `void` (ossia `void*`) può contenere l'indirizzo di un object di qualsiasi tipo

```
void* p;
int a = 3;
p = &a;
```

nessun problema in fase si compilazione. Ma...

```
In [31]: #include <stdio.h>

int main()
{
    void* p;
    int a = 3;
    p = &a;
    printf("%d", *p);

    return 0;
}
```

```

/tmp/tmpkrasiz3x.c: In function 'main':
/tmp/tmpkrasiz3x.c:8:15: warning: dereferencing 'void *' pointer
  8 |     printf("%d", *p);
      |     ^
/tmp/tmpkrasiz3x.c:8:15: error: invalid use of void expression
[C kernel] GCC exited with code 1, the executable will not be executed

```

La dereferenziazione di un puntatore `void` in generale non ha senso.

Il compilatore deve sapere *di che tipo* è la memoria a cui sta accedendo, così da poterla trattare nella maniera corretta (e.g. il valore da gestire da quanti byte è composto? devo considerarlo con segno o no? ecc.)

```
In [32]: #include <stdio.h>
// esempio di casting
int main()
{
    void* p;
    int a = 3;
    p = &a;
    printf("%d", *((int*)p));

    return 0;
}
```

3

```
In [33]: // Conversioni tra puntatori
#include <stdio.h>
int main()
{
    int x = 587;
    char* p;
    p = (char*) &x; // casting a tipo diverso
    printf("The (incorrect) value of x is: %d\n", *p);
    printf("size of int: %d, size of char: %d\n", sizeof(int), sizeof(char));
    printf("correct x value: (dec): %d (hex): %X\n", x, x);
    printf("incorrect x value: (dec): %d (hex): %X\n", *p, *p);
    return 0;
}
```

```
The (incorrect) value of x is: 75
size of int: 4, size of char: 1
correct x value: (dec): 587 (hex): 24B
incorrect x value: (dec): 75 (hex): 4B
```

Dato che `p` è dichiarato come puntatore a `char`, solo 1 byte del dato effettivo viene preso in considerazione invece dei 4 byte che compongono una variabile di tipo byte

Puntatori a strutture

```

#include <stdio.h>
#define MAX_PAZIENTI 3
struct Paziente
{
    float altezza;
    float peso;
    int eta;
};

int main()
{
    struct Paziente paz;
    struct Paziente* p_s;
    p_s = &paz;

    (*p_s).altezza = 1.70; //oppure p_s->altezza = 1.70;

}

```

Operatore `->`

data che l'operazione di accesso ai membri di una struttura puntata è più frequente di quanto si pensi, il C introduce l'operatore `->`.

Data una struct allocata con nome `S` ed avente un campo `B`, e dato un puntatore `A` contenente l'indirizzo di `S`, i.e., `A=&S`; allora:

$$A->B \equiv (*A).B$$

Array di puntatori a strutture

```

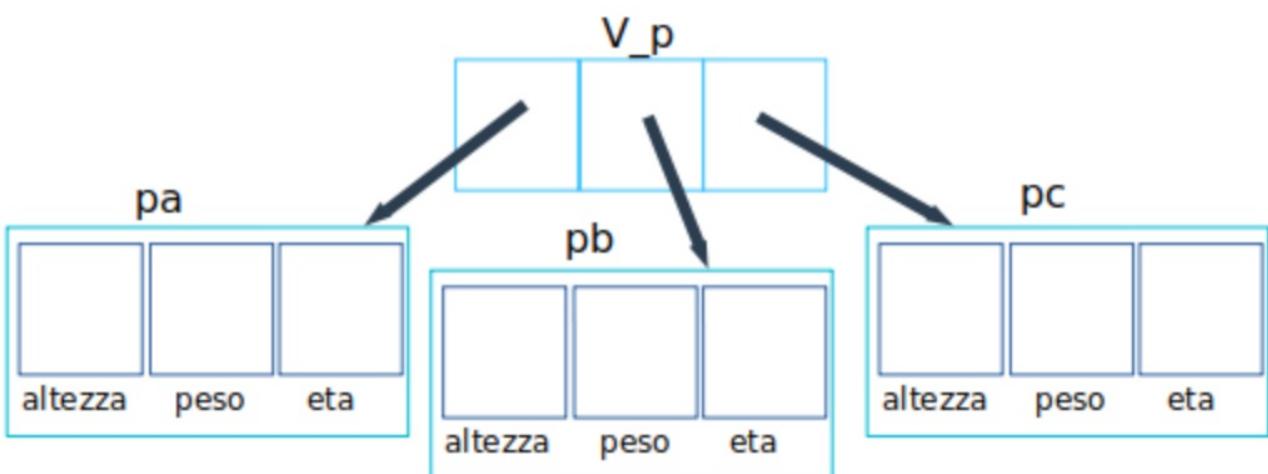
#include <stdio.h>
#define MAX_PAZIENTI 3
struct Paziente
{
    float altezza;
    float peso;
    int eta;
};

int main()
{

    struct Paziente* V_p[MAX_PAZIENTI];
    struct Paziente pa,pb,pc;
    V_p[0] = &pa;
    V_p[1] = &pb;
    V_p[2] = &pc;
    for(int i=0; i < MAX_PAZIENTI; i=i+1)
    {
        printf("inserisci l'altezza del %d paziente: ", i+1);
        scanf("%d", &(*V_p[i]).altezza);
        printf("inserisci il peso del %d paziente: ", i+1);
        scanf("%d", &V_p[i]->peso);
        printf("inserisci l'età del %d paziente: ", i+1);
        scanf("%d", &V_p[i]->eta)
    }

    return 0;
}

```



Ovviamente in questo caso specifico gli array di puntatori non sono né utili né convenienti...ma nel caso di allocazione dinamica delle

strutture sì (vedremo poi)

```
In [5]: #include <stdio.h>
struct Paziente
{
    float altezza;
    float peso;
    int eta;
};

int main()
{
    const int n = 3;
    struct Paziente* V_p[n];
    struct Paziente pa,pb,pc;
    V_p[0] = &pa;
    V_p[1] = &pb;
    V_p[2] = &pc;
    printf("dimensione del diario pazienti (solo vettore): %d Byte\n", sizeof(V_p));
    printf("dimensione di ogni puntatore: %d\n", sizeof(struct Paziente*));
    printf("dimensione di ogni struttura: %d\n", sizeof(struct Paziente));
    return 0;
}
```

dimensione del diario pazienti (solo vettore): 24

dimensione di ogni puntatore: 8

dimensione di ogni struttura: 12

Processing math: 100%

In []:

Il concetto di object

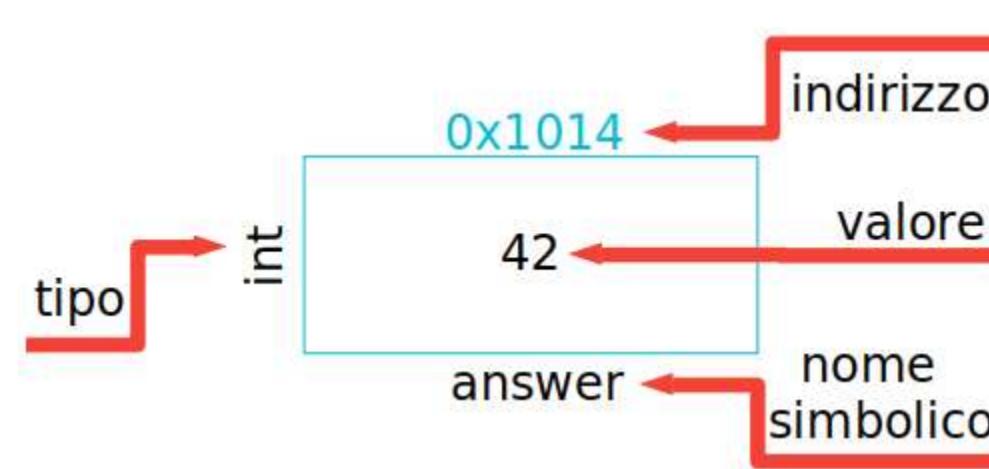
object: region of data storage in the execution environment, the contents of which can represent values

(From: C committee draft www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

NESSUNA relazione con la *Programmazione ad Oggetti* né con i file oggetto

Quindi in C, una regione di memoria contenente dati è chiamata genericamente **object** (variabili, array, etc.)

```
int answer = 42;
```



L'indirizzo corrisponde all'indirizzo di memoria del **primo** byte che componete l'object.

Dichiarazione Vs Definizione (object)

Dichiarazione: dichiara il nome ed il tipo di un object (in sostanza dice che l'object esiste da qualche parte)

Definizione: alloca lo spazio necessario per l'object (in sostanza viene materialmente creato in memoria)

In molti casi, le due fasi corrispondono. Esempio: `int a;` dichiara che esiste un object di nome `a` di tipo `int`, e contemporaneamente ne alloca lo spazio in memoria.

In generale, in uno stesso programma, un object può avere *più* dichiarazioni *ma una sola* definizione.

Esempio di un object *dichiarato* ma non *definito*:

```
extern int g;
```

extern dichiara l'esistenza dell'object `g`, definito (adesso o in futuro) da qualche altra parte. Il compilatore quindi, nel caso in cui sia richiesto l'utilizzo di `g` prima dell'effettiva *definizione*, non se ne preoccupa e continua il processo di compilazione (in quanto semplice traduzione).

Esempio:

```
In [11]: #include <stdio.h>
int main(void)
{
    extern int first, last; /* use global vars */
    printf("i valori di first e last sono: %d %d\n", first, last);
    return 0;
}
/* global definition of first and last */
int first = 10, last = 20;
i valori di first e last sono: 10 20
```

dato che si richiede l'utilizzo di `first`, `last` prima della loro effettiva *definizione*, vanno almeno *dichiarate* in precedenza.

Inizializzazione Vs Assegnazione

Assegnazione: dare valore ad un object in qualsiasi punto del programma

Inizializzazione: dare valore ad un object *in fase di definizione*

```
{
    int a; // dichiarazione & definizione
    a = 3; // assegnazione
}
{
    int a = 3; // dichiarazione & definizione & inizializzazione
}
```

In C, alcune cose lecite in fase di inizializzazione **non lo sono** in fase di assegnazione.

Ad ogni modo, un object è detto **inizializzato** se gli è stato dato un valore dal programma, indipendentemente dal modo in cui ciò sia avvenuto.

Esempio:

```
int V[] = {1,2,3}; // si può fare
int Q[]; // non si può fare
Q[] = {4,5,6}; // non si può fare
const int i = 5; // si può fare
const int i;
i = 5; // non si può fare
```

Layout della memoria di un programma C

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 `external`, `global` e `static`. Si divide a sua volta in:
 - **initialized data segment**: variabili `extern`, `global` e `static` i cui valori sono esplicitamente 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. **execution stack**, o *call stack*, o spesso chiamato comunemente *stack* (anche se fonte di ambiguità), contiene gli indirizzi di ritorno delle funzioni invocate, 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 comando linux `size` fornisce informazioni sull'occupazione di memoria di un programma:

esempio.c :

```
#include<stdio.h>

int main() {
    return 0;
}
gcc esempio.c -o esempio

size esempio
```

output:

```
text      data      bss      dec      hex   filename
1418      544        8     1970      7b2   esempio
```

La dimensione massima dell' *execution stack* può essere recuperata con il comando `ulimit -s` (l'output è espresso in KB).

ulimit -s

output:

```
8192
```

In questo caso, lo spazio a disposizione nell' *execution stack* è 8192 KB.

Tipi di allocazione in C

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 variabili 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.

Strutture dati native del C

Una struttura dati è un insieme di dati di tipi base aggregati assieme secondo uno schema.

Il C mette a disposizione in maniera nativa le seguenti strutture dati:

- **array**:
 - monodimensionali
 - multidimensionali
- **struct**:
 - semplici
 - ricorsive

Laboratorio di Programmazione

Corso di Laurea in Informatica

Gr. 3 (N-Z)

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Programma del corso (sezione I)

Dal sorgente al programma (parte 1)

- ricapitolazione dei processi di compilazione/linking
- compilazione/linking da linea di comando su sistemi linux
- struttura di un programma C in memoria
- il debugger GDB *

Ricapitolazione di alcuni fondamenti

- input/output di base
- Array monodimensionali
- Array multidimensionali
- struct *
- puntatori

Algoritmi notevoli su array (parte 1)

- alcuni noti algoritmi iterativi

Funzioni

- passaggio dei parametri di tipi base
- passaggio di parametri di struct *
- passaggio di parametri di array

Programma del corso (sezione II)

Puntatori e funzioni - argomenti avanzati

- aritmetica dei puntatori
- puntatori ad array multidimensionali
- le callbacks *
- argomenti da linea di comando *

I File

- File ASCII
- File binari *

Politiche di accesso (parte 1)

- code e pile
- implementazione con array *

La ricorsione *

- Algoritmi notevoli su array (parte 2) *
- alcuni noti algoritmi ricorsivi

Programma del corso (sezione III)

Allocazione dinamica in C

- allocazione dinamica di variabili
- allocazione dinamica di vettori
- allocazione dinamica di matrici
- allocazione dinamica di struct *

Liste concatenate *

- liste concatenate semplici
- liste concatenate doppie

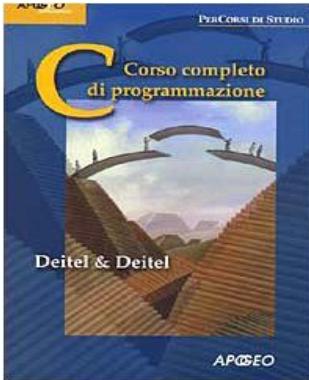
Politiche di accesso (parte 2) *

- pile e code con liste concatenate

Dal sorgente al programma (parte 2) *

- Programmazione multifile
- Header files vs C files vs Libraries
- Cyclic Include Problem

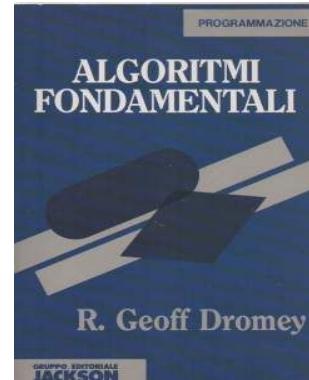
Libri di testo consigliati



Deitel & Deitel
C
**Corso completo di
programmazione**
(qualsiasi edizione)



B.W. Kernighan, D.M Ritchie
**Il linguaggio C. Principi di
programmazione e manuale
di riferimento**



G.R. Dromey
**Algoritmi
fondamentali**

Materiale didattico fornito dal docente in formato:

- pdf
- HTML (NB: aprire con un browser tipo Firefox o Chrome, NON usare il browser integrato di Teams in quanto potrebbe non visualizzare correttamente i file)

Lezioni

- Lunedì 11.00-13.00, aula T7
- Mercoledì 16.30-18.30, Laboratorio di Informatica (Dip. di Biologia)

Riferimenti del docente

Per qualsiasi comunicazione:
andrea.apicella@unina.it

Eventuali comunicazioni del docente saranno date sulla homepage:
<https://www.docenti.unina.it/andrea.apicella>

in caso di problemi sul sito, le comunicazioni verranno date sul canale Teams del corso.

In generale, riferirsi sempre alla homepage del docente

Modalità d'esame

- **Scritto + orale in caso di esito positivo dello scritto**
- La prova scritta verrà effettuata, a meno di variazioni, sui computer del laboratorio

Ambiente di sviluppo

- In laboratorio, potete usare l'ambiente di sviluppo che preferite tra quelli che trovate già installati sulle macchine
- Ciò non toglie che il corretto svolgimento dell'esame prevede la produzione di un programma eseguibile attraverso linea di comando con compilatore *gcc*
- In generale, se un programma viene generato in un dato ambiente, non è detto che venga generato allo stesso modo in altri ambienti (e.g. non standard ANSI)
- E' quindi consigliabile:
 - che gli esercizi siano svolti in ambiente Linux
 - che i sorgenti siano scritti da terminale attraverso editor di testo tipo *pico* o *nano*
 - che l'eseguibile venga prodotto attraverso linea di comando

Ambiente di sviluppo

- In laboratorio, potete usare l'ambiente di sviluppo che preferite tra quelli che trovate già installati sulle macchine
- Ciò non toglie che il corretto svolgimento dell'esame prevede la produzione di un programma eseguibile attraverso linea di comando con compilatore *gcc*
- In generale, se un programma viene generato in un dato ambiente, non è detto che venga generato allo stesso modo in altri ambienti (e.g. non standard ANSI)
- E' quindi consigliabile:
 - che gli esercizi siano svolti in ambiente Linux
 - che i sorgenti siano scritti da terminale attraverso editor di testo tipo *pico* o *nano*
 - che l'eseguibile venga prodotto attraverso linea di comando

An IDE, or Integrated Development Environment, will turn you stupid. They are the worst tools if you want to be a good programmer because they hide what's going on from you, and your job is to know what's going on. They are useful if you're trying to get something done and the platform is designed around a particular IDE, but for learning to code C (and many other languages) they are pointless. (Zed Shaw, "Learn C the hard way")

Funzione printf(...)

Definita come

`int printf(char *format, arg list ...)`**format** è una stringa costante che può contenere :

- whitespaces: spazi, tabulazioni, invii a capo (\n)
- caratteri di testo standard (qualsiasi sequenza di caratteri non bianchi che non inizi per %)
- specificatori di formato (che iniziano per %)

Ritorna il numero di caratteri stampati. Se qualcosa non va, restituisce un numero negativo.

Formato (%)	Tipo	Risultato
c	char	singolo carattere
i,d	int	numero decimale
o	int	numero ottale
x,X	int	numero esadecimale
u	int	intero senza segno
s	char *	stampa una stringa
f	double/float	formato -ddd.ddd...
e,E	double/float	formato scientifico (e.g. -1.34e003)
p	*	valore di un puntatore (ossia indirizzo contenuto)
%	-	stampa il carattere %

per maggiori informazioni

<https://cplusplus.com/reference/cstdio/printff>

In [13]:

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

int main()
{
    printf("Hello World");
    sleep(2);
    return 0;
}
```

Hello World

Domanda Perchè la stringa viene stampata dopo i 2 secondi e non prima?

In [9]:

// altro esempio

#include <stdio.h>

```
int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denominator:");
    scanf("%d", &d);
    printf("result:");
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

Hello World

numerator:

denominator:

[C kernel] Executable exited with code -8

Domanda

Perchè la stringa **result**: non viene stampata in caso di errore?

Risposta

I/O in C è *buffered*. Il dato viene conservato in un'area di memoria, e viene letto/stampato solo se si verificano determinate condizioni.

Questo per evitare un eccesso di chiamate di sistema, pesanti in termini computazionali.

In sostanza, invece di leggere/scrivere tante volte dati, si preferisce leggerli/scrivere tutti in una volta in un'unica chiamata interna.

la stringa **result**: rimane nel buffer in caso di errore.

Per l'output, C permette di utilizzare 3 tipi di strategia:

- *unbuffered*: l'output viene generato immediatamente, senza memorizzare i caratteri in nessun area temporanea
- *block buffered*: i caratteri sono conservati in un'area di memoria temporanea (buffer). Quando il buffer si riempie, il suo contenuto viene scritto in massa sul dispositivo di output (esempio, un file di testo). Tale buffer è svuotato anche in caso di una invocazione di `fflush(...)` o quando il dispositivo di output viene chiuso.
- *line buffered*: come block buffered, ma con in più lo svuotamento del buffer anche in caso di presenza del carattere newline \n.

Di default, in C l'output su file è block buffered, sullo standard output (i.e. lo schermo) è invece line buffered. Lo `stderr` è invece unbuffered.

Even though it is not mandatory in standards, functions which read from stdin can flush to the stdio file first.

In []:

#include <stdio.h>

```
int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denominator:");
    scanf("%d", &d);
    printf("result:\n"); // <-- il buffer viene svuotato quando trova un carattere \n
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

In C, è possibile "svuotare" (flush) qualsiasi stream di output. L'effetto è che qualsiasi dato contenuto viene effettivamente inserito nel relativo stream: la funzione di riferimento è `fflush(...)` ed è definita nell'header `stdio.h`.

From linux manual page:

Output streams that refer to terminal devices are always line buffered by default; pending output to such streams is written automatically whenever an input stream that refers to a terminal device is read.

Domanda: se la funzione `printf` è line buffered, perchè nel caso precedente le stringhe `numerator:` e `denominator:` vengono stampate subito nonostante la mancanza dello \n?

Risposta: anche se non obbligatorio negli standard, è prassi comune che funzioni che leggano dallo standard input "svuotino" in automatico lo stdio prima di essere eseguite.

In [10]:

#include <stdio.h>

```
int main()
{
    int n, d;
    printf("Hello World\n");
    printf("numerator:");
    scanf("%d", &n);
    printf("denominator:");
    scanf("%d", &d);
    printf("result:");
    fflush(stdout); // alternativa: fflush svuota i buffer di output
    int q = d/n;
    printf("%d\n", q);

    return 0;
}
```

Hello World

numerator:

denominator:

result:

[C kernel] Executable exited with code -8

Funzione scanf(...)

`int scanf (const char * format, ...);`da <https://cplusplus.com/reference/cstdio/scnaf/>

- in format, the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters).
- Non-whitespace character, except format specifier (%): Any character that is not either a whitespace character (blank, newline or tab) or part of a format specifier (which begin with a % character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of format. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- Format specifiers: A sequence formed by an initial percentage sign (%) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the stream and stored into the locations pointed by the additional arguments.

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the end-of-file.

In [6]:

#include<stdio.h>

int main(){

int a,b;

printf("inserisci interi:");

scanf("%d%d",&a,&b);

printf("%d %d", a, b);

return 0;

}

inserisci intero:1

2

inserisci intero:3

1 2

Nella `scanf(...)`, far seguire uno spazio ad uno specificatore di formato può creare problemi. Questo perché la `scanf(...)`, quando incontra uno spazio nella sua stringa di formato, non termina finché non viene inserito un carattere diverso da space o newline. E' necessario quindi inserire un ulteriore carattere (e.g., 2) per far terminare la `scanf(...)`. Il carattere 2 inserito per far terminare la `scanf("%d\n", &a)` non viene però "consumato", rimanendo nel buffer. La seconda chiamata `scanf("%d\n", &b)` "consumerà" il carattere 2, ponendolo nella variabile `b` (al posto del carattere 3, che rimane nel buffer, e viene utilizzato solo per far terminare la `scanf("%d\n", &b)`), in quanto anch'essa contiene uno spazio dopo lo specificatore.A whitespace (blank, newline, tab) character in a scanf format causes it to explicitly read and ignore as many whitespace characters as it can. So with `scanf("%d ", ...)`, after reading a number, it will continue to read characters, discarding all whitespace until it sees a non-whitespace character on the input. That non-whitespace character will be left as the next character to be read by an input function.

#include<stdio.h>

int main(){

int a;

char car;

printf("inserisci intero:");

scanf("%d", &a);

printf("inserisci carattere:");

scanf("%c", &car);

printf("hai inserito il carattere %c", car);

return 0;

}

Problema: la `scanf("%d", &a)` acquisisce e mette nella variabile `a` il valore inserito da tastiera.

La conferma del valore viene data con il tasto <Enter> che corrisponde ad un carattere di \n

Tale valore però non viene elaborato rimanendo nel buffer di tastiera (che ricorda essere un'area di memoria dedicata ai caratteri acquisiti da tastiera), aspettando la prossima acquisizione.

La successiva acquisizione è `scanf("%c", &car)`; ossia l'acquisizione di un carattere (%c). Tale acquisizione troverà il precedente carattere \n, considerandolo come carattere da mettere nella variabile `car`, e quindi impedendo all'utente di fare l'acquisizione effettiva.

Possibile soluzione:

- fare un'acquisizione a vuoto per consumare il carattere rimanente
 - `getchar()`; oppure `getch(stdin)`; nel caso di un solo carattere nel buffer

più in generale:

#include <stdio.h>

void clean_stdin(void)

{

int c;

do

{

c = getchar();

}

while (c != '\n' && c != EOF);

}

Esempio di NON soluzione

- In alcuni casi, viene riportata come possibile soluzione l'utilizzo della funzione `fflush(stdin)`. Tuttavia, tale soluzione potrebbe avere comportamento indefinito in alcune implementazioni/versioni delle librerie che la contengono.
- Questo perchè la funzione `fflush(...)` non nasce per lavorare su buffer di input (come la tastiera), ma su buffer di output -> soluzione non valida in quanto non standard.

#include<stdio.h>

int main(){

int a;

int b;

printf("inserisci a e b:");

scanf("%d%d", &a, &b);

return 0;

}

inserisci a e b:1

2

inserisci a e b:3

1 2

Nella `scanf(...)`, far seguire uno spazio ad uno specificatore di formato può creare problemi. Questo perché la `scanf(...)`, quando incontra uno spazio nella sua stringa di formato, non termina finché non viene inserito un carattere diverso da space o newline. E' necessario quindi inserire un ulteriore carattere (e.g., 2) per far terminare la `scanf(...)`. Il carattere 2 inserito per far terminare la `scanf("%d\n", &a)` non viene però "consumato", rimanendo nel buffer. La seconda chiamata `scanf("%d\n", &b)` "consumerà" il carattere 2, ponendolo nella variabile `b` (al posto del carattere 3, che rimane nel buffer, e viene utilizzato solo per far terminare la `scanf("%d\n", &b)`), in quanto anch'essa contiene uno spazio dopo lo specificatore.A whitespace (blank, newline, tab) character in a scanf format causes it to explicitly read and ignore as many whitespace characters as it can. So with `scanf("%d ", ...)`, after reading a number, it will continue to read characters, discarding all whitespace until it sees a non-whitespace character on the input. That non-whitespace character will be left as the next character to be read by an input function.

#include<stdio.h>

int main(){

int a;

char car;

printf("inserisci intero:");

scanf("%d", &a);

printf("inserisci carattere:");

scanf("%c", &car);

printf("hai inserito il carattere %c", car);

return 0;

}

Problema: la `scanf("%d", &a)` acquisisce e mette nella variabile `a` il valore inserito da tastiera.

La conferma del valore viene data con il tasto <Enter> che corrisponde ad un carattere di \n

Tale valore però non viene elaborato rimanendo nel buffer di tastiera (che ricorda essere un'area di memoria dedicata ai caratteri acquisiti da tastiera), aspettando la prossima acquisizione.

La successiva acquisizione è `scanf("%c", &car)`; ossia l'acquisizione di un carattere (%c). Tale acquisizione troverà il precedente carattere \n, considerandolo come carattere da mettere nella variabile `car`, e quindi impedendo all'utente di fare l'acquisizione effettiva.

Possibile soluzione:

- fare un'acquisizione a vuoto per consumare il carattere rimanente
 - `getchar()`; oppure `getch(stdin)`; nel caso di un solo carattere nel buffer

più in generale:

#include <stdio.h>

void clean_stdin(void)

{

int c;

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Le Funzioni

Si desidera implementare in C un programma *CalcolatriceCarina* che, dati gli operandi, visualizzi il risultato delle 4 operazioni base (+,-,/,x) circondato da una cornice di asterischi.

```
dammi in primo operando
4
dammi il secondo operando
2
la somma di 4.00 e 2.00 e'
*****
**       6.00   **
*****
la differenza tra 4.00 e 2.00 e'
*****
**       2.00   **
*****
il prodotto tra 4.00 e 2.00 e'
*****
**       8.00   **
*****
il quoziente tra 4.00 e 2.00 e'
*****
**       2.00   **
*****
```

```
In [44]: #include <stdio.h>
int main()
{
    float op1;
    float op2;
    float ris;
    printf("dammi in primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;
    printf("*****\n");
    printf("%s%.2f%s\n", "****", 10, ris , 10,"****");
    printf("*****\n");
    printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;
    printf("*****\n");
    printf("%s%.2f%s\n", "****", 10,ris , 10,"****");
    printf("*****\n");
    printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;
    printf("*****\n");
    printf("%s%.2f%s\n", "****", 10,ris , 10,"****");
    printf("*****\n");
    printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;
    printf("*****\n");
    printf("%s%.2f%s\n", "****", 10,ris , 10,"****");
    printf("*****\n");
    return 0;
}
```

dammi in primo operando
dammi il secondo operando

```

la somma di 2.00 e 2.00 e'
*****
**      4.00      **
*****
la differenza tra 2.00 e 2.00 e'
*****
**      0.00      **
*****
il prodotto tra 2.00 e 2.00 e'
*****
**      4.00      **
*****
il quoziente tra 2.00 e 2.00 e'
*****
**      1.00      **
*****

```

Lo stesso codice ripetuto più volte!

```

#include <stdio.h>
int main()
{
    float op1;
    float op2;
    float ris;
    printf("dammi in primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;

    printf("*****\n");
    printf("%s%.2f%s\n", "**", 10, ris , 10, "**");
    printf("*****\n");

    printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;

    printf("*****\n");
    printf("%s%.2f%s\n", "**", 10, ris , 10, "**");
    printf("*****\n");

    printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;

    printf("*****\n");
    printf("%s%.2f%s\n", "**", 10, ris , 10, "**");
    printf("*****\n");

    printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;

    printf("*****\n");
    printf("%s%.2f%s\n", "**", 10, ris , 10, "**");
    printf("*****\n");

    return 0;
}

```

- Definisco una funzione col codice ripetuto

```

#include <stdio.h>

void incornicia(float numero)
{
    printf("*****\n");
    printf("%s%.2f%s\n", "**", 10, numero , 10, "**");
    printf("*****\n");
}

```

- Invoco la funzione

```
int main()
```

```

{
    float op1;
    float op2;
    float ris;
    printf("dammi il primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;
    incornicia(ris);
    printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;
    incornicia(ris);
    printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;
    incornicia(ris);
    printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;
    incornicia(ris);
}

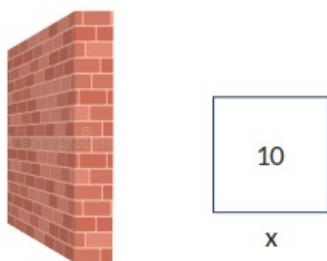
```

L'output effettivo è identico al precedente.

I parametri: perchè?

Le variabili dichiarate all'interno di un blocco **non sono visibili** (ossia non sono accessibili) all'esterno del blocco in cui sono dichiarate

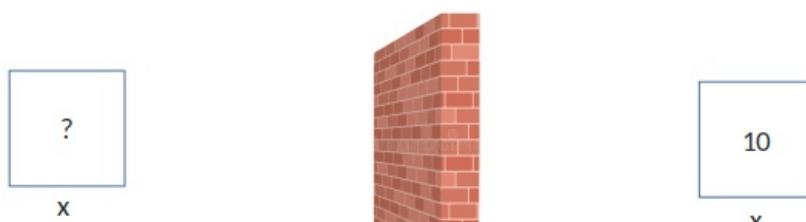
Definizione	Invocazione
<code>```c void incornicia() { printf("*****\n"); printf("%s%.2f%s\n", "", 10, x, 10,"**"); printf("*****\n"); }```</code>	<code>```c int main() { float x = 10; Incornicia(); }```</code>



Errore di compilazione! La variabile **x** non è definita nello scope della funzione *incornicia*

Potrei pensare di dichiarare una nuova variabile in *incornicia*, MA...

Definizione	Invocazione
<code>```c void incornicia() { float x; printf("*****\n"); printf("%s%.2f%s\n", "", 10,x, 10,"**"); printf("*****\n"); }```</code>	<code>```c int main() { float x = 10; Incornicia(); }```</code>



Viene creata una **nuova** variabile, con lo stesso nome ma valore diverso

Il programma verrebbe eseguito, ma l'output non sarebbe quello desiderato

NECESSARIO UN SISTEMA DI COMUNICAZIONE TRA BLOCCHI

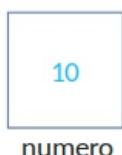


utilizzo dei parametri!

I parametri: come?

Definizione	Invocazione
-------------	-------------

```
``c void incornicia(float numero) { printf("*****\n"); printf("%s%.*2f%s\n", "", 10,numero, 10,"**");
printf("*****\n"); } ``c int main() { float x = 10;
Incornicia( x ); } ``
```



Viene creata una **nuova** variabile di nome *numero*, **copia** della variabile x

parametri formali: parametri dichiarati come parte *dichiarazione/definizione* della funzione

parametri attuali: parametri passati durante l'*invocazione* della funzione

Nello standard C è più corretto parlare di *parametri* e *argomenti*, rispettivamente al posto di *parametri formali* e *parametri attuali*.

Ritorno al chiamante

Il meccanismo del passaggio di parametri permette di copiare valori dall'*invocante* all'*invocato*, ma non viceversa.

Problema: implementare un programma che, data base ed altezza, stampi a video l'area ed il perimetro di un rettangolo

```
#include <stdio.h>
int main()
{
    float b, h;
    float area, perimetro;
    printf("inserire il valore della base\n");
    scanf("%f", &b);
    printf("inserire il valore dell'altezza\n");
    scanf("%f", &h);
    area = b * h;
    perimetro = 2*b + 2*h;
    printf("l'area è %.2f \n", area);
    printf("il perimetro è %.2f \n", perimetro);
}
```

Una prima soluzione **sbagliata**

Definizione

```
``c void calcola_area(float b, float h, float area) {
    area = b * h; } void calcola_perimetro(float b,
    float h, float perimetro) { perimetro = 2*b + 2*h;
} ``c int main() { float b, h; float area, perimetro; printf("inserire il valore della base\n");
scanf("%f", &b); printf("inserire il valore dell'altezza\n"); scanf("%f", &h); calcola_area(b, h, area);
calcola_perimetro(b, h, perimetro); printf("l'area è %.2f \n", area); printf("il perimetro è %.2f \n",
perimetro); } ``
```

Invocazione

La tecnica del passaggio di parametri, in C, è definita **solo per copia**. Tale copia viene eseguita soltanto all'atto dell'invocazione della funzione dal chiamante al chiamato, e non alla fine dal chiamato al chiamante!

Definizione

```
``c float calcola_area(float b, float h) { float a;
a = b * h; return a; } float calcola_perimetro(float b,
float h) { float p; p = 2*b + 2*h; return p; ``c int main() { float b, h; float area, perimetro; printf("inserire il valore della base\n");
scanf("%f", &b); printf("inserire il valore dell'altezza\n"); scanf("%f", &h); area = calcola_area(b, h);
perimetro = calcola_perimetro(b, h); printf("l'area è %.2f \n", area); printf("il perimetro è %.2f \n",
perimetro); } ``
```

Invocazione

Il valore che segue la parola chiave `return` viene copiato all'interno della variabile a sinistra dell'assegnazione nell'invocante (o in un'eventuale espressione). Se una funzione restituisce un valore attraverso la parola chiave `return`, il tipo di ritorno deve essere dello stesso tipo del valore restituito (e non più `void`)

In generale:

Una funzione in C viene **dichiarata/definita** nel seguente modo:

```
tipo Ritorno nome_fun(tipo_param1 nome_param1,
                        tipo_param2 nome_param2, ...)
{
    ...corpo...
    return valore_ritorno;
}
```

- tutto ciò che viene prima della `{` è detto **header**
- la lista dei parametri (formali) è composta da **dichiarazioni di variabili** (quindi compresa di tipo) separati da `,`
- il tipo di ritorno deve corrispondere al tipo del valore di ritorno. In caso contrario, il valore di ritorno sarà convertito nel tipo di ritorno.
- nel caso in cui non sia necessario che la funzione restituisca un valore, `tipo Ritorno` può essere `void` e la parola chiave `return` può essere omessa (o lasciata senza valore, ossia `return;`)
- I valori che assumono i parametri in fase di invocazione sono detti **parametri attuali**
- una funzione può anche solo essere **dichiarata** senza essere (ancora) definita esplicitando l'header seguito da un `;`. Esempio:

```
tipo Ritorno nome_fun(tipo_param1 nome_param1,
                      tipo_param2 nome_param2, ...);
```

La **dichiarazione** senza definizione di una funzione serve a segnalare che esiste(rà) una funzione di nome `nome_fun` avente parametri di tipo `tipo_param1, tipo_param2, ...`, la cui **definizione** è specificata altrove.

In [4]:

```
#include <stdio.h>
int f()
{
    double a = 3.5;
    return a;
}

int main()
{
    double r = f(); // invocazione
    printf("%f\n", r);

    return 0;
}
```

3.000000

In [7]:

```
#include <stdio.h>
void stampa_doppio(int n)
{
    printf("%d", n*2);
}

int main()
{
    stampa_doppio(4); //invocazione

    return 0;
}
```

8

L'invocazione di una funzione che restituisce un valore presuppone che tale invocazione sia fatta all'interno di una espressione o un'assegnazione.

Esempio:

```
double quadrato(double l)
{
    return l*l;
}

int main()
{
    int q = quadrato(10);
    printf("il quadrato di 10 è %d\n", q);

    printf("il quadrato di 3 è %d\n", quadrato(3));

    int l = q + quadrato(3);

    printf("la loro somma è %d\n", l);

    return 0;
}
```

da notare:

- non salvare `quadrato(3)` in una variabile non è convenuto, in quanto l'ho dovuto ricalcolare con una seconda invocazione
- l'object `int l` dichiarato nel corpo del `main()` non ha nulla a che vedere con il `double l` dichiarato come parametro della funzione `quadrato(double l)`. Gli object dichiarati nel corpo di una funzione sono visibili solo all'interno di quest'ultima
→

Gli **scope** sono separati (anche per questo sono necessari i parametri)

- `return` esce dalla funzione in esecuzione (anche se quest'ultima non è terminata, quindi attenzione a dove viene messa!)

In [10]:

```
# include <stdio.h>
//Esempio:
double calcola_media(double a, double b)
{
    double media = a + b;
    return media;
    media = media / 2.0;
}

int main()
{
    printf("la media tra 2 e 3 è %f\n", calcola_media(2,3));
    return 0;
}
```

la media tra 2 e 3 è 5.000000

In [11]:

```
# include <stdio.h>
double calcola_valore_assoluto(double n)
{
    if( n >= 0 )
    {
        return n;
    }
    else
    {
        return -n;
    }
}

int main()
{
    printf("il valore assoluto di -4 è %f\n",
           calcola_valore_assoluto(-4));
    return 0;
}
```

il valore assoluto di -4 è 4.000000

La funzione main()

il cosiddetto `main()` è una funzione a tutti gli effetti.

`main()` è la funzione che viene invocata all'atto dell'esecuzione del programma (*entry point*). Sarà quindi sua cura invocare al suo interno le funzioni che devono essere a loro volta utilizzate nel programma (se necessario). Niente vieta di dichiarare/definire funzioni senza mai invocarle.

In quanto funzione:

- viene dichiarata con tipo di ritorno (`int`, in versioni di C precedenti era accettato anche `void`)
- restituisce un valore (per questo `return 0;`) al termine della sua esecuzione, che corrisponde al termine del programma.
- può anche avere parametri, ma lo vedremo più in là

Perchè proprio

0

?

Exit code

Soltanamente il valore di ritorno della funzione `main()` (o anche *exit code*) è un intero tra

0

e

255

che può servire al sistema operativo ad innescare determinate operazioni una volta terminato il programma. Se il programma termina correttamente ci si aspetta che tale valore sia

0

, se invece ci sono stati errori il valore di ritorno potrebbe essere

$\neq 0$

.

Se si desidera visualizzare l'exit code dell'ultimo programma mandato in esecuzione, dare il comando linux `echo $?`

In [13]:

```
# include <stdio.h>
int main()
{
    return 34;
}
```

Il call stack

```
In [4]: # include <stdio.h>

double get_somma(double a, double b)
{
    double ret = a + b;
    return ret;
}

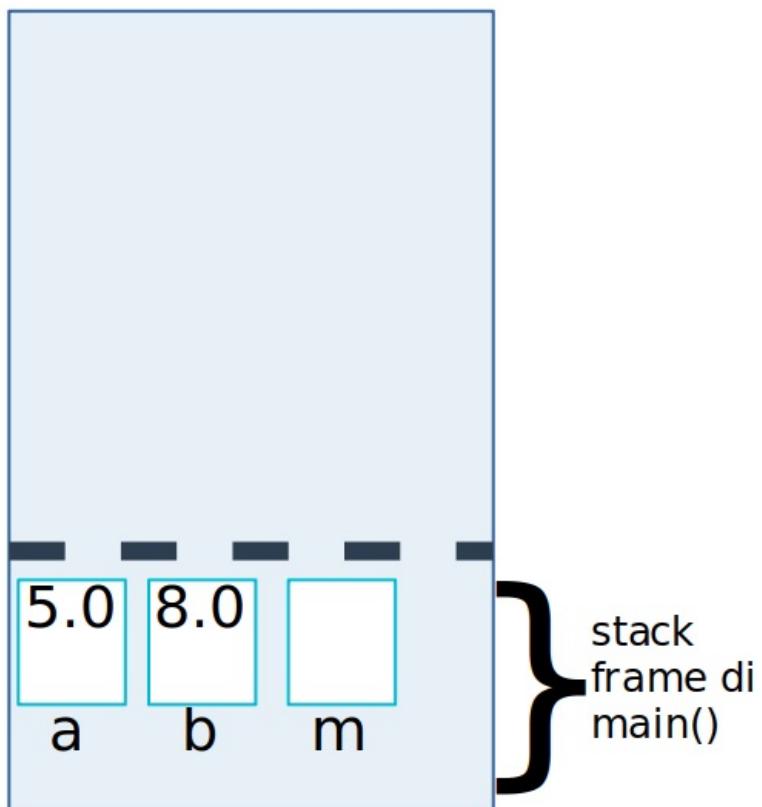
double get_media(double a, double b)
{
    double s;
    s      = get_somma(a, b);
    double ret;
    ret   = s/2.0;
    return ret;
}

int main()
{
    double a = 5;
    double b = 8;
    double m;
    m      = get_media(a,b);
    printf("la media tra %.2lf e %.2lf è %.2lf\n",a,b,m);

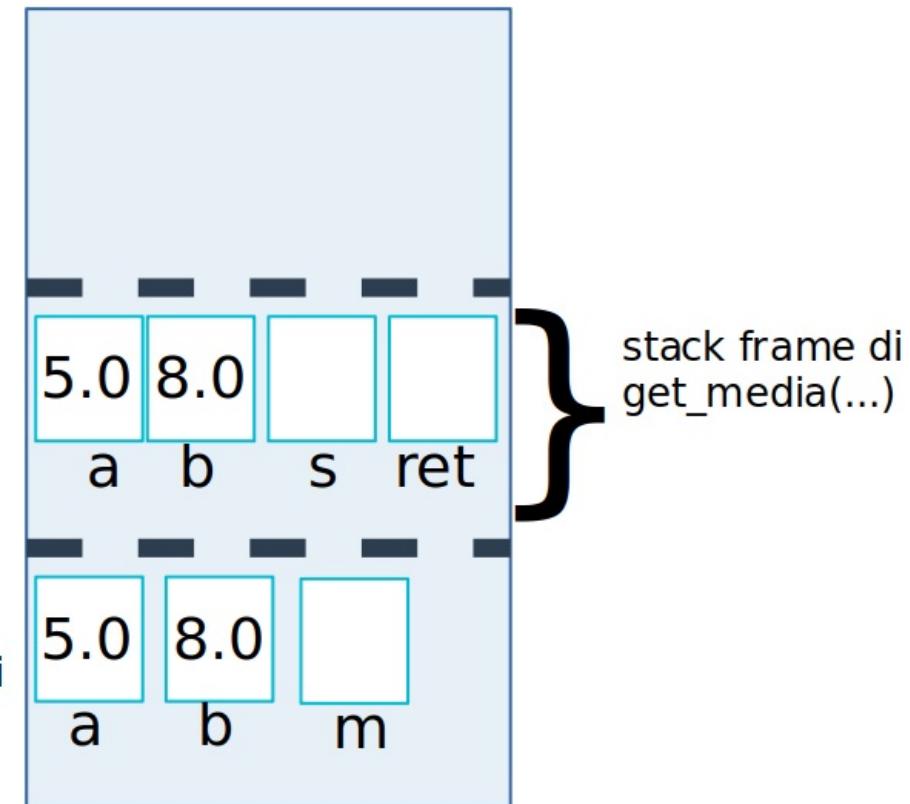
    return 0;
}
```

la media tra 5.00 e 8.00 è 6.50

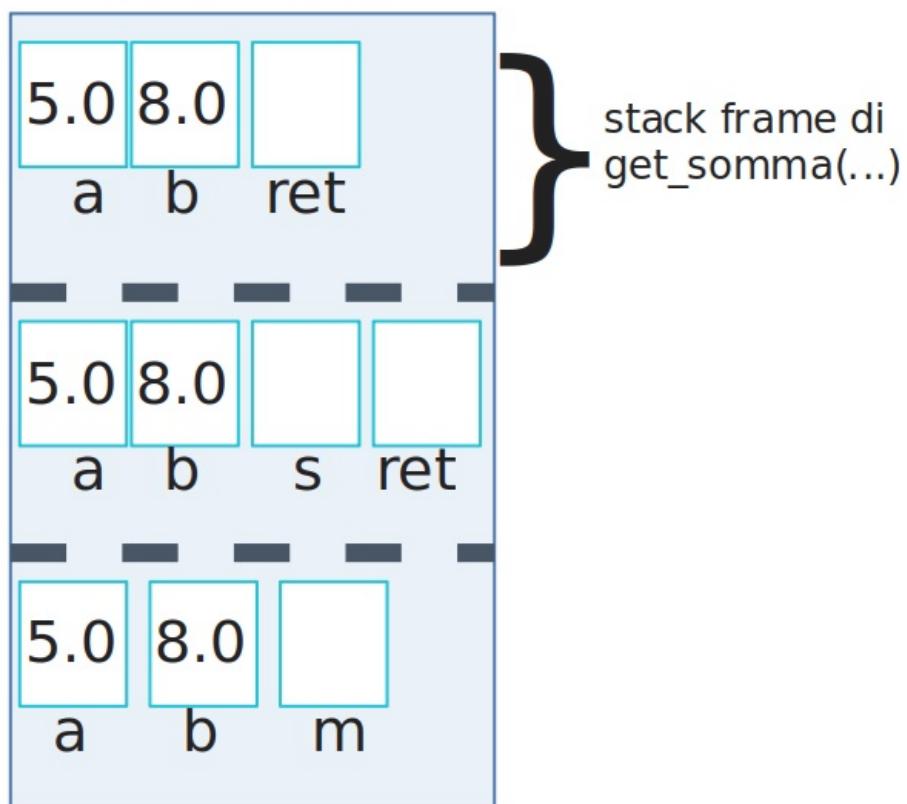
Call Stack



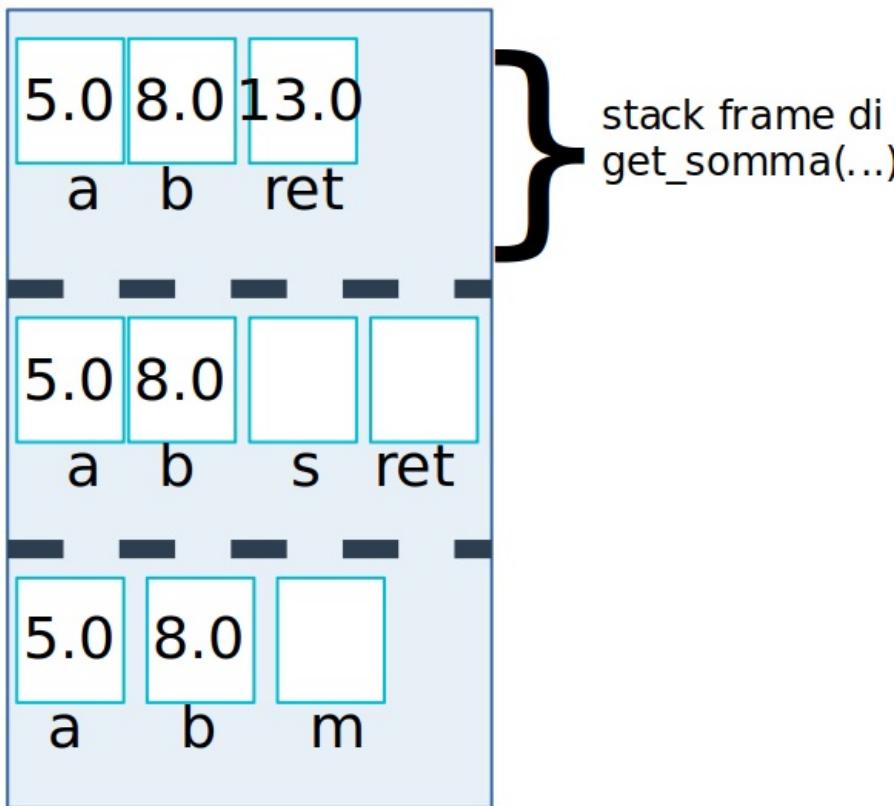
Call Stack



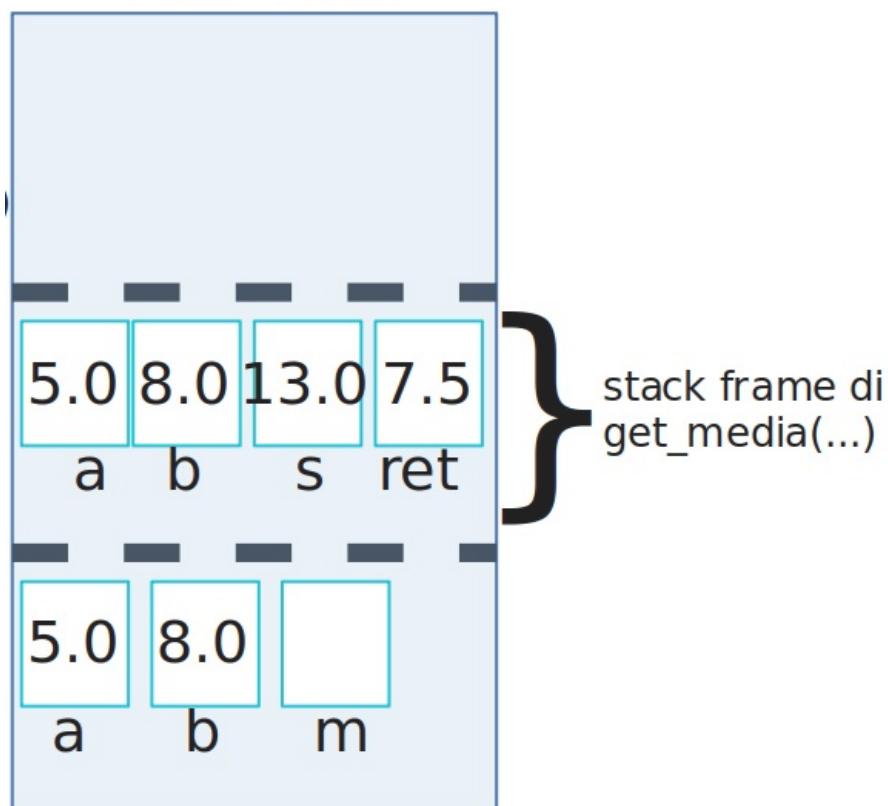
Call Stack



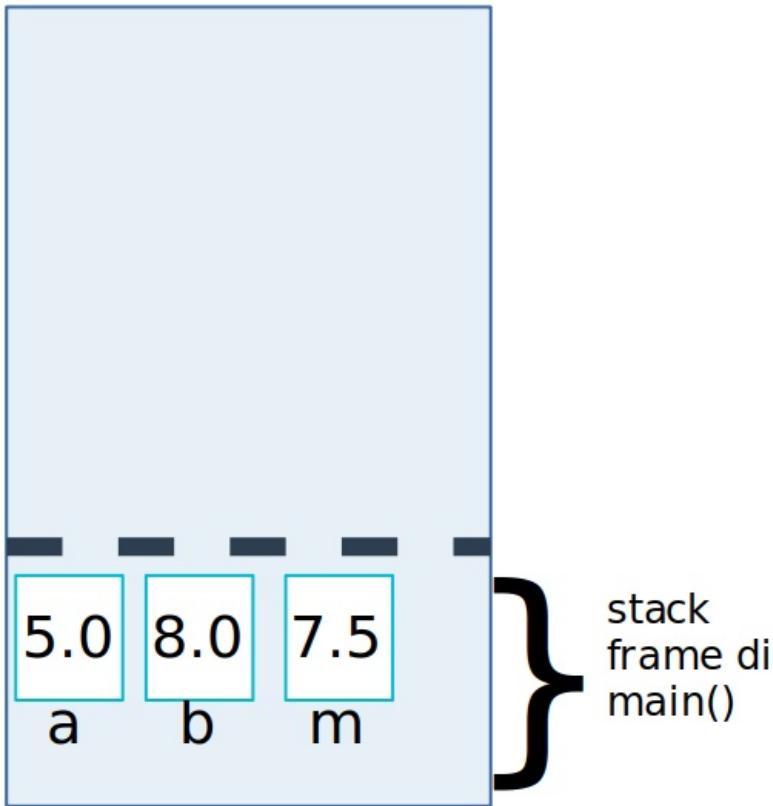
Call Stack



Call Stack



Call Stack



Ogni volta che viene invocata una funzione, viene impegnato un frammento (Stack Frame) del *call stack*. Tale frammento conterrà:

- le variabili locali della funzione
- eventuali parametri attuali
- dati (e.g., l'indirizzo di ritorno) per ripristinare lo stato al programma al termine della funzione Ogni stack frame verrà eliminato dal call stack al termine della relativa funzione.

Parametri di uscita: come?

Il problema dello scambio

Definire una funzione che, date due variabili, ne scambi il contenuto. Stampare quindi, nel chiamante, le variabili scambiate

Versione senza funzioni:

```
#include <stdio.h>
int main()
{
    int a;
    int b;
    int t; // necessaria per lo scambio
    printf("dammi il valore della variabile a\n");
    scanf("%d", &a);
    printf("dammi il valore della variabile b\n");
    scanf("%d", &b);
    t = a;
    a = b;
    b = t;
    printf("dopo lo scambio a vale %d e b vale %d\n", a, b);
}
```

Una soluzione sbagliata

Definizione

```
```c void scambia(int a, int b) { int a; int b; int t; // necessaria per lo scambio t = a; a = b; b = t; }```
```c int main() { int a; int b; printf("dammi il valore della variabile a\n"); scanf("%d", &a); printf("dammi il valore della variabile b\n"); scanf("%d", &b); scambia(a,b); printf("dopo lo scambio a vale %d e b vale %d\n", a, b); }```

```

Invocazione

Il programma, compilato ed eseguito, **non** scambierà il contenuto delle variabili.

Definizione

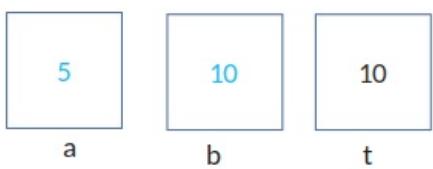
Invocazione

```

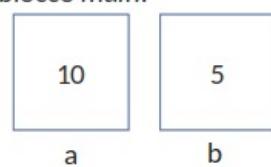
```c void scambia(int a, int b) { int t;           ``c int main() { int a; int b; printf("dammi il valore della variabile a\n"); scanf("%d", &a); printf("dammi il valore della variabile b\n"); scanf("%d", &b); scambia(a,b); printf("dopo lo scambio a vale %d e b vale %d\n", a, b); }
// necessaria per lo scambio t = a; a = b; b = t; ...
```

```

vengono scambiate le copie nel blocco di *scambia*....



...ma non le variabili nel blocco *main*!



Le variabili `a, b, t` sono variabili *locali* del metodo `scambia(. . .)`, mentre le variabili `a, b` sono variabili *locali* del metodo `main()`

Definizione

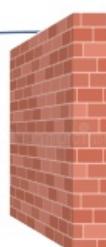
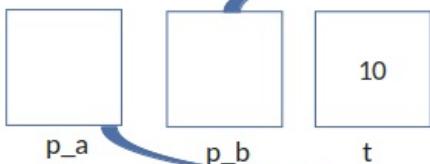
```

```c void scambia(int* p_a, int* p_b){ int t; //      ``c #include int main() { int a; int b; printf("dammi il valore della variabile a\n"); scanf("%d", &a);
necessaria per lo scambio t = *p_a; *p_a = printf("dammi il valore della variabile b\n"); scanf("%d", &b); scambia(&a, &b); printf("dopo lo
*p_b; *p_b = t; } ``c
```

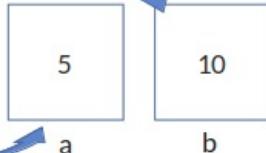
```

Invocazione

vengono scambiate i valori referenziati dai puntatori



...modificando quindi le variabili giuste



Domanda: potevo risolvere lo stesso problema utilizzando la parola chiave `return` ?

Risposta: **no**, in quanto con la parola chiave `return`, in C, posso restituire **al più un valore**. In alternativa, avrei potuto restituire il valore di una variabile con `return` (es. `return a;`), e modificare l'altra (es. `b`) con un puntatore

Domanda: quindi posso utilizzare la tecnica del passaggio di parametri per puntatori e del `return` nella stessa funzione?

Risposta: **sì**, sono due cose totalmente diverse, quindi possono essere utilizzate tranquillamente assieme.

Passaggio di array allocati tramite operatore `[]` ad una funzione

Il passaggio di parametri di array è da considerarsi sempre *per riferimento*

```

void f(int V[10], int n)
{
    for(int i=0; i < n; i++)
    {
        scanf( "%d", &V[i]);
    }
}

int main()
{
    int n;
    int vett[10];
    printf("quanti elementi vuoi inserire?");
    scanf( "%d", &n);

    f(vett,n);

    printf("i valori inseriti sono:\n");
    for(int i=0; i < n; i++)
    {
        printf( "%d ", vett[i]);
    }
    printf( "\n");
    return 0;
}

```

Questo perchè il nome simbolico del vettore `vett` si riferisce sempre all'*indirizzo* del vettore, quindi "passare" un vettore come

parametro significa, di fatto, passare il suo indirizzo.

Analogamente, la dichiarazione del parametro `int V[10]` è simile alla dichiarazione di un puntatore. Volendo, la stessa funzione poteva essere definita come:

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

La dimensione, nel caso di array monodimensionali come parametro, non è necessaria. Dato che, all'atto pratico, ciò che viene passato è l'indirizzo ad un `int` (nello specifico, l'indirizzo del primo elemento dell'array), tale funzione poteva anche essere scritta come:

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

Discorso simile, ma non identico, con array multidimensionali:

```
void f(int M[10][3], int n_r, int n_c)
{
    for(int i=0; i < n_r; i++)
    {
        for(int j=0; j < n_c; j++)
        {
            scanf("%d", &M[i][j]);
        }
    }
}

int main()
{
    int n_r, n_c;
    int mat[10][3];
    printf("quante righe vuoi inserire?");
    scanf("%d", &n_r);
    printf("quante colonne vuoi inserire?");
    scanf("%d", &n_c);
    f(vett,n);

    printf("i valori inseriti sono:\n");
    for(int i=0; i < n_r; i++)
    {
        for(int j=0; j < n_c; j++)
            printf("%d ", mat[i][j]);
    }
    printf("\n");
    return 0;
}
```

MA... in questo caso la dichiarazione del parametro `int M[10][3]` come `int M[][]` non va bene!

Perchè?

semplifichiamo il problema analizzando la seguente funzione (sbagliata):

```
void f(int M[][])
{
    M[1][2] = 42; // dove finisce la riga 0
                  // e dove inizia la colonna 1??
}
```

ricordiamo che una matrice in memoria è rappresentata in maniera sequenziale in modalità row-major (in C). Senza conoscere il numero di colonne, come può la funzione determinare dove finisce la riga con indice `0` ed inizia la riga con indice `1`? Fornire alla funzione almeno il numero di colonne è indispensabile. Quindi una versione corretta della funzione è la seguente:

```
void f(int M[][3])
{
    M[1][2] = 42;
}
```

Questa funzione è quindi valida per qualsiasi matrice avente 3 colonne, indipendentemente dal numero di righe che tale matrice abbia.

Come specificato quando si è parlato di puntatori, però, i nomi delle matrici (o più in generale gli array multidimensionali) allocate tramite

l'operatore `[]` **non** possono essere visti come *pointers to pointers* (i.e. `**`), quindi una dichiarazione di parametro *formale* fatta come nella seguente funzione:

```
void f(int** M)
{
    ...
}
```

non sarebbe compatibile con un parametro *attuale* di tipo matrice allocata tramite operatore `[]`.

Passare un array ad una funzione per copia

Se proprio indispensabile, c'è un trick per passare un vettore ad una funzione per copia. La tecnica consiste nell'incapsularlo in una struttura:

```
struct Involutro
{
    int vett[10];
}

void f(struct Involutro inv, int n)
{
    for(int i=0; i < n; i++)
    {
        scanf("%d", &(inv.vett[i]));
    }
}

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

    f(t,n);

    printf("i valori inseriti sono:\n");
    for(int i=0; i < n; i++)
    {
        printf("%d ", t.vett[i]);
    }
    printf("\n");
    return 0;
}
```

Una struttura, in C, di default viene passata per copia, e quindi con essa viene **copiato** tutto il suo contenuto, compresi eventuali array definiti attraverso `[]`.

Questa funzione, quindi, farà ciò che ci aspettiamo che faccia??

Ovviamente la risposta è **no**, in quanto la struct `t` viene passata *per copia*.

In []:

In []:

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

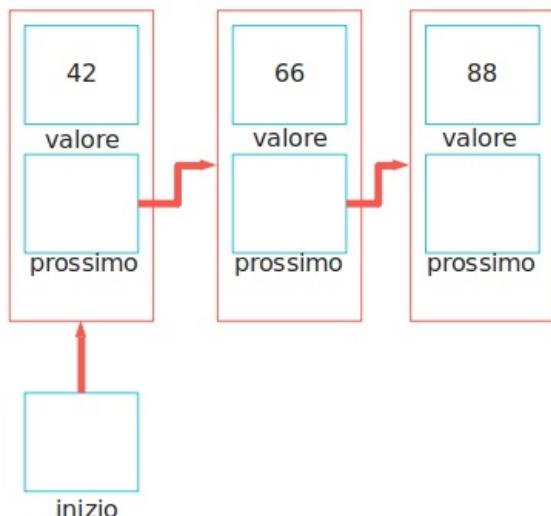
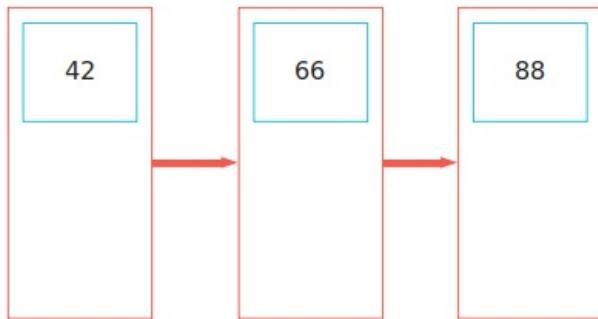
Liste concatenate

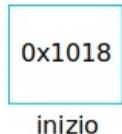
Una lista concatenata è una struttura dati dinamica in cui gli elementi possono essere inseriti e/o eliminati in qualsiasi punto. In questo modo ogni elemento può essere allocato se e solo quando sarà necessario.

In base a come sono collegati i dati tra loro, possono essere creati diversi tipi di lista concatenata. Ad esempio:

1. a collegamento singolo: ogni elemento è collegato solo all'elemento successivo
2. a collegamento doppio: ogni elemento è collegato sia con l'elemento successivo che con l'elemento precedente

Liste a collegamento singolo





Una lista concatenata quindi può essere vista come una sequenza di elementi (chiamati **nodi**) in cui ogni elemento punta all'elemento successivo.

Ovviamente, è necessario un puntatore per accedere al primo elemento della lista. Tale primo elemento è generalmente chiamato **testa**.

Strutture ricorsive

Una struttura è detta *ricorsiva* se contiene al suo interno un membro di tipo puntatore dello stesso tipo della struttura.

E.g.

```

struct elemento {
    int valore;
    struct elemento* prossimo;

};
    
```

In generale, una struttura non può contenere un'*istanza* del suo stesso tipo, ma nulla vieta che contenga un *puntatore* ad un'*istanza* del suo stesso tipo

E.g.

```

struct Paziente
{
    float altezza;
    float peso;
    int eta;
    struct Paziente* prossimo;
};
    
```

Utili nella progettazione di *liste concatenate* o, ad esempio, di strutture gerarchiche.

```

In [ ]: ````c
#include <stdio.h>
struct Persona
{
    char nome[30];
    char cognome[30];
    float altezza;
    float peso;
    int eta;
    struct Persona* padre;
    struct Persona* madre;
};
    
```

```

In [6]: // Esempio di (brutta) implementazione di una lista concatenata
#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

int main()
{
    struct elemento* inizio = malloc(sizeof(struct elemento));
    
```

```

inizio->valore = 42;
inizio->prossimo = malloc(sizeof(struct elemento));
inizio->prossimo-> valore = 66;
inizio->prossimo-> prossimo = malloc(sizeof(struct elemento));
inizio->prossimo-> prossimo-> valore = 88;
inizio->prossimo-> prossimo-> prossimo = NULL;

while(inizio != NULL)
{
    printf("%d\n", inizio->valore);
    inizio = inizio->prossimo; //perdo l'inizio
}

return 0;
}

```

42
66
88

Nell'implementazione precedente, ogni elemento viene inserito in maniera hard-coded.

Possiamo migliorare la procedura?

Problema: copiare il contenuto di un array in una lista concatenata

```
In [15]: /* Problema: copiare il contenuto di un vettore in una lista concatenata */
// Strategia 0 (errata)
#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

int main()
{
    int valori[] = {42,66,88};
    const int n = 3;

    struct elemento* inizio = malloc(sizeof(struct elemento));

    struct elemento* in_esame = inizio;
    for(int i = 0; i < n; i=i+1)
    {
        in_esame->valore = valori[i];
        in_esame->prossimo = malloc(sizeof(struct elemento));
        in_esame = in_esame->prossimo;
    }

    in_esame = inizio;
    while(in_esame != NULL)
    {
        printf("%d\n", in_esame->valore);
        in_esame = in_esame->prossimo; //perdo l'inizio
    }

    return 0;
}
```

42
66
88
0

Nella precedente implementazione, viene:

- allocato il primo elemento della lista, il cui indirizzo finisce nel puntatore `inizio`
- in maniera iterativa, ad ogni iterazione, viene:
 - assegnato il valore al campo `valore` del record
 - allocato il *successivo* elemento della lista, il cui indirizzo finisce nel puntatore `successivo` dell'elemento della lista in esame

Tale implementazione è sbagliata, in quanto ad ogni iterazione viene allocato spazio per un ipotetico elemento successivo, senza considerare se tale elemento esista realmente o meno

⇒

viene allocato un elemento in più.

Una strategia migliore consiste nel generare, ad ogni iterazione, l'elemento "attuale" su cui lavorare, e non il successivo.

In tale impostazione, bisogna tener conto che il primo elemento, che a differenza dei successivi deve andare nel puntatore `inizio`.

Posso utilizzare un puntatore che tiene traccia del predecessore (ossia l'ultimo elemento della lista fino a quel momento inserito).

Questo serve essenzialmente a 2 cose:

- capire se l'elemento allocato è il primo elemento della lista o meno
 - se è il primo della lista (ossia non ha predecessore), l'indirizzo va posto nel puntatore `inizio`
 - se non lo è (ossia esiste un predecessore), va posto come elemento successivo al precedente
- avere già pronto l'indirizzo del predecessore da collegare all'indirizzo dell'elemento attuale.

```
In [36]: /* Problema: copiare il contenuto di un array in una lista concatenata */
// strategia 1

#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

/* in questa implementazione, un ciclo alloca lo spazio materiale
   per inserire l'elemento corrente
 */

int main()
{
    int valori[] = {42,66,88};
    const int n = 3;

    struct elemento* inizio = NULL;
    struct elemento* in_esame = NULL;

    for(int i = 0; i < n; i++)
    {
        // definisco il nuovo elemento da inserire
        // alloco lo spazio
        in_esame = malloc(sizeof(struct elemento));
        // inserisco il valore
        in_esame->valore = valori[i];
        // presuppongo che non ci siano ulteriori elementi
        in_esame->prossimo = NULL;

        // se i == 0,
        // vuol dire che l'elemento che si sta inserendo è il primo
        if (i == 0)
        {
            inizio = in_esame;
        }
        else // se i>0, la lista non è vuota
        {
            // scorro la lista per individuare l'ultimo elemento
            struct elemento* cursore = inizio; // per scorrere la lista
            while(cursore->prossimo != NULL)
                cursore = cursore->prossimo;
            // colloco il nuovo elemento all'ultimo nodo della lista
            cursore->prossimo = in_esame;
        }
    }

    in_esame = inizio;
    while(in_esame != NULL)
    {
        printf("%d\n", in_esame->valore);
        in_esame = in_esame->prossimo;
    }

    return 0;
}
```

42
66
88

Nella strategia precedente, ad ogni iterazione viene:

- allocato lo spazio per il nuovo elemento
- cercata la posizione giusta nella lista.
 - Nel caso sia la prima iterazione (`i==0`), allora l'elemento da inserire è il primo
⇒ viene modificato il puntatore `inizio`
 - altrimenti

- si scorre la lista fino ad arrivare all'ultimo elemento
- si collega il nuovo elemento al precedente

Tale strategia ha come principale inconveniente il dover scorrere la lista ad ogni inserimento. E' possibile evitarlo?

```
In [37]: /* Problema: copiare il contenuto di un vettore in una lista concatenata */
// Strategia 2
// Ad ogni iterazione, tengo traccia dell'ultimo elemento inserito
#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

int main()
{
    int valori[] = {42,66,88};
    const int n = 3;

    struct elemento* inizio = NULL;
    struct elemento* in_esame = NULL;
    struct elemento* predecessore = NULL; // punta all'elemento precedente
                                            // della lista.
                                            // se la lista è vuota,
                                            // contiene NULL
    for(int i = 0; i < n; i=i+1)
    {
        //alloco lo spazio
        in_esame = malloc(sizeof(struct elemento));

        // inserisco il valore
        in_esame->valore = valori[i];

        // presuppongo che non ci siano ulteriori elementi
        in_esame->prossimo = NULL;

        // se non esiste un predecessore,
        // vuol dire che l'elemento che si sta inserendo è il primo
        if (predecessore == NULL)
        {
            inizio = in_esame;
        }
        else // se esiste un predecessore
        {
            // collego l'elemento appena creato al precedente
            predecessore->prossimo = in_esame;
        }

        // aggiorno il predecessore con l'indirizzo dell'elemento nuovo,
        // così che possa essere utilizzato alla prossima iterazione
        // come punto di inserimento di un eventuale nuovo elemento
        predecessore = in_esame;
    }

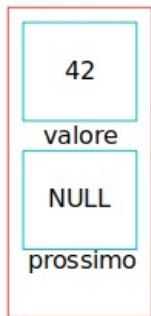
    in_esame = inizio;
    while(in_esame != NULL)
    {
        printf("%d\n", in_esame->valore);
        in_esame = in_esame->prossimo;
    }
}

return 0;
}
```

42
66
88



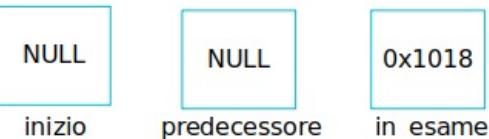
0x1018



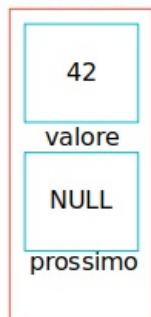
```
in_esame = (struct elemento*) malloc(sizeof(struct elemento));
in_esame->valore = valori[i];
in_esame->prossimo = NULL;
```

0

i



0x1018



```
if (predecessore == NULL)
{
    inizio = in_esame;
}

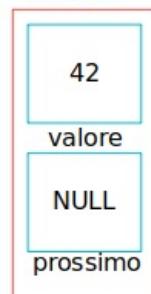
predecessore = in_esame;
```

0

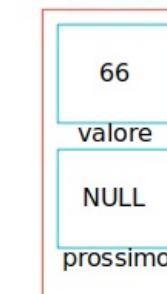
i



0x1018



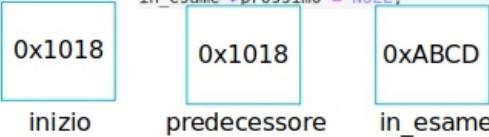
0xABCD

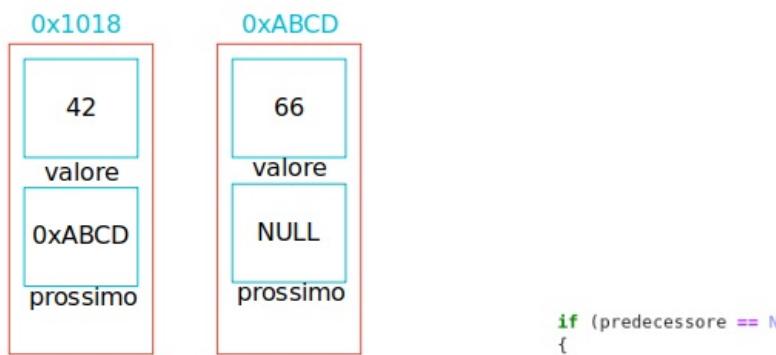


1

i

```
in_esame = (struct elemento*) malloc(sizeof(struct elemento));
in_esame->valore = valori[i];
in_esame->prossimo = NULL;
```

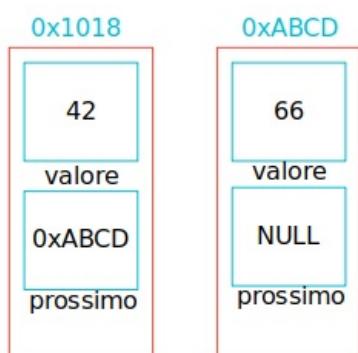




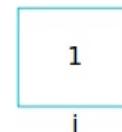
```

if (predecessore == NULL)
{
    ...
}
else
{
    predecessore->prossimo = in_esame;
}
predecessore = in_esame;

```



predecessore = in_esame;



E così via...

Nel caso precedente, abbiamo utilizzato un puntatore `predecessore` per "scegliere" dove mettere l'indirizzo dell'elemento appena generato.

Alternative?

Un'altra alternativa è quella di utilizzare un **puntatore a puntatore** che sia sempre aggiornato sulla posizione corretta in cui inserire l'indirizzo dell'elemento allocato.

In altri termini,

Definiamo `in_esame` questa volta come *puntatore a puntatore*.

- inizialmente `in_esame` punterà al puntatore `inizio`, ossia `in_esame = &inizio` (i.e. `*in_esame = ...`)
⇒
`inizio=...).`
- dalla seconda iterazione in poi, `in_esame` punterà al puntatore `prossimo` del record appena generato

In questo modo, potrà essere usato `in_esame` per inserire l'indirizzo di ogni record nella posizione corretta. Questo perché:

- `in_esame` inizialmente punterà ad `inizio`
⇒

`*in_esame = malloc(...)`

⇒

`inizio = malloc(...)`

- nelle iterazioni successive, `in_esame` punterà al campo `prossimo` dell'ultimo elemento della lista,
⇒

`*in_esame = malloc(...)`

⇒

`predecessore->prossimo = malloc(...)`

```

In [1]: /* Problema: copiare il contenuto di un vettore in una lista concatenata */
// Strategia 3

```

```

// utilizzo di un puntatore a puntatore
#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

int main()
{
    int valori[] = {42,66,88};
    const int n = 3;

    struct elemento* inizio = NULL;

    // definisco un puntatore a puntatore che,
    // inizialmente, punta al puntatore "inizio"
    struct elemento** in_esame = &inizio;

    for(int i=0; i < n; i=i+1)
    {
        // nella prima iterazione, accedere a "*in_esame"
        // è equivalente ad accedere ad "inizio".
        // Nelle iterazioni successive invece, accedere a "*in_esame"
        // corrisponde ad accedere al puntatore "prossimo"
        // dell'ultimo elemento nella lista
        *in_esame = malloc(sizeof(struct elemento));

        (*in_esame) -> valore = valori[i];
        (*in_esame) -> prossimo = NULL;

        // aggiorno il puntatore in esame, facendolo puntare al puntatore
        // "prossimo" del record appena allocato.
        // In questo modo in esame è pronto per l'iterazione successiva (se ci sarà)
        in_esame = &(*in_esame)->prossimo;
    }

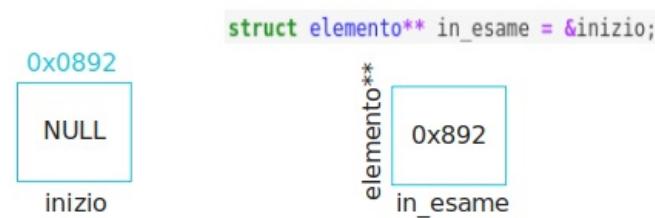
    in_esame = &inizio;
    while(*in_esame != NULL)
    {
        printf("%d\n", (*in_esame)->valore);
        in_esame = &(*in_esame)->prossimo;
    }

    return 0;
}

```

42
66
88

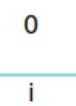
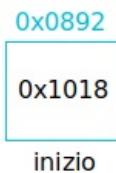
Inizio



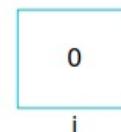
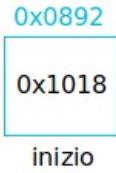
Prima iterazione



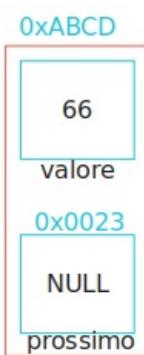
```
*in_esame = (struct elemento*) malloc(sizeof(struct elemento));
(*in_esame) -> valore = valori[i];
(*in_esame) -> prossimo = NULL;
```



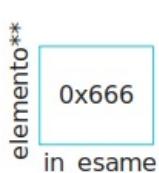
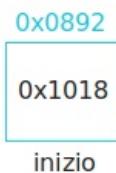
```
in_esame = &(*in_esame)->prossimo;
```

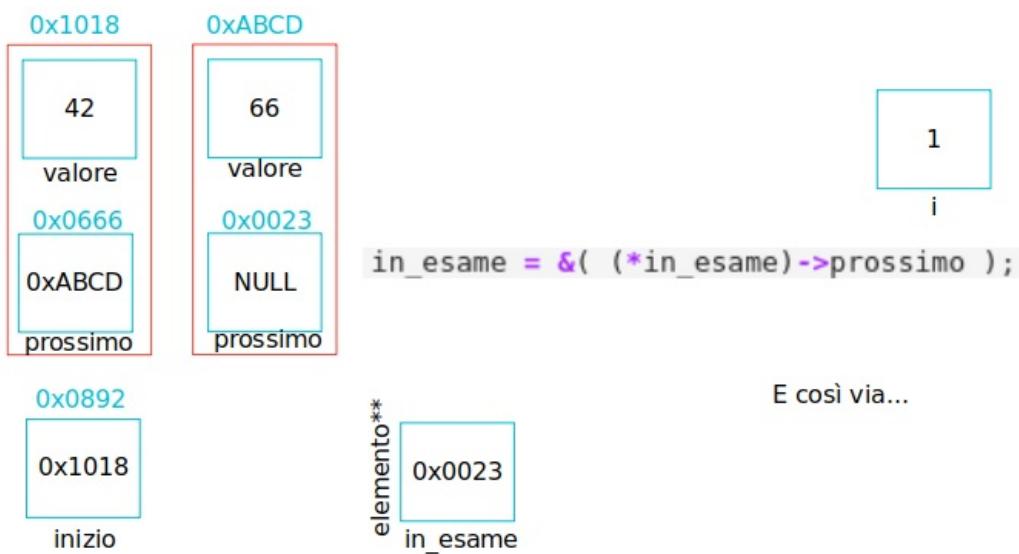


Seconda iterazione



```
*in_esame = (struct elemento*) malloc(sizeof(struct elemento));
(*in_esame) -> valore = valori[i];
(*in_esame) -> prossimo = NULL;
```





Buone norme

- Prima di concludere il programma, **liberare tutto lo spazio allocato**.

```

struct elemento* in_esame = inizio;
while(in_esame != NULL)
{
    free(in_esame);
    in_esame = in_esame->prossimo;
}
  
```

perchè non va bene?

(una) versione corretta:

```

struct elemento* in_esame = inizio;
while(in_esame != NULL)
{
    struct elemento* prox = in_esame->prossimo;
    free(in_esame);
    in_esame = prox;
}
inizio = NULL; // buona norma
  
```

Problema: Scrivere un programma che permetta all'utente di inserire degli interi, uno dietro l'altro, in una lista concatenata. Dare la possibilità all'utente di visualizzare tale lista.

Considerazioni:

- Ricordiamo che una lista è composta da:
 - un *puntatore* al primo nodo;
 - un insieme di nodi concatenati. Ogni nodo avrà al suo interno un puntatore al nodo successivo.
- per l'inserimento in lista devo quindi considerare i seguenti casi:
 - la lista è vuota, quindi il nodo che si sta inserendo è il primo
⇒ devo modificare il puntatore al primo nodo
 - la lista non è vuota
⇒ non devo modificare il puntatore al primo nodo, ma il puntatore interno dell'ultimo nodo

Diverse strategie:

```

In [ ]: /*
Scrivere un programma che permetta all'utente di inserire degli interi, uno dietro l'altro, in una lista concatenata.
Dare la possibilità all'utente di visualizzare tale lista.
*/
// Strategia 1
#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
}
  
```

```

};

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->{%d}", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}

void deallocate(struct elemento* start)
{
    while(start != NULL)
    {
        struct elemento* prox = start->prossimo;
        free(start);
        start = prox;
    }
}

/*
la funzione insert inserisce in una lista il valore val.
Prende in input:
- l'indirizzo del puntatore alla testa della lista
- il valore da inserire

La funzione quindi, se necessario,
modificherà direttamente il puntatore alla testa della lista
(in quanto passato per indirizzo)
*/
void insert(struct elemento** start, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo;
    nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. scorro la lista fino a raggiungere l'ultimo elemento
    struct elemento* in_esame   = *start;
    struct elemento* precedente = NULL; // tengo traccia del predecessore

    while(in_esame!= NULL)
    {
        precedente = in_esame;
        in_esame   = in_esame->prossimo;
    }

    // se non c'è predecessore, vuol dire che si sta inserendo il primo elemento
    if( precedente == NULL )
        *start = nuovo; //se la lista è vuota,
                        //inserisco il nuovo elemento come primo elemento
    else
        precedente->prossimo = nuovo;
}

int main()
{
    struct elemento* start_lista = NULL;

    int scelta = -1;
    while(scelta != 0)
    {
        printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
        scanf("%d", &scelta);

        int val;
        switch(scelta)
        {
            case 1:
                printf("valore da inserire: ");
                scanf("%d", &val);
                insert(&start_lista, val);
                break;
            case 2:
                print(start_lista);
                break;
            case 3:
                deallocate(start_lista);
                start_lista = NULL;
                break;
            case 0:
                deallocate(start_lista);
                start_lista = NULL;
        }
    }
}

```

```

        break;
    default:
        printf("scelta sbagliata! Ripetere\n");
    }
}

return 0;
}

```

Vediamo più nel dettaglio cosa succede:

Prima invocazione di `insert(...)`

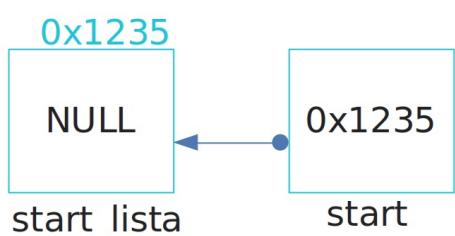
- all'atto della **prima** invocazione di `insert(...)` (i.e. lista vuota)

```

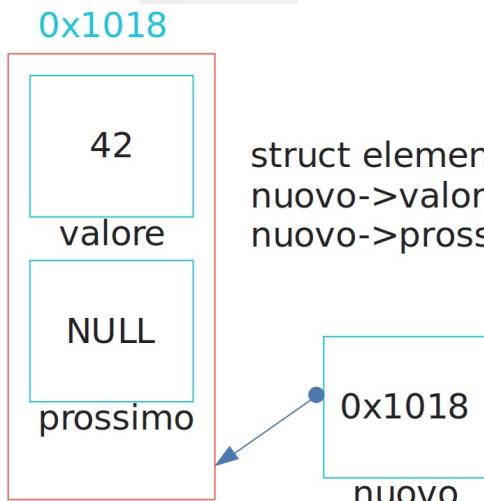
void insert(struct elemento** start, int val)
{
...
}

int main()
{
...
insert(&start_lista, val);
...
}

```



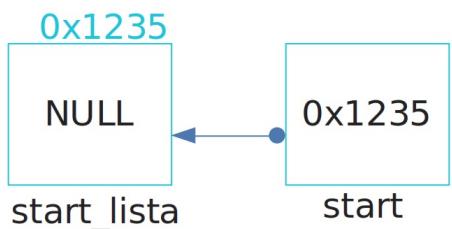
- nella funzione `insert(...)`, viene allocato ed inizializzato il nuovo nodo



```

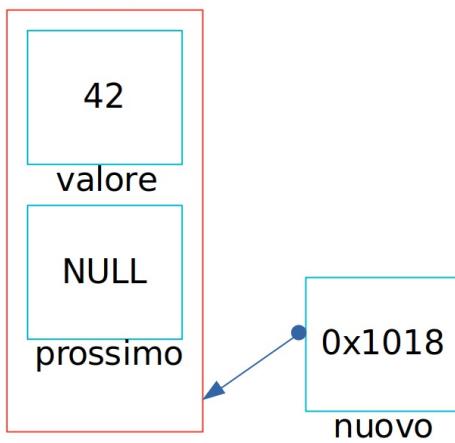
struct elemento* nuovo = malloc(sizeof(struct elemento));
nuovo->valore      = val;
nuovo->prossimo     = NULL;

```



- si scorre la lista fino alla fine. Dato che è vuota, `in_esame` e `precedente` restano entrambe a `NULL`

0x1018



```
struct elemento* in_esame = *start;
struct elemento* precedente = NULL;

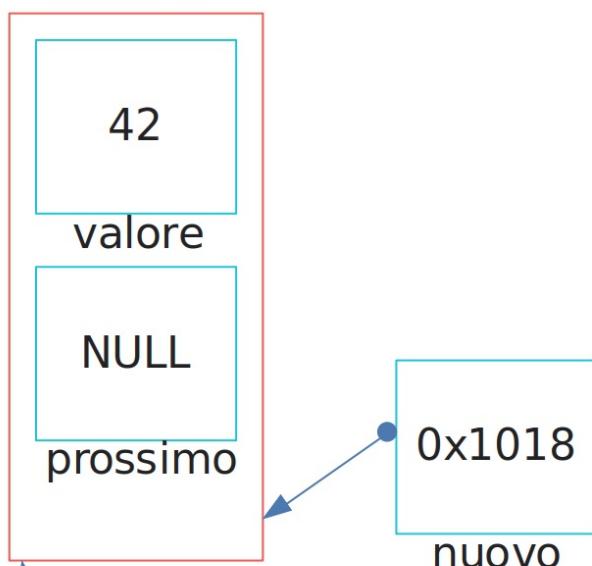
while(in_esame != NULL)
{
    precedente = in_esame;
    in_esame = in_esame->prossimo;
}
```

0x1235



- si controlla il valore di `precedente`. Dato che è `NULL`, l'elemento che si sta inserendo è il primo
⇒ si aggiorna il puntatore `start_list` (attraverso la dereferenziazione del puntatore `start`)

0x1018



```
if( precedente == NULL )
    *start = nuovo;
```

0x1235

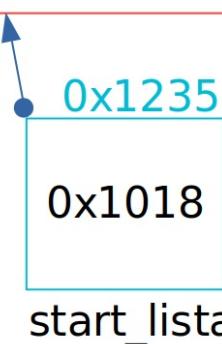
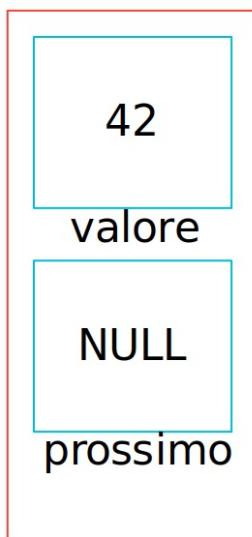
0x1018
start_list

0x1235
start

NULL
in_esame
NULL
precedente

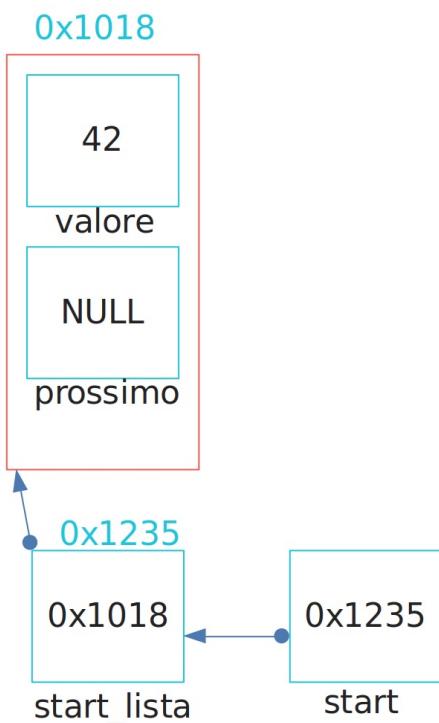
- situazione al termine della prima invocazione di `insert(...)`

0x1018



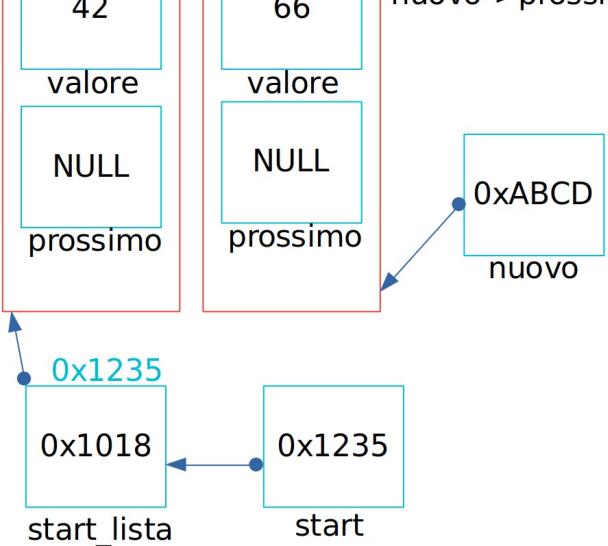
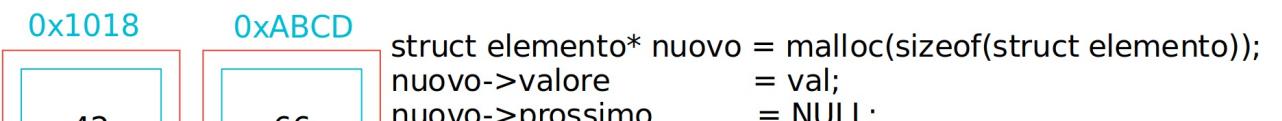
Seconda invocazione di `insert(...)`

- all'atto della seconda invocazione, la situazione è la seguente:

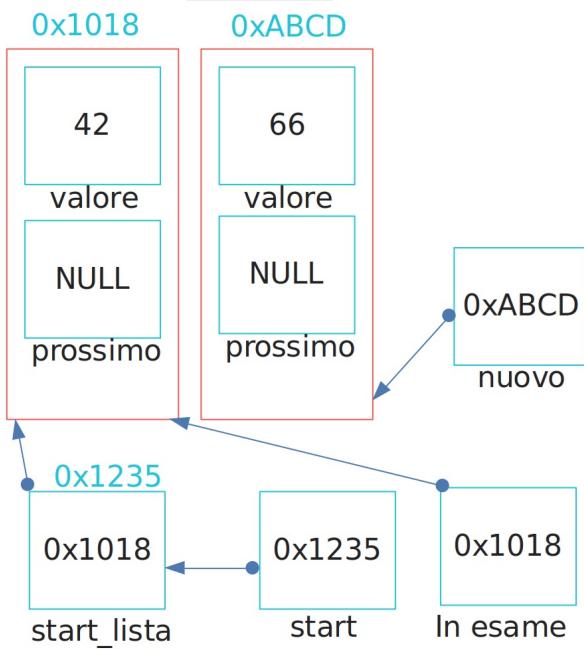


```
void insert(struct elemento** start, int val)
{
...
}
...
int main()
{
...
insert(&start_lista, val);
...
}
```

- viene allocato ed inizializzato il nuovo nodo



- si scorre la lista fino alla fine.
 - prima di entrare nel `while(...)`



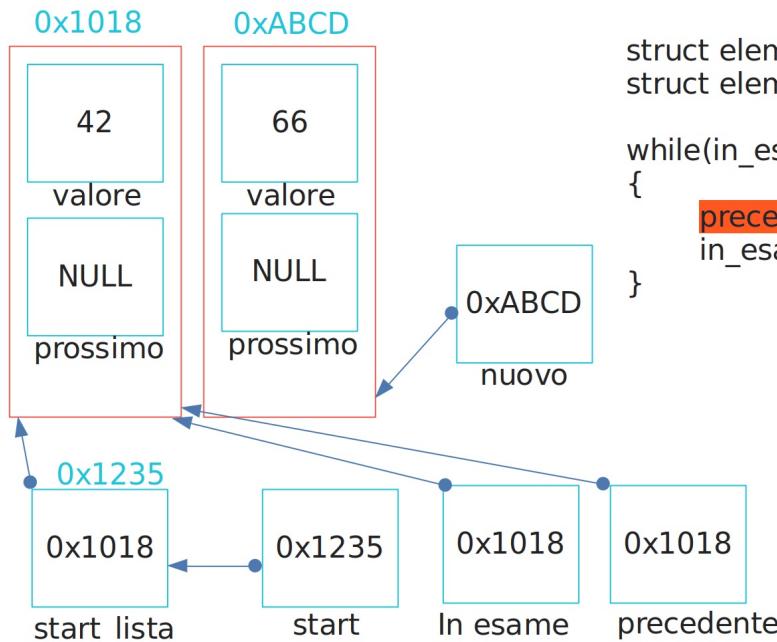
```

struct elemento* in_esame = *start;
struct elemento* precedente = NULL;

while(in_esame != NULL)
{
    precedente = in_esame;
    in_esame = in_esame->prossimo;
}

```

- durante la prima iterazione del `while(...)`

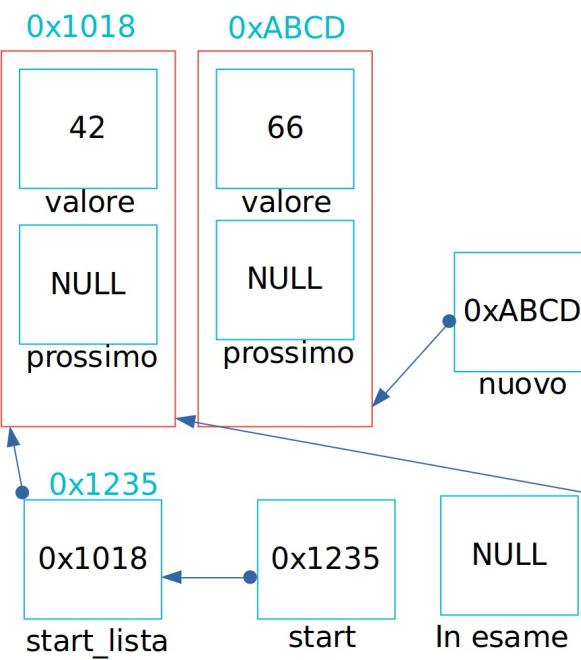


```

struct elemento* in_esame = *start;
struct elemento* precedente = NULL;

while(in_esame != NULL)
{
    precedente = in_esame;
    in_esame = in_esame->prossimo;
}

```



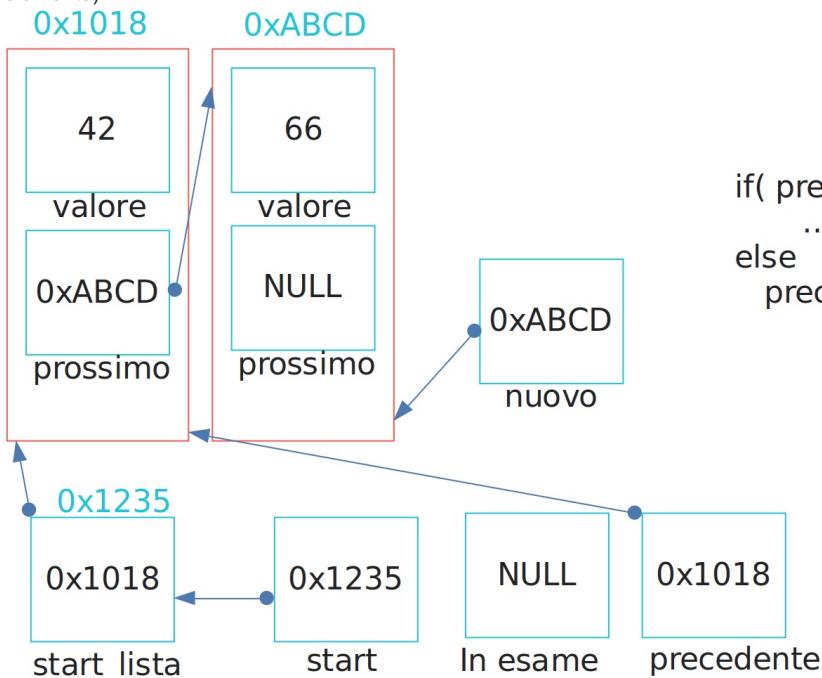
```

struct elemento* in_esame = *start;
struct elemento* precedente = NULL;

while(in_esame!= NULL)
{
    precedente = in_esame;
    in_esame = in_esame->prossimo;
}

```

- si controlla il valore di `precedente`. Dato che **non** è `NULL`, l'elemento che si sta inserendo **non** è il primo
⇒ si aggiorna il successore dell'ultimo elemento (attraverso la dereferenziazione del puntatore `precedente` che punterà all'ultimo elemento)

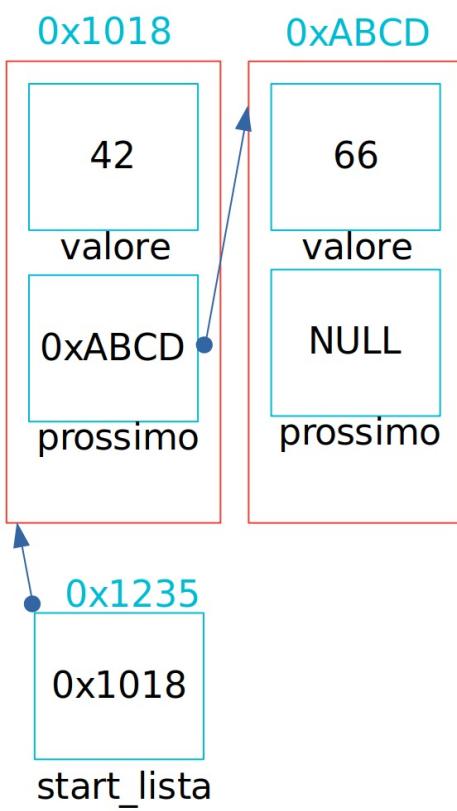


```

if( precedente == NULL )
    ...
else
    precedente->prossimo = nuovo;

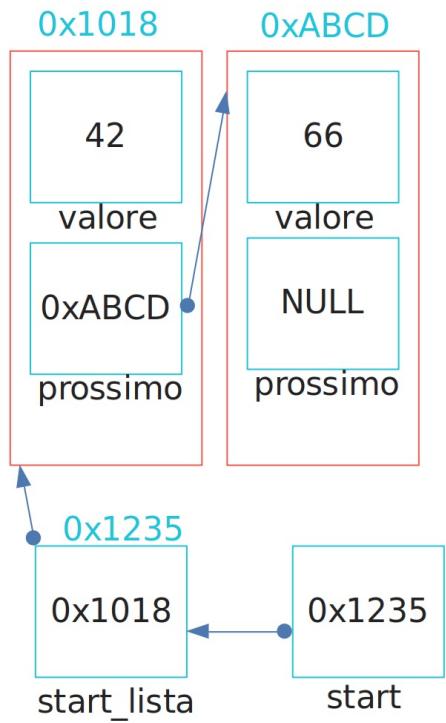
```

- situazione al termine della seconda invocazione di `insert(...)`



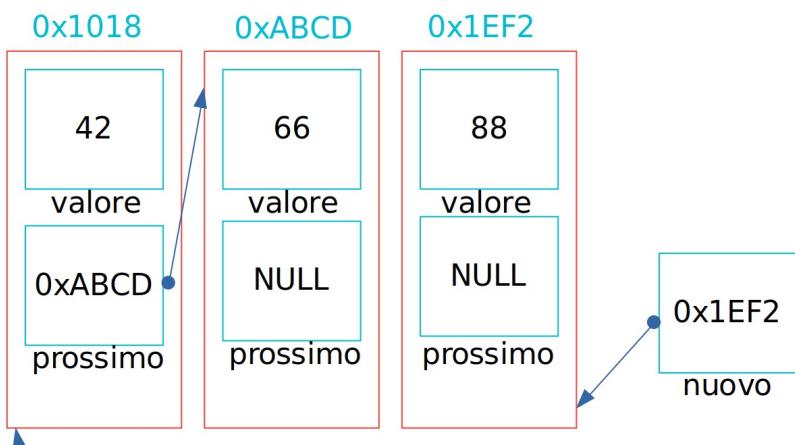
Terza invocazione di `insert(...)`

- all'atto della terza invocazione, la situazione è la seguente



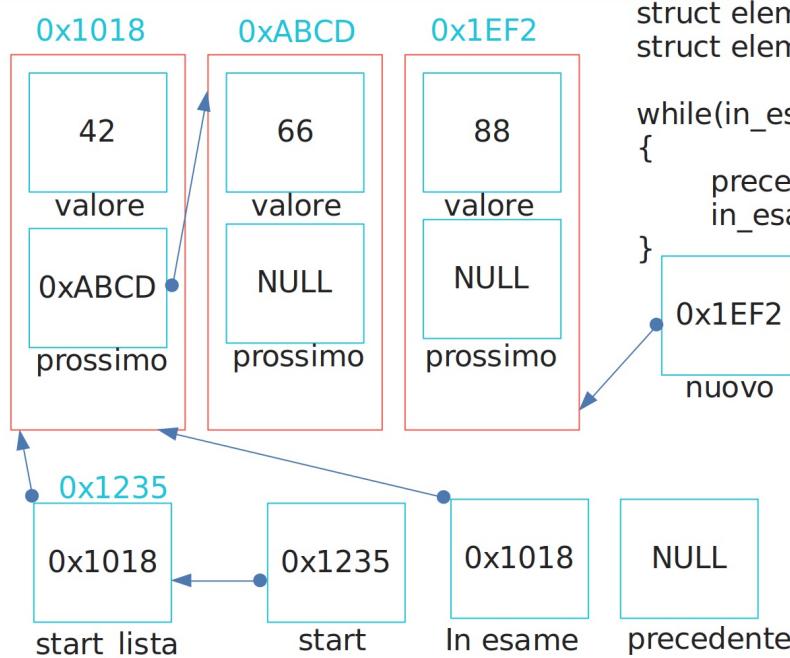
```
void insert(struct elemento** start, int val)
{
...
}
...
int main()
{
...
insert(&start_lista, val);
...
}
```

- viene allocato ed inizializzato il nuovo nodo



```
struct elemento* nuovo = malloc(sizeof(struct elemento));
nuovo->valore      = val;
nuovo->prossimo     = NULL;
```

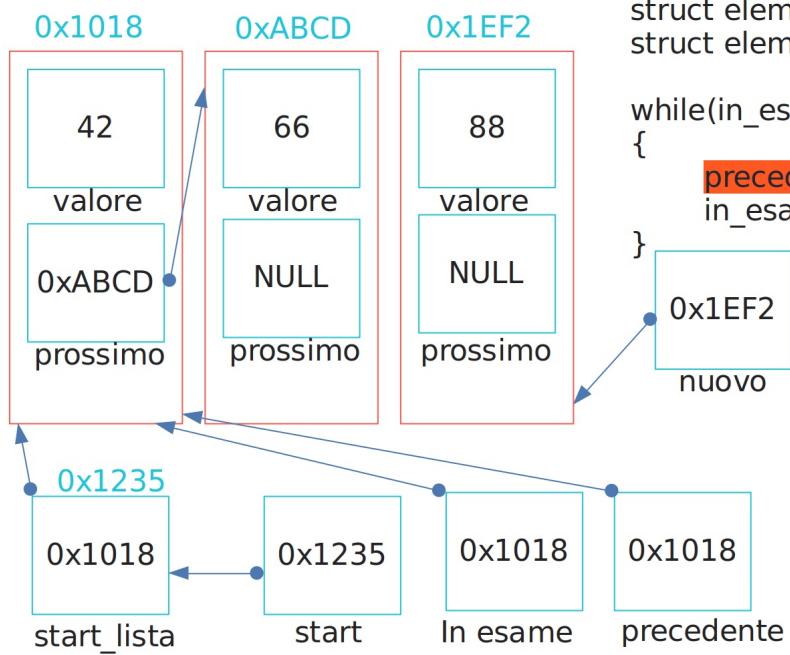
- si scorre la lista fino alla fine.
 - situazione prima di entrare nel ciclo



```
struct elemento* in_esame = *start;
struct elemento* precedente = NULL;
```

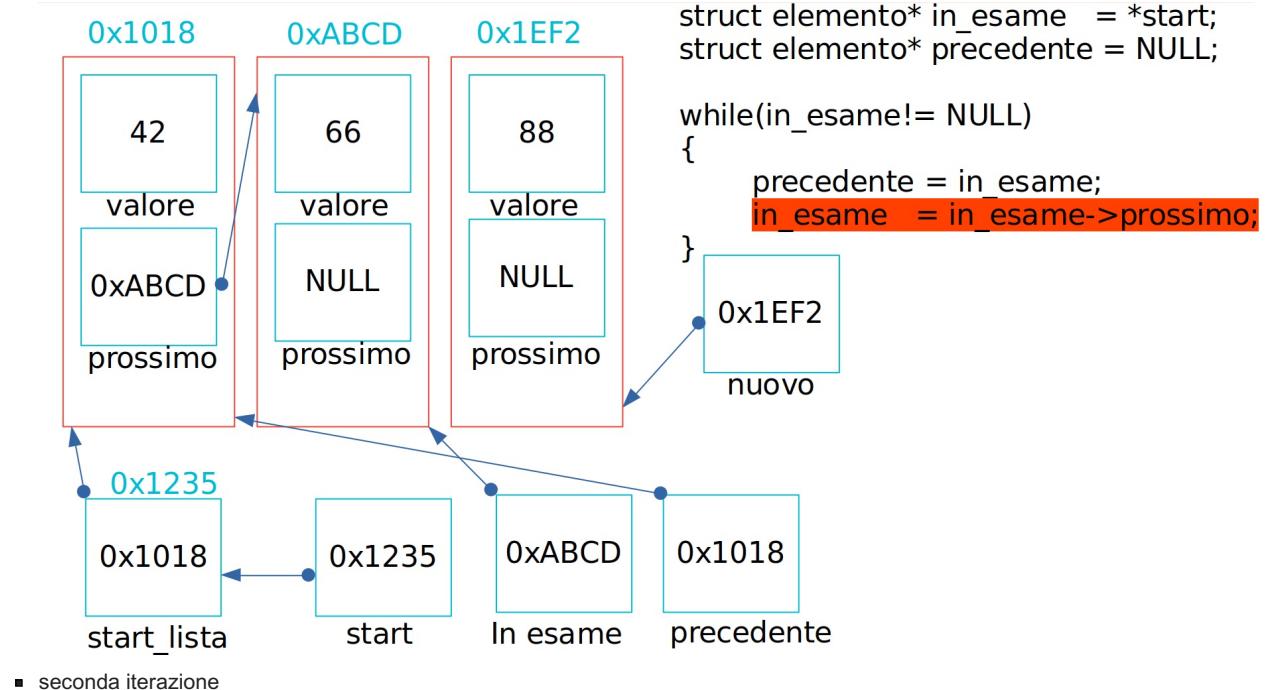
```
while(in_esame!= NULL)
{
    precedente = in_esame;
    in_esame   = in_esame->prossimo;
}
```

- prima iterazione

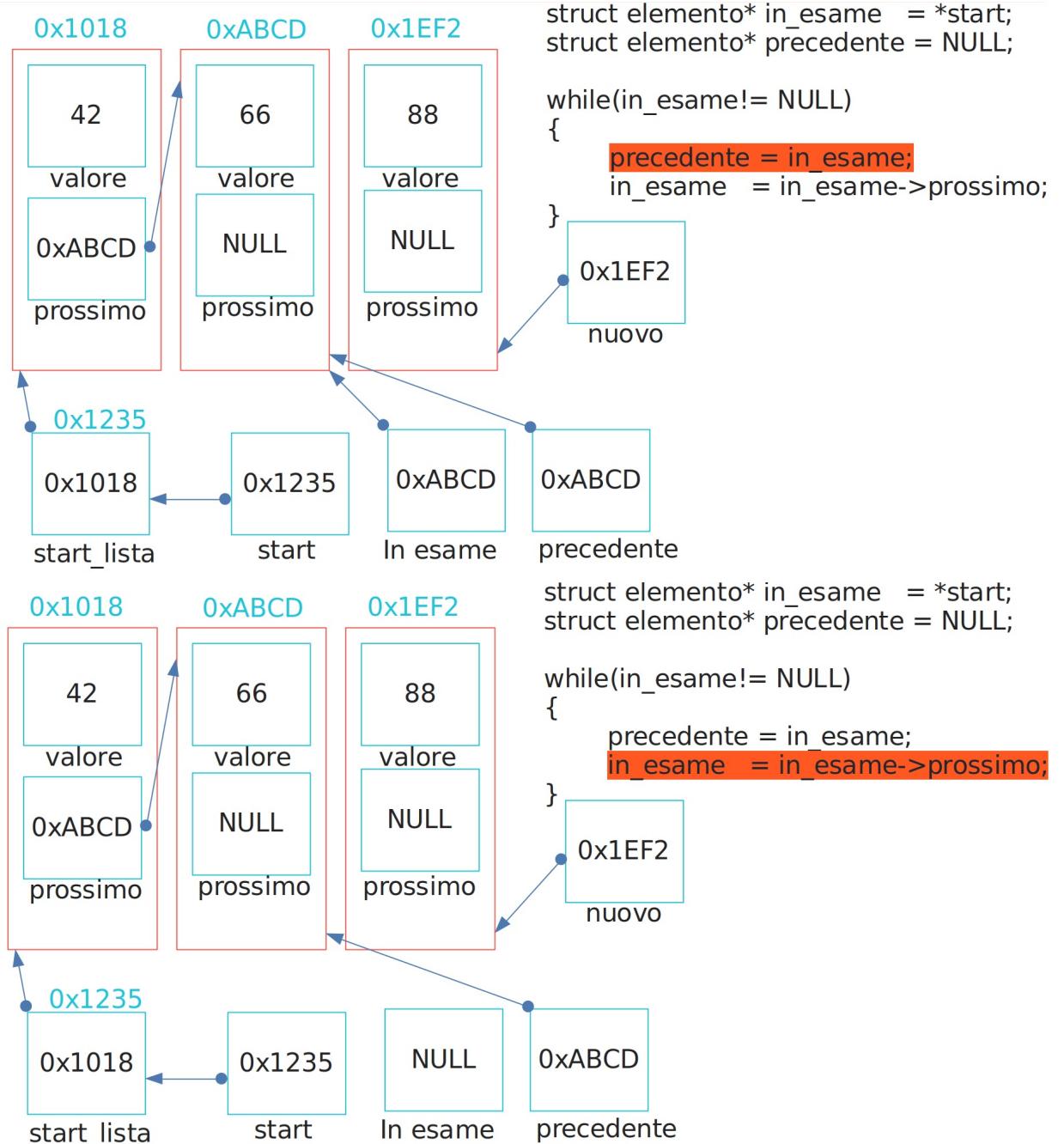


```
struct elemento* in_esame = *start;
struct elemento* precedente = NULL;
```

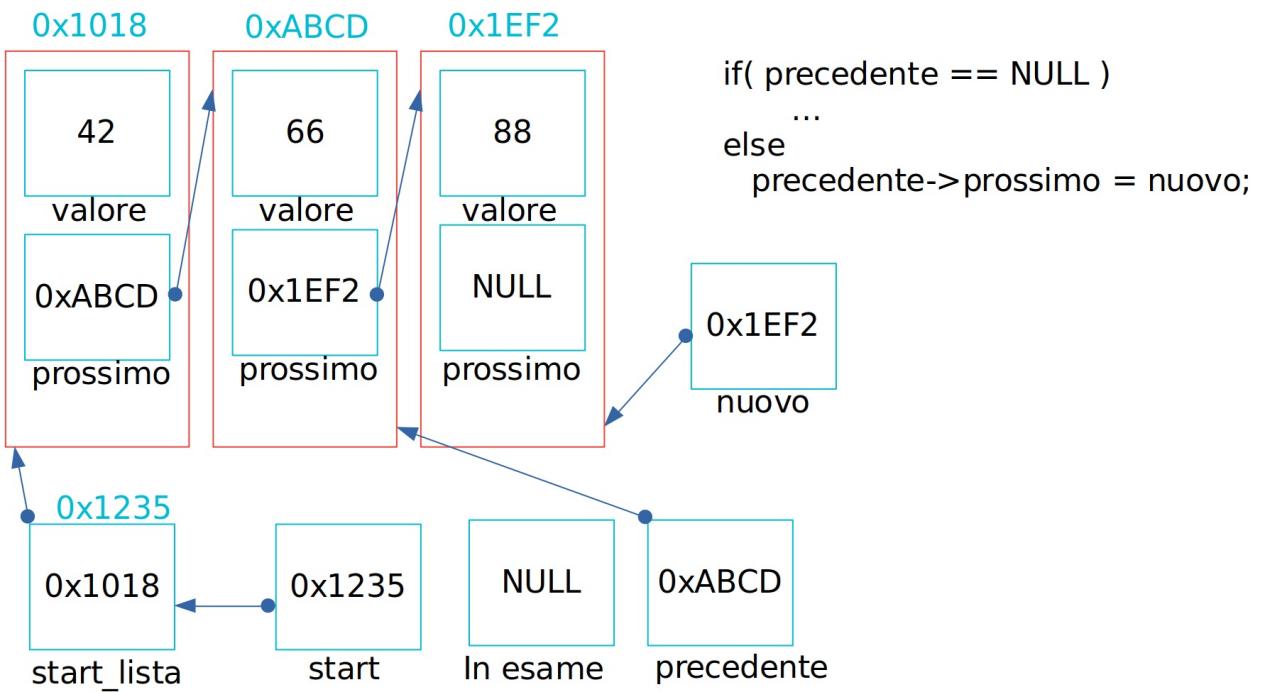
```
while(in_esame!= NULL)
{
    precedente = in_esame;
    in_esame   = in_esame->prossimo;
}
```



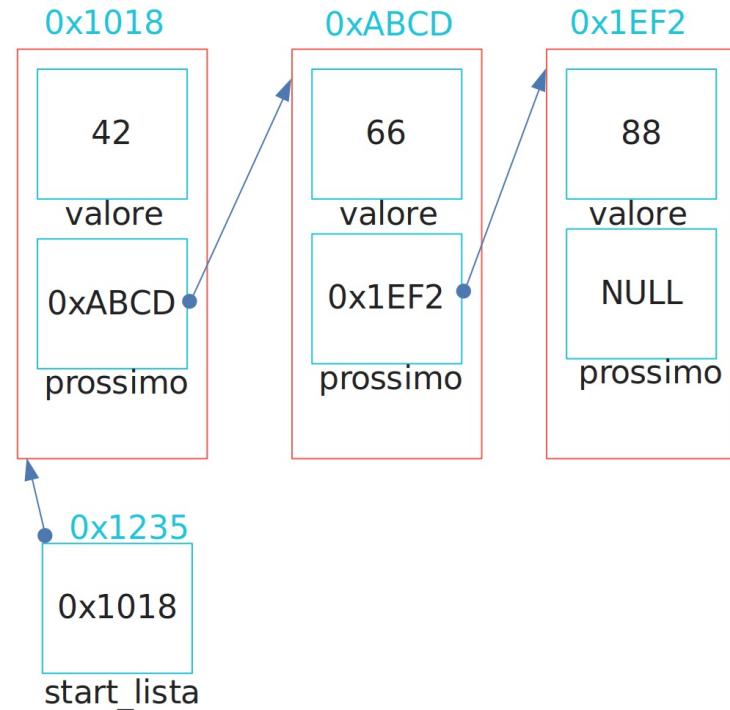
■ seconda iterazione



- si controlla il valore di `precedente`. Dato che **non** è `NULL`, l'elemento che si sta inserendo **non** è il primo
⇒ si aggiorna il successore dell'ultimo elemento (attraverso la dereferenziazione del puntatore `precedente` che punterà all'ultimo elemento)



- situazione al termine della terza invocazione di `insert(...)`



```
In []:
/*
Scrivere un programma che permetta all'utente di inserire degli interi
ed inserirli, uno dietro l'altro, in una lista concatenata.
Dare la possibilità all'utente di visualizzare tale lista.
*/
// strategia 2 (simile alla precedente)

#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->%d", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}
```

```

void deallocate(struct elemento* start)
{
    while(start != NULL)
    {
        struct elemento* prox = start->prossimo;
        free(start);
        start = prox;
    }
}

/*
la funzione insert inserisce in una lista il valore val.
Prende in input:
- l'indirizzo del puntatore alla testa della lista
- il valore da inserire

La funzione quindi, se necessario,
modificherà direttamente il puntatore alla testa della lista
(in quanto passato per indirizzo)
*/
void insert(struct elemento** start, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;
    //
    // 2. se la lista è vuota, inserisco il nuovo elemento
    // come primo elemento
    if(*start == NULL)
    {
        *start = nuovo;
        return;
    }

    // 3. Se la lista non è vuota, cerco dove inserirlo
    struct elemento* in_esame = *start;

    // scorrila fino a raggiungere l'ultimo elemento
    while(in_esame->prossimo!= NULL)
        in_esame = in_esame->prossimo;
    // "in esame" punterà all'ultimo elemento
    in_esame->prossimo = nuovo;
}

int main()
{
    struct elemento* start_lista = NULL;

    int scelta = -1;
    while(scelta != 0)
    {
        printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
        scanf("%d", &scelta);

        int val;
        switch(scelta)
        {
            case 1:
                printf("valore da inserire: ");
                scanf("%d", &val);
                insert(&start_lista, val);
                break;
            case 2:
                print(start_lista);
                break;
            case 3:
                deallocate(start_lista);
                start_lista = NULL;
                break;
            case 0:
                deallocate(start_lista);
                start_lista = NULL;
                break;
            default:
                printf("scelta sbagliata! Ripetere\n");
        }
    }

    return 0;
}

```

In []: /*
Scrivere un programma che permetta all'utente di inserire degli interi
ed inserirli, uno dietro l'altro, in una lista concatenata.

```

Dare la possibilità all'utente di visualizzare tale lista.
*/
// Strategia 3: in questa strategia, viene sfruttato il valore
// di ritorno per modificare il puntatore alla testa della lista

#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->%d", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}

void deallocate(struct elemento* start)
{
    while(start != NULL)
    {
        struct elemento* prox = start->prossimo;
        free(start);
        start = prox;
    }
}

/*
la funzione insert inserisce in una lista il valore val.
Prende in input:
- l'indirizzo della testa della lista
- il valore da inserire

Restituisce:
- l'indirizzo della testa della lista.

La funzione quindi NON modificherà direttamente
il puntatore alla testa della lista (in quanto passato per copia).
Per farlo, si deve sfruttare il valore di ritorno
*/
struct elemento* insert(struct elemento* start, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    //2. se la lista è vuota, restituisco il nuovo elemento
    if(start == NULL)
        return nuovo;

    // 3. se la lista non è vuota, cerco dove inserirlo
    // scorrila fino a raggiungere l'ultimo elemento
    struct elemento* in_esame = start;
    while( in_esame->prossimo!= NULL)
        in_esame = in_esame->prossimo;
    // "in_esame" punterà all'ultimo elemento della lista

    in_esame->prossimo = nuovo;

    return start;
}

int main()
{
    struct elemento* start_lista = NULL;

    int scelta = -1;
    while(scelta != 0)
    {
        printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
        scanf("%d", &scelta);

        int val;
        switch(scelta)
        {
            case 1:
                printf("valore da inserire: ");
                scanf("%d", &val);
                start_lista = insert(start_lista, val);

```

```

        break;
    case 2:
        print(start_lista);
        break;
    case 3:
        deallocate(start_lista);
        start_lista = NULL;
        break;
    case 0:
        deallocate(start_lista);
        start_lista = NULL;
        break;
    default:
        printf("scelta sbagliata! Ripetere\n");
}

}

return 0;
}

```

In []: /*
Scrivere un programma che permetta all'utente di inserire degli interi
ed inserirli, uno dietro l'altro, in una lista concatenata.
Dare la possibilità all'utente di visualizzare tale lista.
*/

// Strategia 4: utilizzo di un altro puntatore a puntatore per scorrere la lista

```

#include <stdio.h>

struct elemento {
    int valore;
    struct elemento* prossimo;
};

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->%d", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}

void deallocate(struct elemento** start)
{
    while(*start != NULL)
    {
        struct elemento* prox = (*start)->prossimo;
        free(*start);
        *start = prox;
    }
    *start = NULL;
}

void insert(struct elemento** start, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo
    // della lista,
    // ma ad un puntatore ad un nodo
    struct elemento** p_dest = start;
    // se la lista non è vuota,
    // scorri la fino a raggiungere l'ultimo elemento
    while( *p_dest!= NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo elemento (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo elemento della lista
    *p_dest = nuovo;
}

int main()
{
    struct elemento* start_lista = NULL;
    int scelta = -1;
}
```

```

while(scelta != 0)
{
    printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
    scanf("%d", &scelta);

    int val;
    switch(scelta)
    {
        case 1:
            printf("valore da inserire: ");
            scanf("%d", &val);
            insert(&start_lista, val);
            break;
        case 2:
            print(start_lista);
            break;
        case 3:
            deallocate(&start_lista);
            break;
        case 0:
            deallocate(&start_lista);
            break;
        default:
            printf("scelta sbagliata! Ripetere\n");
    }
}

return 0;
}

```

Vediamo più nel dettaglio cosa succede:

Prima invocazione di `insert(...)`

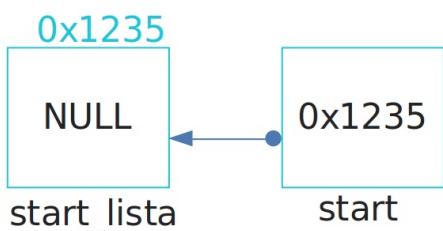
- all'atto della **prima** invocazione di `insert(...)` (i.e. lista vuota)

```

void insert(struct elemento** start, int val)
{
    ...
}

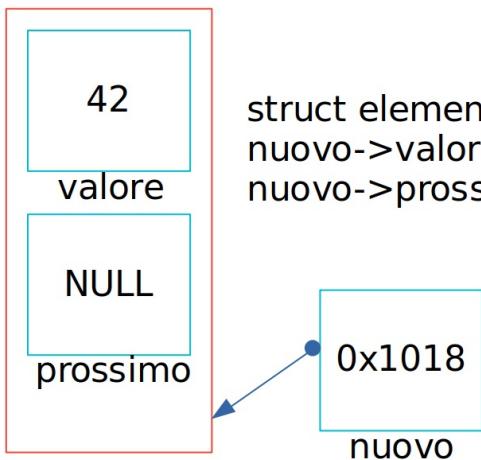
int main()
{
    ...
    insert(&start_lista, val);
    ...
}

```



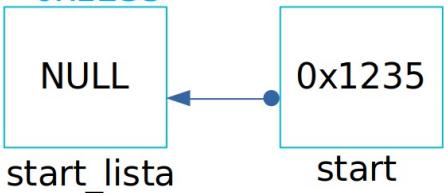
- nella funzione `insert(...)`, viene allocato ed inizializzato il nuovo nodo

0x1018



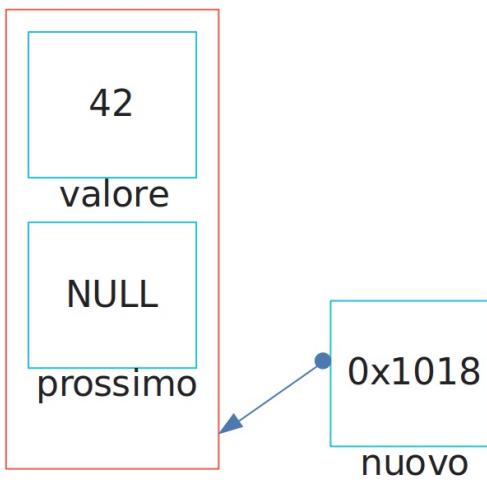
```
struct elemento* nuovo = malloc(sizeof(struct elemento));
nuovo->valore = val;
nuovo->prossimo = NULL;
```

0x1235



- viene allocato un cursore `p_dest` di tipo **puntatore a puntatore** che punta al puntatore che dovrà contenere la testa della lista

0x1018

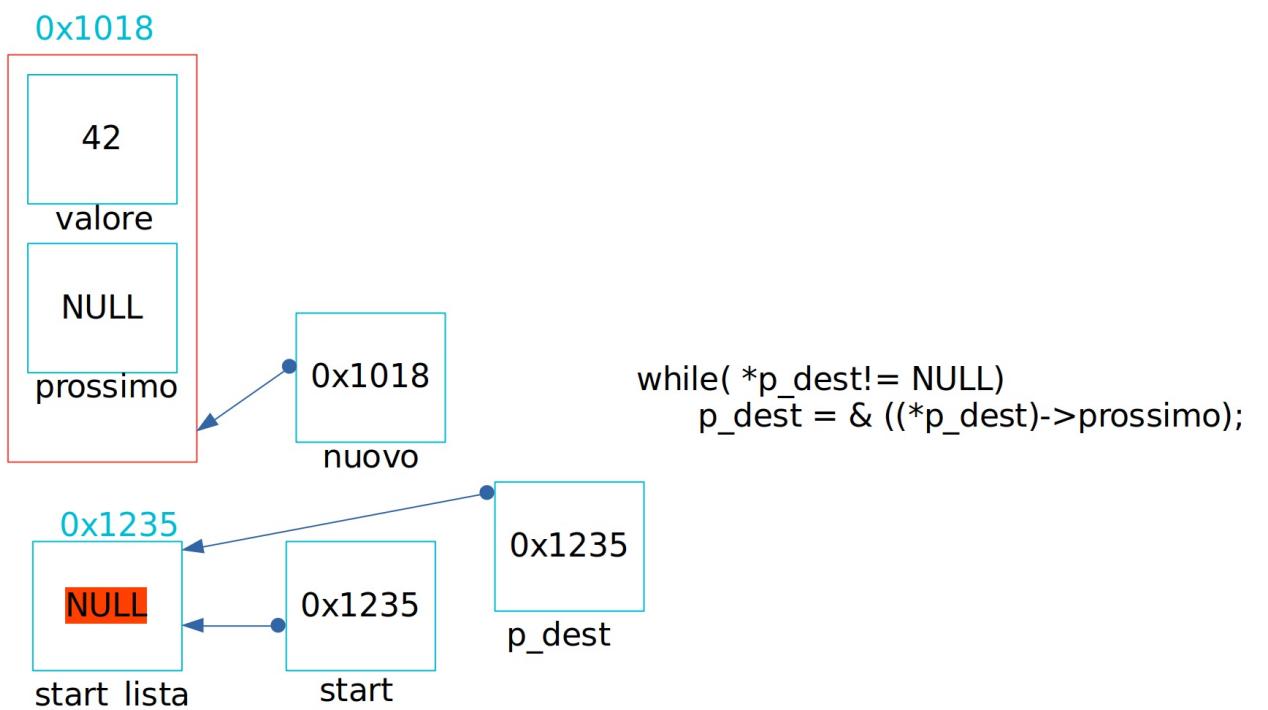


```
struct elemento** p_dest = start;
```

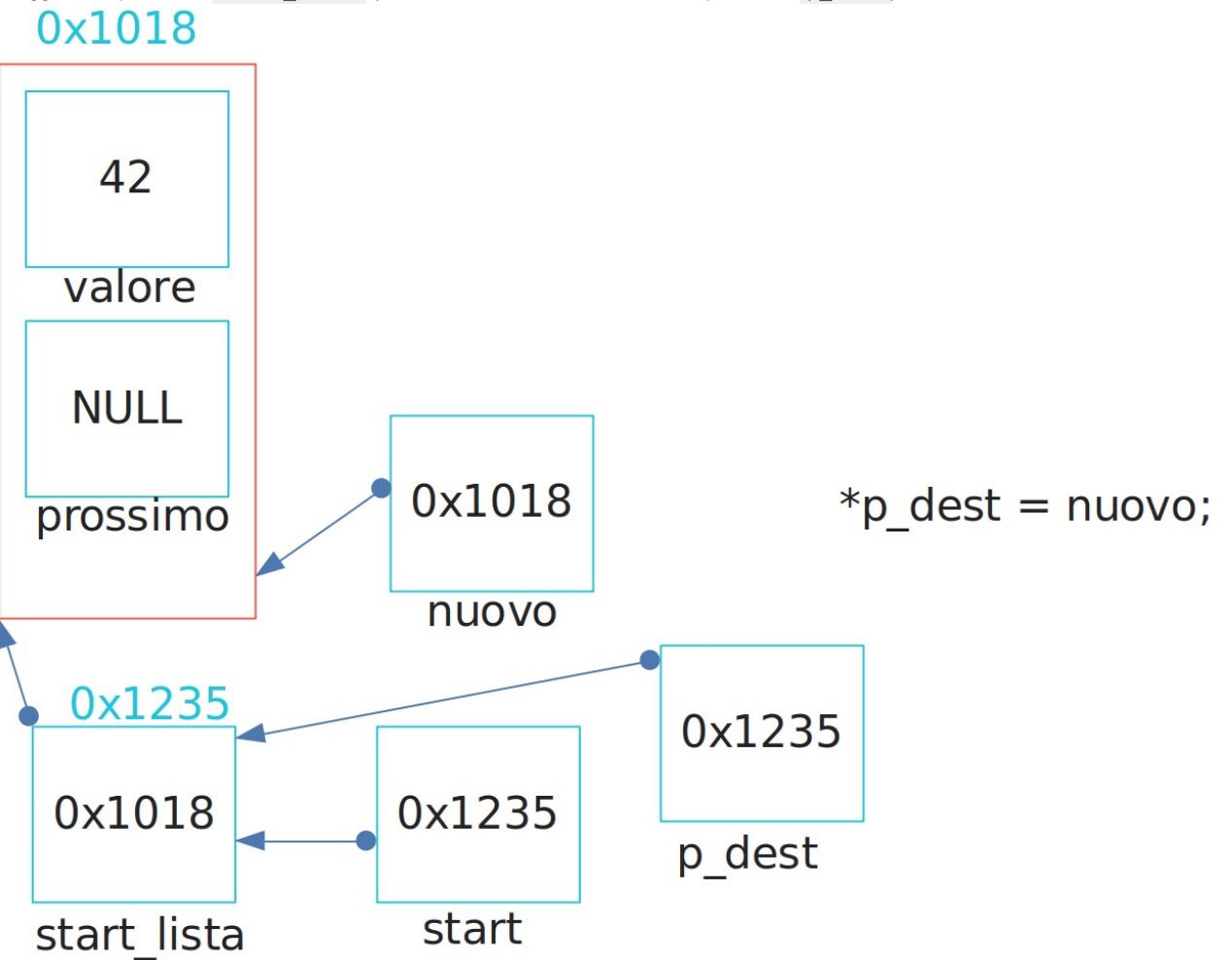
0x1235



- si scorre la lista fino alla fine usando `p_dest` come cursore. Dato che è vuota, `p_dest` continua a puntare al puntatore della testa

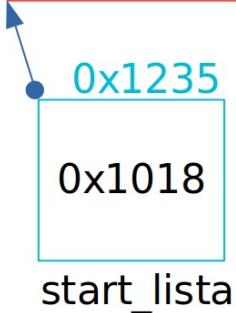
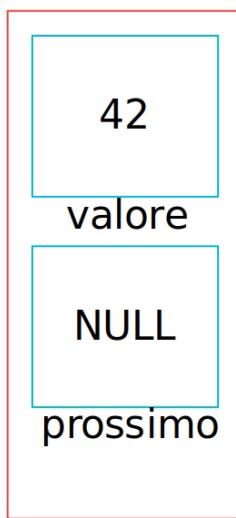


- si aggiorna il puntatore `start_list` (attraverso la dereferenziazione del puntatore `p_dest`)



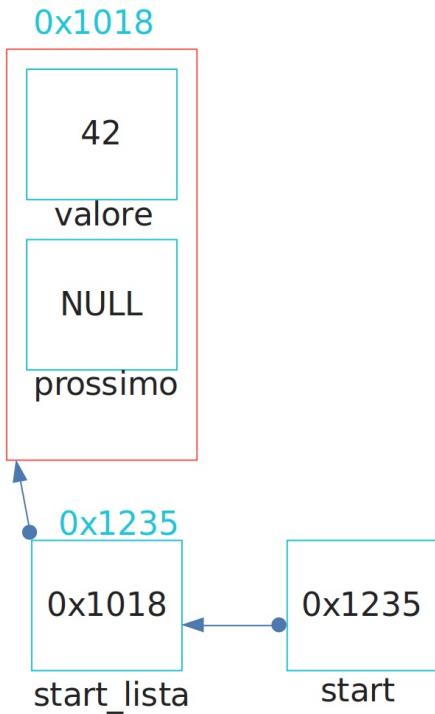
- situazione al termine della prima invocazione di `insert(...)`

0x1018



Seconda invocazione di `insert(...)`

- all'atto della seconda invocazione, la situazione è la seguente:



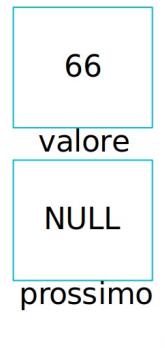
```
void insert(struct elemento** start, int val)
{
...
}
...
int main()
{
...
insert(&start_lista, val);
...
}
```

- viene allocato ed inizializzato il nuovo nodo

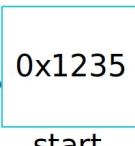
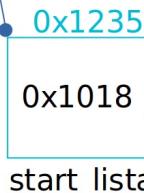
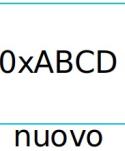
0x1018



0xABCD

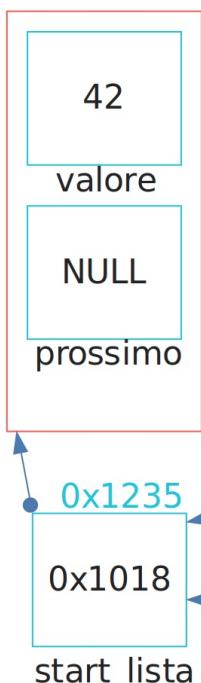


```
struct elemento* nuovo = malloc(sizeof(struct elemento));
nuovo->valore      = val;
nuovo->prossimo     = NULL;
```



- viene allocato un cursore `p_dest` di tipo **puntatore a puntatore** che punta al puntatore che dovrà contenere la testa della lista

0x1018



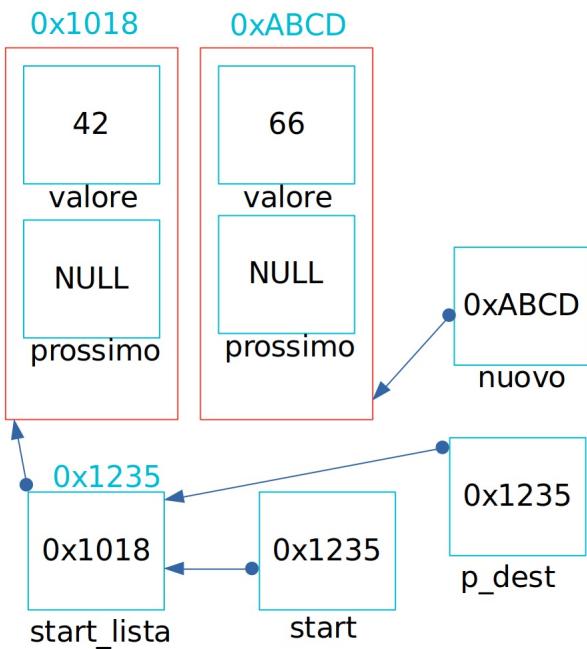
0xABCD



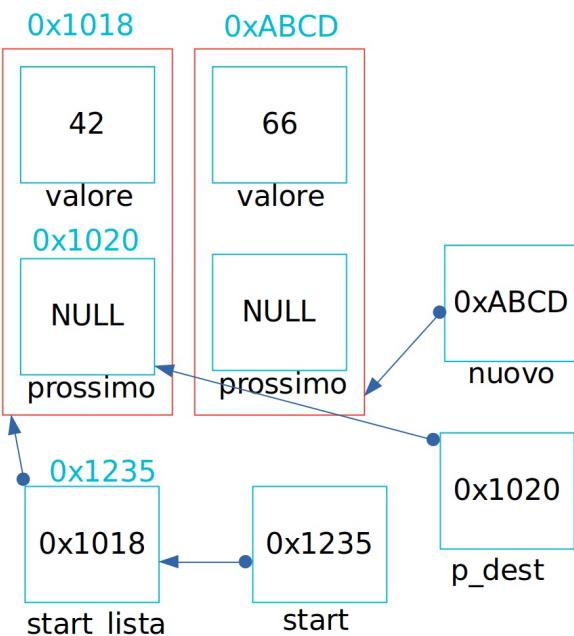
```
struct elemento** p_dest = start;
```

- si scorre la lista fino alla fine utilizzando il cursore `p_dest`.

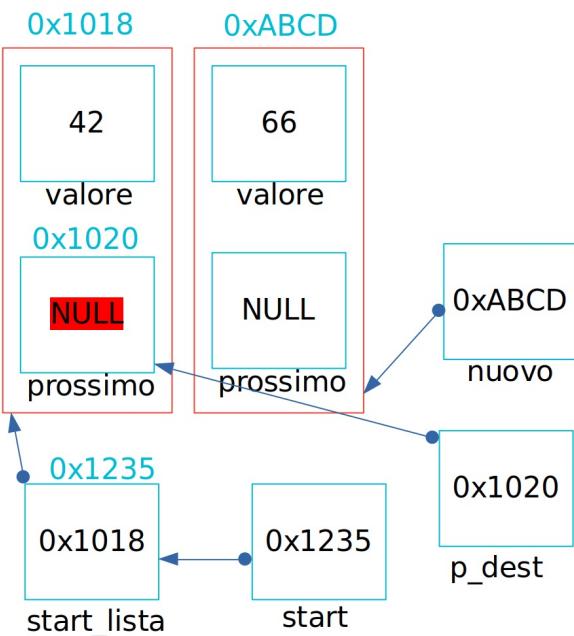
- durante la prima iterazione del `while(...)`



```
while(*p_dest!=NULL)
    p_dest = &((*p_dest)->prossimo);
```

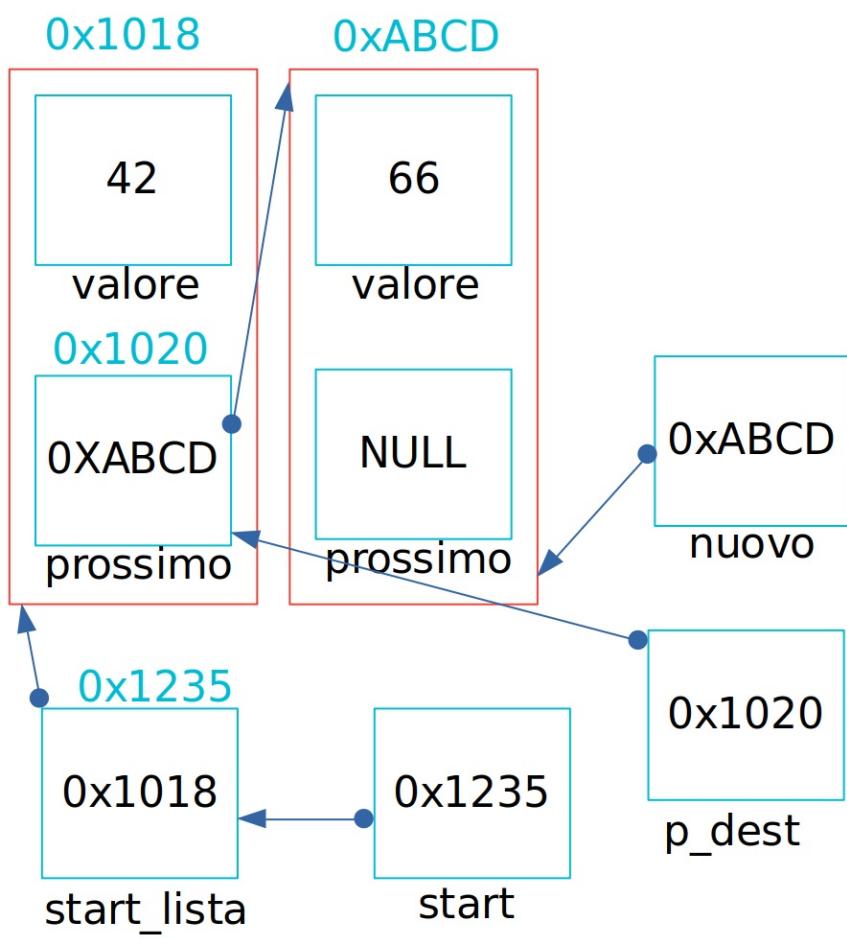


```
while( *p_dest!=NULL)
    p_dest = &((*p_dest)->prossimo);
```

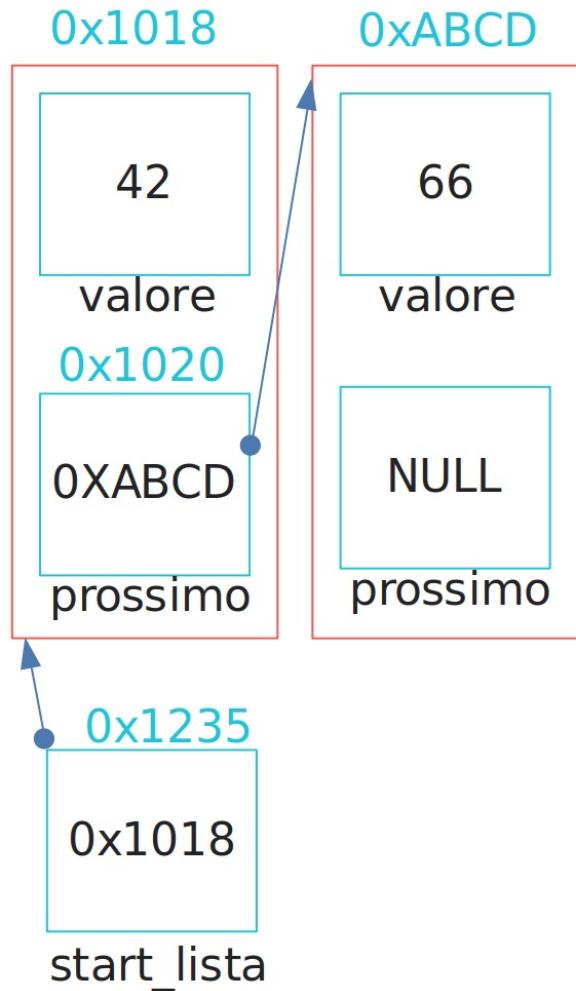


```
while( *p_dest!=NULL)
    p_dest = &((*p_dest)->prossimo);
```

- si aggiorna il puntatore `prossimo` dell'ultimo (ed unico) elemento in lista (attraverso la dereferenziazione del puntatore `p_dest`)

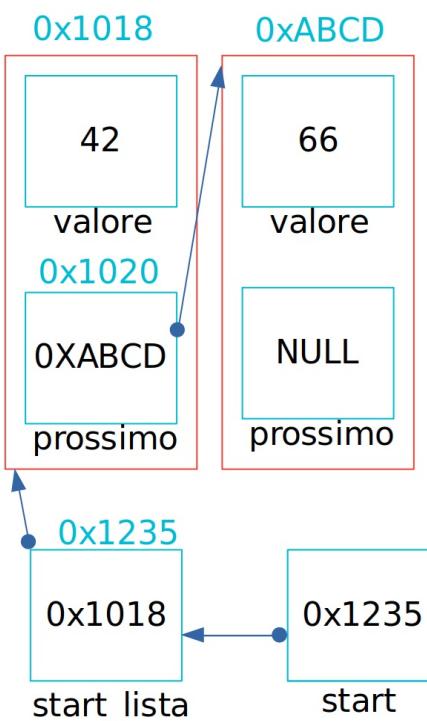


- situazione al termine della seconda invocazione di `insert(...)`



Terza invocazione di `insert(...)`

- all'atto della terza invocazione, la situazione è la seguente

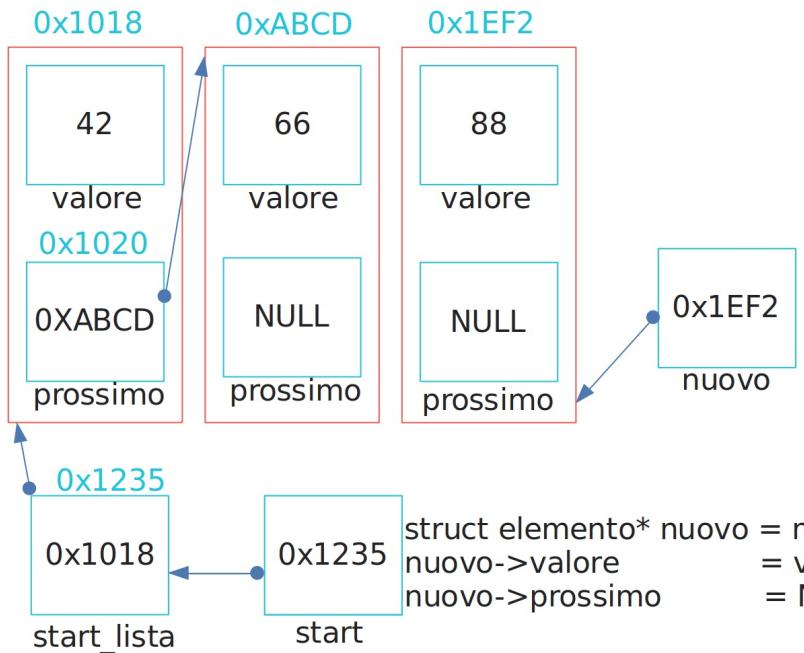


```

void insert(struct elemento** start, int val)
{
...
}
...
int main()
{
...
insert(&start_lista, val);
...
}

```

- viene allocato ed inizializzato il nuovo nodo

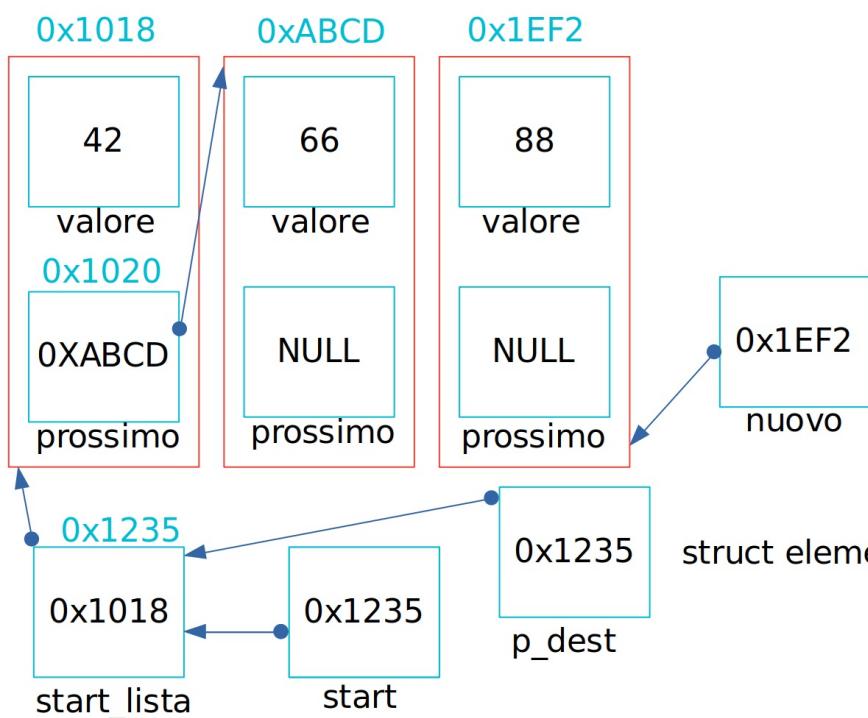


```

struct elemento* nuovo = malloc(sizeof(struct elemento));
nuovo->valore = val;
nuovo->prossimo = NULL;

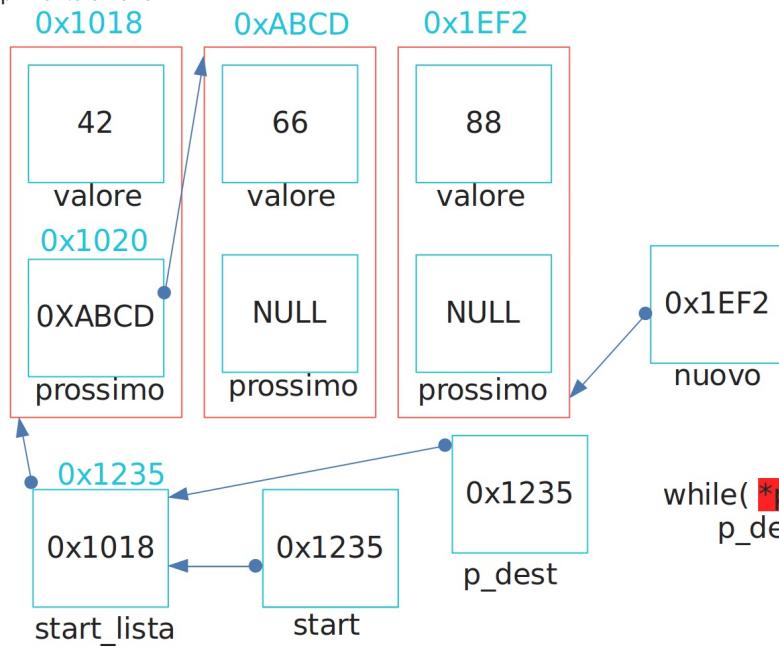
```

- □ viene allocato un cursore `p_dest` di tipo **puntatore a puntatore** che punta al puntatore che dovrà contenere la testa della lista



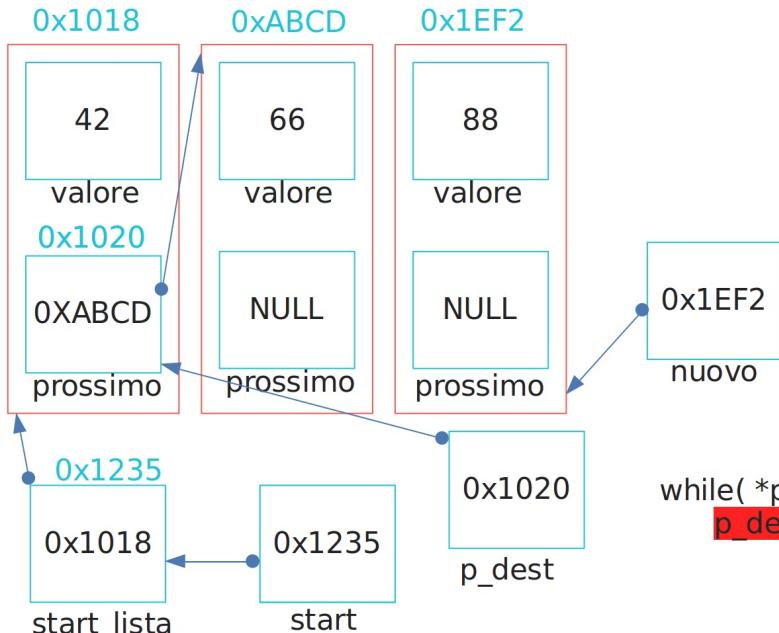
- si scorre la lista fino alla fine utilizzando il cursore `p_dest`.

- prima iterazione

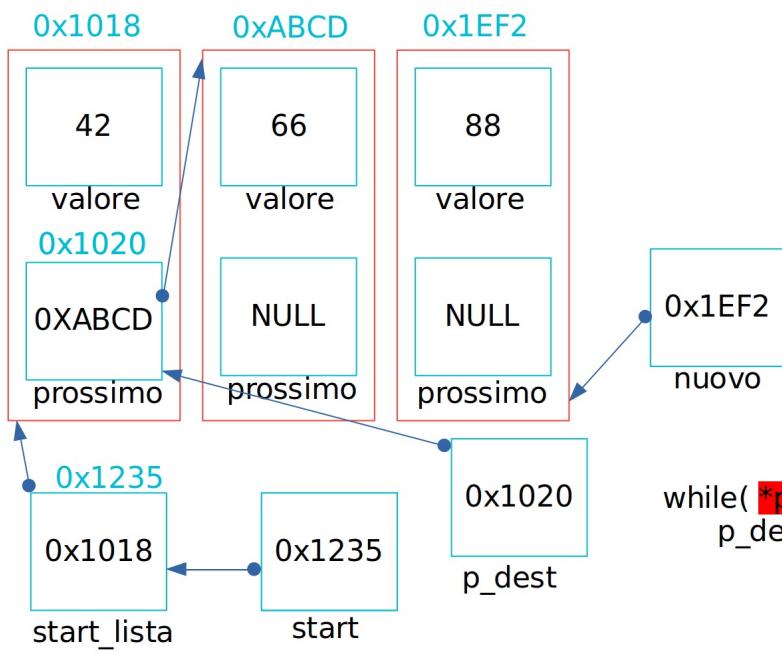


```
struct elemento** p_dest = start;
```

```
while( *p_dest!=NULL)
    p_dest = & ((*p_dest)->prossimo);
```

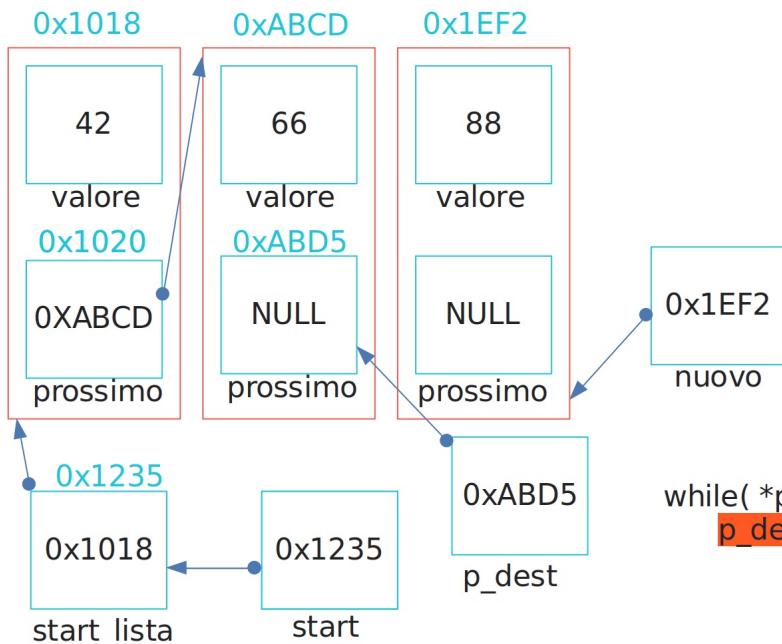


```
while( *p_dest!=NULL)
    p_dest = & ((*p_dest)->prossimo);
```

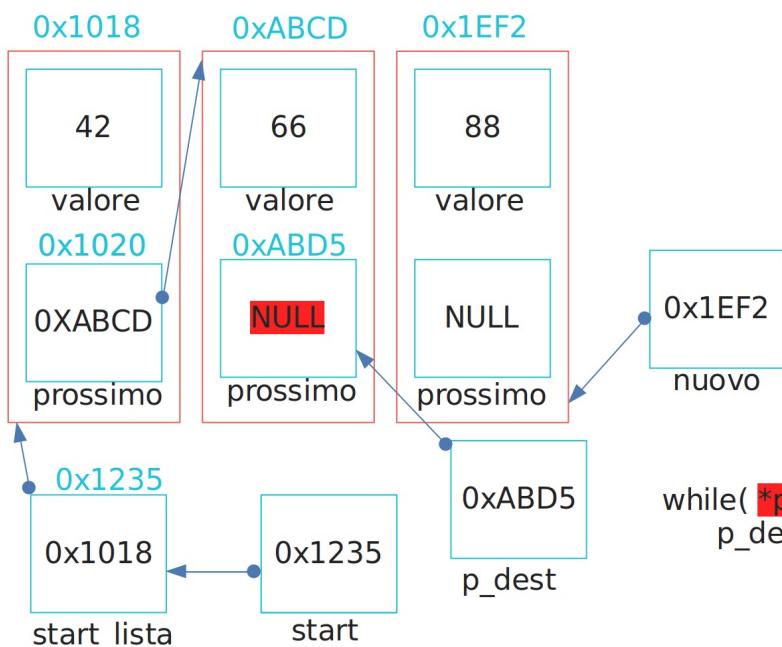


```
while( *p_dest!=NULL)
    p_dest = & ((*p_dest)->prossimo);
```

■ seconda iterazione

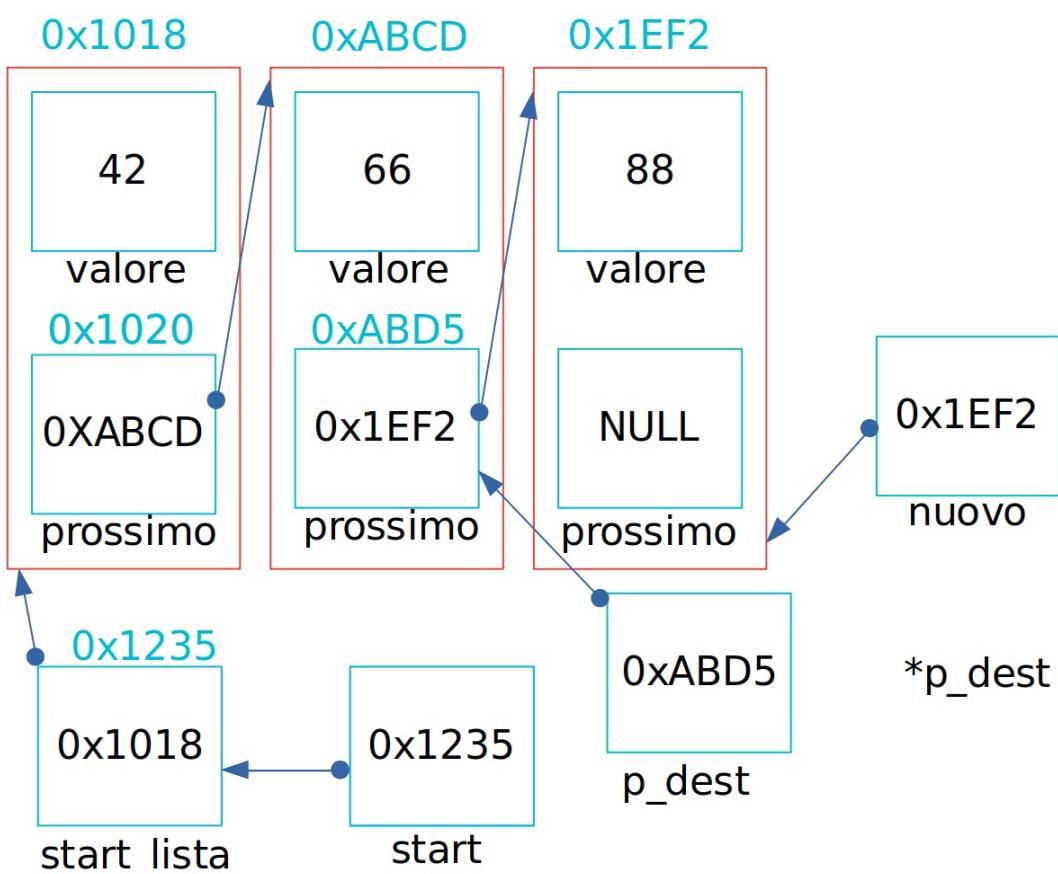


```
while( *p_dest!=NULL)
    p_dest = & ((*p_dest)->prossimo);
```

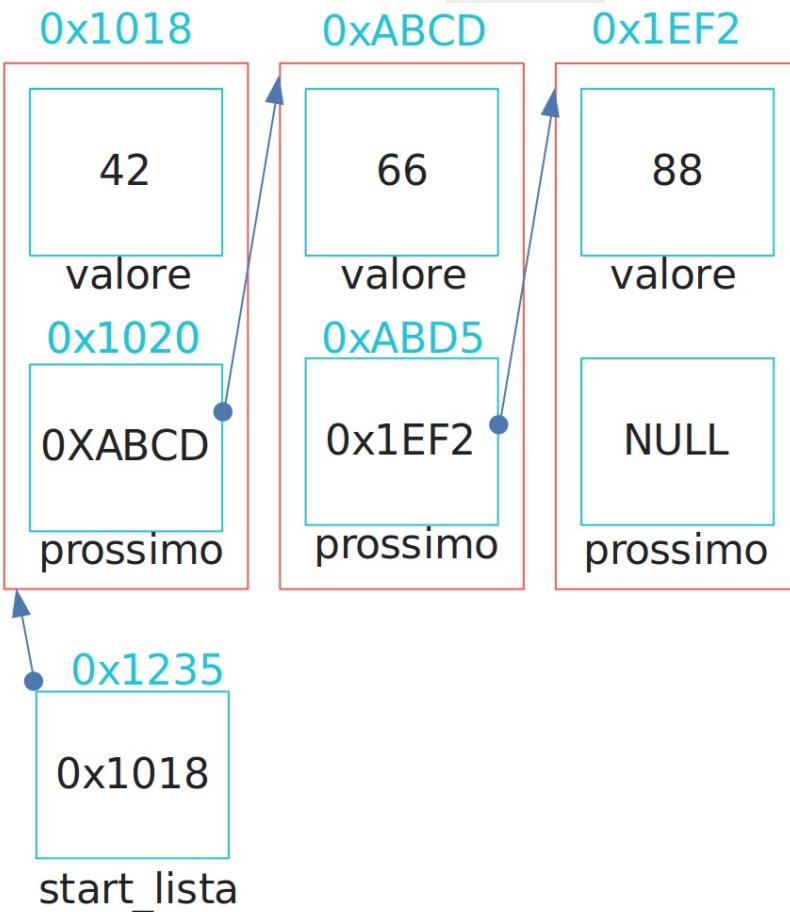


```
while( *p_dest!=NULL)
    p_dest = & ((*p_dest)->prossimo);
```

- si aggiorna il puntatore `prossimo` dell'ultimo elemento in lista (attraverso la dereferenziazione del puntatore `p_dest`)



- situazione al termine della terza invocazione di `insert(...)`



In [38]:

```
/*
Scrivere un programma che permetta all'utente di inserire degli interi
ed inserirli, uno dietro l'altro, in una lista concatenata.
Dare la possibilità all'utente di visualizzare tale lista.
*/
```

```
// Strategia 5: simile alla precedente, unica differenza è che risparmio un puntatore a puntatore, sfruttando
// quello che già tengo (dato che posso sovrascriverlo tranquillamente essendo un parametro formale)
```

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

```

struct elemento {
    int valore;
    struct elemento* prossimo;
};

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->{%d}", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}

void deallocate(struct elemento** start)
{
    while(*start != NULL)
    {
        struct elemento* prox = (*start)->prossimo;
        free(*start);
        *start = prox;
    }
    *start = NULL;
}

void insert(struct elemento** start, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // se la lista non è vuota,
    // scorrila fino a raggiungere l'ultimo elemento
    while( *start!= NULL)
        start = & ((*start)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo elemento (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo elemento della lista
    *start = nuovo;
}

int main()
{
    struct elemento* start_lista = NULL;

    int scelta = -1;
    while(scelta != 0)
    {
        printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
        scanf("%d", &scelta);

        int val;
        switch(scelta)
        {
            case 1:
                printf("valore da inserire: ");
                scanf("%d", &val);
                insert(&start_lista, val);
                break;
            case 2:
                print(start_lista);
                break;
            case 3:
                deallocate(&start_lista);
                break;
            case 0:
                deallocate(&start_lista);
                break;
            default:
                printf("scelta sbagliata! Ripetere\n");
        }
    }

    return 0;
}

```

- 1) inserisci
- 2) stampa
- 3) cancella tutto
- 0) esci

```
#  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{10}->{20}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{10}->{20}->{30}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{50}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{50}->{60}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

In [2]: /*
Scrivere un programma che permetta all'utente di inserire degli interi
ed inserirli, uno dietro l'altro, in una lista concatenata.
Dare la possibilità all'utente di visualizzare tale lista.
*/

```
// Strategia 6  
#include <stdio.h>  
  
struct elemento {  
    int valore;  
    struct elemento* prossimo;  
};
```

```

void print(struct elemento* start)
{
    printf("#");
    while(start!=NULL)
    {
        printf("->{%d}", start->valore);
        start = start->prossimo;
    }
    printf("\n");
}

void deallocate(struct elemento** start)
{
    while(*start != NULL)
    {
        struct elemento* prox = (*start)->prossimo;
        free(*start);
        *start = prox;
    }
    *start = NULL;
}

void insert(struct elemento** pp_head, int val)
{
    // 1. genero il nuovo elemento
    struct elemento* nuovo;
    nuovo = malloc(sizeof(struct elemento));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. controllo se la lista è vuota. In tal caso, l'elemento da inserire è il primo
    if( *pp_head == NULL )
    {
        *pp_head = nuovo;
        return;
    }

    // 3. scorro la lista fino a raggiungere l'ultimo elemento
    struct elemento* in_esame   = *pp_head;

    // alla fine del ciclo, in_esame punterà all'ultimo elemento della lista
    while(in_esame->prossimo!= NULL)
    {
        in_esame   = in_esame->prossimo;
    }
    // aggiorno
    in_esame->prossimo = nuovo;
}

int main()
{
    struct elemento* start_lista = NULL;

    int scelta = -1;
    while(scelta != 0)
    {
        printf("1) inserisci\n2) stampa\n3) cancella tutto\n0) esci\n");
        scanf("%d", &scelta);

        int val;
        switch(scelta)
        {
            case 1:
                printf("valore da inserire: ");
                scanf("%d", &val);
                insert(&start_lista, val);
                break;
            case 2:
                print(start_lista);
                break;
            case 3:
                deallocate(&start_lista);
                break;
            case 0:
                deallocate(&start_lista);
                break;
            default:
                printf("scelta sbagliata! Ripetere\n");
        }
    }

    return 0;
}

```

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{10}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{10}->{20}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{10}->{20}->{30}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

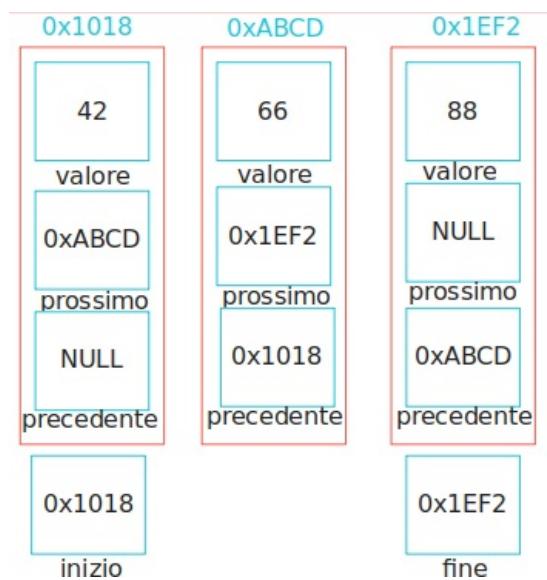
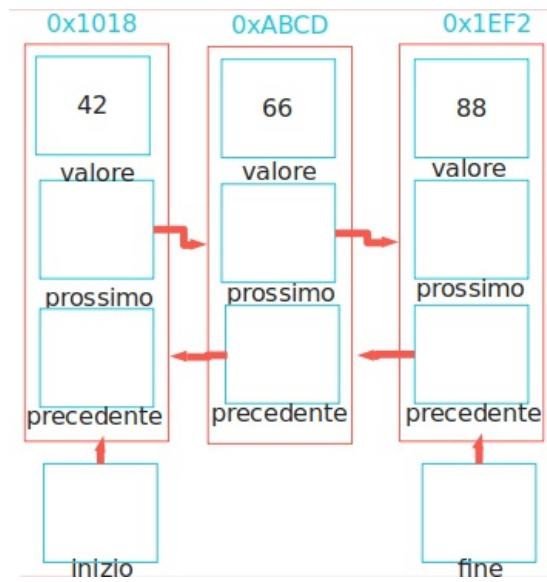
```
#->{50}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

valore da inserire:

```
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

```
#->{50}->{60}  
1) inserisci  
2) stampa  
3) cancella tutto  
0) esci
```

Liste doppiamente concatenate



Quando usare le LC

Appropriate quando:

- non è possibile determinare *a priori* il numero di elementi della struttura dati: la lunghezza di una LC può aumentare o diminuire in base alle necessità grazie all'assenza di un obbligo di contiguità degli elementi che la compongono
- si vogliono inserire/rimuovere elementi all'interno della struttura dati senza spostare tutti gli altri elementi: lavorando soltanto sui membri `successivo` (ed eventualmente `precedente`) è possibile inserire/rimuovere elementi in/da posizioni arbitrarie della lista senza rischiare di lasciare locazioni di memoria inutilizzate o di dover spostare tutta la struttura dati

Non appropriate quando:

- i tempi di accessi contano: a differenza di altre strutture dati come gli array, le liste necessitano di tempo per essere navigate, in quanto gli elementi che la compongono non è detto che siano memorizzati in maniera contigua. Ad esempio, per leggere il contenuto della 50-esima posizione di una lista semplicemente legata, dovrò per forza scorrere sequenzialmente i primi 49 elementi, a differenza di un vettore in cui l'accesso alla 50-esima posizione è diretto.

In []:

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

```
In [ ]: #include <stdio.h>
#define MAXLEN 16
int main()
{
    char str[MAXLEN];
    printf("inserisci stringa: ");
    fgets(str, MAXLEN, stdin);
    printf("Hai inserito la stringa:\\"%s\".\nCiao\n",str);

    return 0;
}
```

Esecuzione:

```
inserisci stringa: ciao pollo
Hai inserito la stringa:"ciao pollo
".
Ciao
```

da <https://pubs.opengroup.org/onlinepubs/009696699/functions/fgets.html> :

```
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

The `fgets()` function shall read bytes from stream into the array pointed to by `s`, until `n-1` bytes are read, or a `\n` is read **and transferred to `s`**, or an end-of-file condition is encountered. The string is then terminated with a null byte. Upon successful completion, `fgets()` shall return `s`. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and `fgets()` shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, `fgets()` shall return a null pointer,

```
In [ ]: int get_cleaned_line(char str[], int maxlen)
{
    int len = -1;
    if (fgets(str, maxlen, stdin) != NULL)
    {
        len = 0;
        while(str[len] != '\0')
            len++;
        if (len > 0 && str[len-1] == '\n')
        {
            str[len-1] = '\0';
            len--;
        }
    }
    return len; // dato che l'ho calcolata, può sempre servire...
}
```

```
In [ ]: int main()
{
    char str[MAXLEN];
    printf("inserisci stringa: ");
    int len = get_cleaned_line(str, MAXLEN);
    printf("Hai inserito la stringa:\\"%s\".\nCiao\n",str);
    printf("PS: tale stringa è lunga %d caratteri.\n", len);

    return 0;
}
```

```
inserisci stringa: ciao pollo
Hai inserito la stringa:"ciao pollo".
Ciao
PS: tale stringa è lunga 10 caratteri.
```

In []:



Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Puntatori a funzione

Puntatori in grado di contenere l'indirizzo (di partenza) di una funzione Esempio:

```
int main()
{
    float (*fn_ptr) (int, int);
}
```

Ho dichiarato un puntatore di nome `fn_ptr` che può puntare ad una qualsiasi funzione che accetta due parametri di tipo `int` e che restituisce un `float`. L'utilizzo delle parentesi è necessario per far capire al puntatore che si vuole un puntatore a funzione e non un puntatore a `float`.

Caso di studio:

Scrivere un programma che, attraverso due funzioni, stampi sia il minimo che il massimo di un array.

Possibile soluzione:

funzioni ausiliarie	main()
<pre>float min_array(float V[], int n) { float min = V[0]; for (int i=1; i < n) min = V[i]; return min; }</pre>	<pre>int main() { float V[] = {3,1,2,4,6,8,9,-5}; int n = 8; float min = min_array(V,n); float max = max_array(V,n); printf("min: %f\nmax: %f\n", min, max); }</pre>

Ho due funzioni dal corpo quasi identico, l'unica differenza sostanziale è nel confronto.

Posso scomporre ogni funzione in una coppia di sottoproblemi:

funzioni ausiliarie	main()
<pre>int confronto_less(float a, float b) { return a < b; } float max_array(int V[], int n) { float m = V[0]; for (int i=1; i < n) m = V[i]; return m; }</pre>	<pre>int main() { float V[] = {3,1,2,4,6,8,9,-5}; int n = 8; float min = confronto_less(0, max_array(V,n)); float max = max_array(V,n); printf("min: %f\nmax: %f\n", min, max); }</pre>

Le due funzioni `min_array(...)` e `max_array(...)` sono adesso quasi identiche. L'unica differenza risiede nel fatto che la prima invoca al suo interno la funzione `confronto_less(...)`, mentre la seconda invoca `confronto_greater(...)`. Potrei avere un'unica funzione invocante che, in base ad un parametro fornito in fase di invocazione, decide quale delle due funzioni invocare al suo interno (meccanismo delle *callbacks*)

In generale, una *callback* è una funzione che viene "passata" come parametro ad un'altra funzione

```
In [1]: #include <stdio.h>
int confronto_less(float a, float b)
{
    return a < b;
}

int confronto_greater(float a, float b)
{
```

```
    return a>b;
}

float find_in_array(float V[], int n, int (*confronto_fn)(float, float))
{
    float m = V[0];
    for (int i=1; i<n; i++)
        if ( confronto_fn(V[i], m) )
            m = V[i];
    return m;
}

int main()
{
    float V[] = {3,1,2,4,6,8,9,-5};
    int n      = 8;
    float min = find_in_array(V,n, confronto_less);
    float max = find_in_array(V,n, confronto_greater);

    printf("min: %f\nmax: %f\n", min, max);
}
```

```
min: -5.000000
max: 9.000000
```

In []:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Struct (Record)

una `struct` in C è una aggregato di dati i cui tipi non sono per forza omogenei

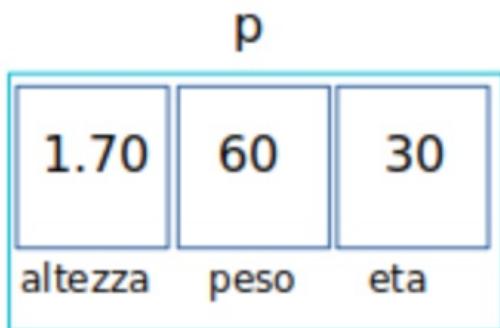
L'allocazione di una `struct` in memoria necessita di due fasi:

- (eventuale) *dichiarazione* della struttura (ossia del suo nome, non sempre necessaria)
- *definizione* della struttura: viene specificato *quali* elementi la struttura dovrà contenere. Da notare che in questa fase non viene effettuata alcuna allocazione, ma viene soltanto *descritto* quali *campi* comporranno le eventuali strutture di quel tipo
- allocazione *effettiva* dell'*object* (o degli *object*) di tipo struttura

```
// dichiarazione
struct Paziente;

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

int main()
{
    struct Paziente p; //allocazione
    p.eta      = 30;
    p.peso     = 60;
    p.altezza  = 1.70;
    return 0;
}
```



Array di struct

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

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

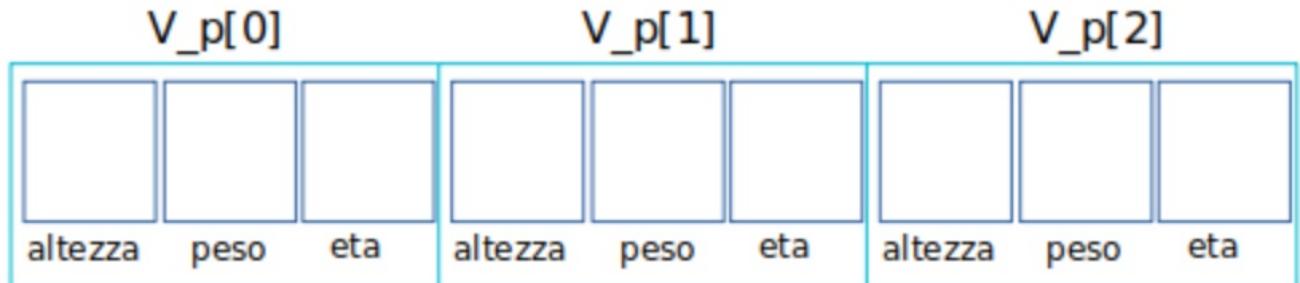
int main()
{
    struct Paziente V_p[N];
```

```

for(int i=0; i<N; i=i+1)
{
    printf("inserisci l'altezza del %d paziente: ", i+1);
    scanf("%d", &V_p[i].altezza);
    printf("inserisci il peso del %d paziente: ", i+1);
    scanf("%d", &V_p[i].peso);
    printf("inserisci l'età del %d paziente: ", i+1);
    scanf("%d", &V_p[i].eta)
}

return 0;
}

```



Precedenza degli operatori

Precedence	Operator	Description
	<code>++ --</code>	Suffix/postfix increment and decrement
	<code>()</code>	Function call
1	<code>[]</code>	Array subscripting
	<code>.</code>	Structure and union member access
	<code>-></code>	Structure and union member access through pointer
	<code>(type){list}</code>	Compound literal(C99)
	<code>++ --</code>	Prefix increment and decrement
	<code>+ -</code>	Unary plus and minus
	<code>! ~</code>	Logical NOT and bitwise NOT
2	<code>(type)</code>	Cast
	<code>*</code>	Indirection (dereference)
	<code>&</code>	Address-of
	<code>sizeof</code>	Size-of
	<code>_Alignof</code>	Alignment requirement(C11)
3	<code>* / %</code>	Multiplication, division, and remainder
4	<code>+ -</code>	Addition and subtraction
5	<code><< >></code>	Bitwise left shift and right shift
6	<code>< <=</code>	For relational operators < and \leq respectively
	<code>> >=</code>	For relational operators > and \geq respectively
7	<code>== !=</code>	For relational = and \neq respectively
8	<code>&</code>	Bitwise AND
9	<code>^</code>	Bitwise XOR (exclusive or)
10	<code> </code>	Bitwise OR (inclusive or)
11	<code>&&</code>	Logical AND
12	<code> </code>	Logical OR
13	<code>? :</code>	Ternary conditional
	<code>=</code>	Simple assignment
	<code>+= -=</code>	Assignment by sum and difference
14	<code>*= /= %=</code>	Assignment by product, quotient, and remainder
	<code><<= >>=</code>	Assignment by bitwise left shift and right shift
	<code>&= ^= =</code>	Assignment by bitwise AND, XOR, and OR
15	<code>,</code>	Comma

```

In [1]: # include <stdio.h>
#define MAX_PAZIENTI 3

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

void acquisisci_pazienti(struct Paziente vett[], int n)
{
    for(int i=0; i<n; i=i+1)
    {
        printf("inserisci l'altezza del %d paziente: ", i+1);
        scanf("%f", &vett[i].altezza);
        printf("inserisci il peso del %d paziente: ", i+1);
        scanf("%f", &vett[i].peso);
        printf("inserisci l'età del %d paziente: ", i+1);
        scanf("%d", &vett[i].eta);
    }
}

void stampa_pazienti(struct Paziente vett[], int n)
{
    for(int i=0; i<n; i=i+1)
    {
        printf("PAZIENTE: %d\n", i+1);
        printf("- altezza: %f\n", vett[i].altezza);
        printf("- peso: %f\n", vett[i].peso);
        printf("- eta: %d\n", vett[i].eta);
    }
}

int main()
{
    struct Paziente V_p[MAX_PAZIENTI];

    printf("dimensione del diario dei pazienti: %d Byte\n", sizeof(V_p));
    printf("dimensione di ogni struttura: %d\n", sizeof(struct Paziente));

    int n_pazienti;
    printf("quanti pazienti hai in studio?");
    scanf("%d", &n_pazienti);
    acquisisci_pazienti(V_p, n_pazienti);
    stampa_pazienti(V_p, n_pazienti);

    return 0;
}

```

dimensione del diario dei pazienti: 36 Byte
dimensione di ogni struttura: 12
quanti pazienti hai in studio?
inserisci l'altezza del 1 paziente:
inserisci il peso del 1 paziente:
inserisci l'età del 1 paziente:
inserisci l'altezza del 2 paziente:
inserisci il peso del 2 paziente:
inserisci l'età del 2 paziente:
PAZIENTE: 1
- altezza: 1.700000
- peso: 70.000000
- eta: 30
PAZIENTE: 2
- altezza: 1.680000
- peso: 70.000000
- eta: 34

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Layout della memoria di un programma C

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 `external`, `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 comando linux `size` fornisce informazioni sull'occupazione di memoria di un programma:

`esempio.c`:

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

```
int main() {
    return 0;
}
```

```
gcc esempio.c -o esempio
```

```
size esempio
```

output:

text	data	bss	dec	hex	filename
1418	544	8	1970	7b2	esempio

La dimensione massima dell' *execution stack* può essere recuperata con il comando `ulimit -s` (l'output è espresso in KB).

```
ulimit -s
```

output:

```
8192
```

In questo caso, lo spazio a disposizione nell' *execution stack* è 8192 KB.

Tipi di allocazione in C

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 variabili 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.

Il concetto di **object**

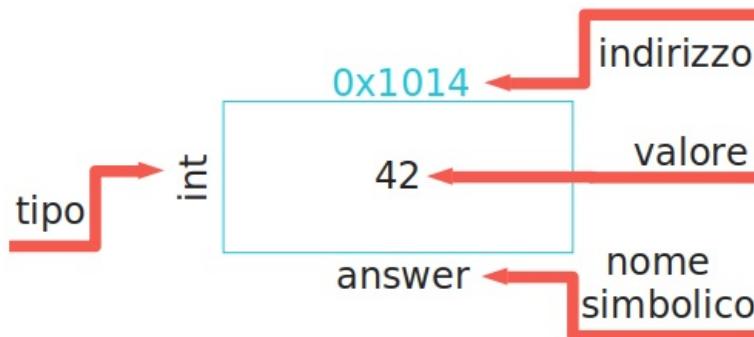
object: region of data storage in the execution environment, the contents of which can represent values

(From: C committee draft www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

NESSUNA relazione con la *Programmazione ad Oggetti*

Quindi in C, una regione di memoria contenente dati è chiamata genericamente **object** (variabili, array, etc.)

```
int answer = 42;
```



Dichiarazione Vs Definizione (object)

Dichiarazione: dichiara il nome ed il tipo di un object (in sostanza dice che l'object esiste da qualche parte)

Definizione: alloca lo spazio necessario per l'object (in sostanza viene materialmente creato in memoria)

In molti casi, le due fasi corrispondono. Esempio: `int a;` dichiara che esiste un object di nome `a` di tipo `int`, e contemporaneamente ne alloca lo spazio in memoria.

In generale, in uno stesso programma, un object può avere *più* dichiarazioni *ma una sola* definizione.

Esempio di un object *dichiarato* ma non *definito*:

```
extern int g;
```

`extern` dichiara l'esistenza dell'object `g`, *definito* (adesso o in futuro) da qualche altra parte. Il compilatore quindi, nel caso in cui sia richiesto l'utilizzo di `g` prima dell'effettiva *definizione*, non se ne preoccupa e continua il processo di compilazione (in quanto semplice traduzione).

Esempio:

```
In [11]: #include <stdio.h>
int main(void)
{
    extern int first, last; /* use global vars */
    printf("i valori di first e last sono: %d %d\n", first, last);
    return 0;
}
/* global definition of first and last */
int first = 10, last = 20;
i valori di first e last sono: 10 20
dato che si richiede l'utilizzo di first, last prima della loro effettiva definizione, vanno almeno dichiarate in precedenza.
```

Inizializzazione Vs Assegnazione

Assegnazione: dare valore ad un object in qualsiasi punto del programma

Inizializzazione: dare valore ad un object *in fase di definizione*

```
{
    int a; // dichiarazione & definizione
    a = 3; // assegnazione
}
{
    int a = 3; // dichiarazione & definizione & inizializzazione
}
```

In C, alcune cose lecite in fase di inizializzazione non lo sono in fase di assegnazione.

Ad ogni modo, un object è detto **inizializzato** se gli è stato dato un valore dal programma, indipendentemente dal modo in cui ciò sia avvenuto.

Esempio:

```
int V[] = {1,2,3}; // si può fare
int Q[];
Q[] = {4,5,6}; // non si può fare
const int i = 5; // si può fare
const int i;
i = 5; // non si può fare
```

Strutture dati native del C

Una struttura dati è un insieme di dati di tipi base aggregati assieme secondo uno schema.

Il C mette a disposizione in maniera nativa le seguenti strutture dati:

- **array:**
 - monodimensionali
 - multidimensionali
- **struct:**
 - semplici
 - ricorsive

Array

Un array è una sequenza di elementi di tipo omogeneo

Array **Monodimensionali**:

```
int V[12];
```

Array **Multidimensionali**:

```
int K[3][4];      // array di 12 elementi distribuiti lungo 2 dimensioni
int W[2][3][2];   // array di 12 elementi distribuiti lungo 3 dimensioni
int Q[1][2][2][3]; // array di 12 elementi distribuiti lungo 4 dimensioni
```

Gli *indici* di un array vengono utilizzate per identificare un elemento in fase di lettura/scrittura, esattamente come in un sistema di coordinate.

```
K[2][0]      = 5; // assegno il valore 5 alla posizione di indici (2,0)
W[2][0][0]    = 5; // assegno il valore 5 alla posizione di indici (2,0,0)
Q[0][2][0][0] = 5; // assegno il valore 5 alla posizione di indici (0,2,0,0)
```

Array monodimensionali

```
int V[12];
V[4] = 2;
```

Da notare che l'operatore `[]` assume due significati differenti in base a dove è usato:

- in fase di **dichiarazione/definizione**: specifica **quanti** elementi compongono l'array
- in fase di **accesso**: specifica **dove** leggere/scrivere il dato. In tale fase è detto anche *subscript operator*.

il valore tra `[]` deve essere di tipo intero, e può essere sia un valore costante che una variabile. Esempio:

```
int a = 4;
v[a] = 42; // inserisce nella posizione di indice 4 il valore 42
```

In fase di accesso i valori plausibili vanno da `0` a `n-1` dove `n` è il numero di elementi che compongono l'array esplicitato in fase di dichiarazione.

Matrici

Un array a due dimensioni viene chiamato anche *matrice*. In quanto tale, può venire facilmente rappresentata in forma tabellare in termini di righe (associate alla prima dimensione) e colonne (associate alla seconda dimensione)

```
cint K[3][4] = {{3,2,1,1},{7,9,7,7},{5,4,2,4}};
```

corrisponde logicamente a

$$K = \begin{pmatrix} 3 & 2 & 1 & 1 \\ 7 & 9 & 7 & 7 \\ 5 & 4 & 2 & 4 \end{pmatrix}$$

L'istruzione

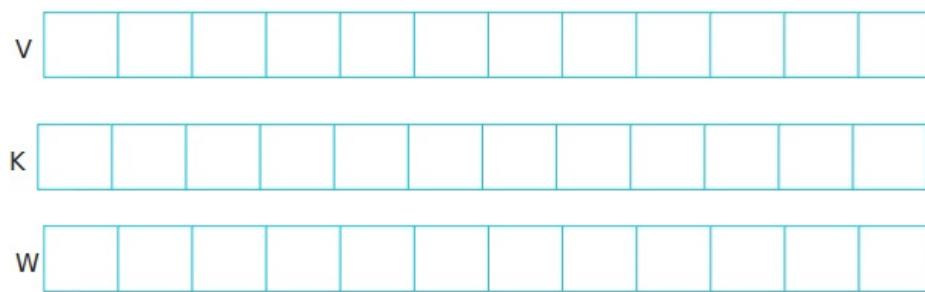
`K[1][2] = 0;`

corrisponde a modificare la matrice nel seguente modo:

$$K = \begin{pmatrix} 3 & 2 & 1 & 1 \\ 7 & 0 & 7 & 7 \\ 5 & 4 & 2 & 4 \end{pmatrix}$$

Allocazione di un array

Essendo le locazioni di memoria disposte fisicamente in maniera "sequenziale", l'allocazione di un array multidimensionale in memoria non è, ovviamente, tabellare/multidimensionale, ma sempre monodimensionale, ossia:



Come si vede, a parità di numero di elementi, lo spazio è allocato esattamente nello stesso modo, indipendentemente dal numero di assi/dimensioni utilizzato. Ciò che cambia è l'associazione tra gli indici e gli elementi effettivi.

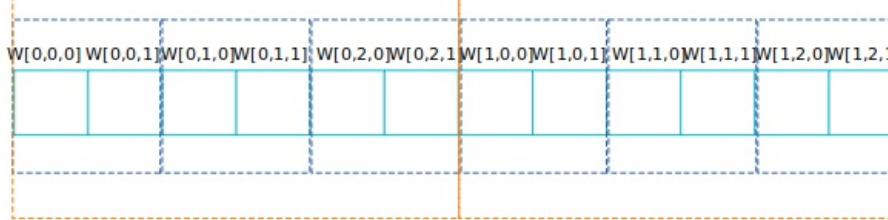
`V[0] V[1] V[2] V[3] V[4] V[5] V[6] V[7] V[8] V[9] V[10] V[11]`



`K[0,0] K[0,1] K[0,2] K[0,3] K[1,0] K[1,1] K[1,2] K[1,3] K[2,0] K[2,1] K[2,2] K[2,3]`



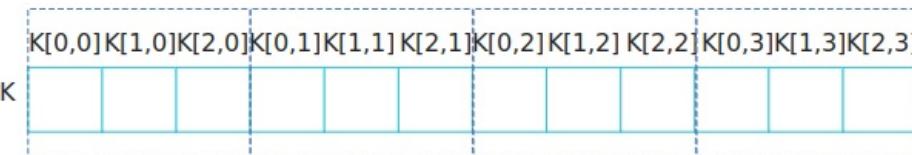
`W[0,0,0] W[0,0,1] W[0,1,0] W[0,1,1] W[0,2,0] W[0,2,1] W[1,0,0] W[1,0,1] W[1,1,0] W[1,1,1] W[1,2,0] W[1,2,1]`



La strategia di associazione indici-elementi di un array multidimensionale utilizzati in C è detta *row-major order*.

- Se un linguaggio utilizza il *row-major order*, allora gli elementi in memoria saranno disposti in modo tale che, leggendo gli elementi in sequenza, la "velocità" di cambiamento degli indici aumenti dall'ultima dimensione alla prima dimensione.
- La strategia di associazione dipende dal linguaggio. E.g., FORTRAN, MATLAB utilizzano il *column-major order*, in cui, leggendo gli elementi in sequenza, la "velocità" di cambiamento degli indici aumenta dalla prima dimensione all'ultima dimensione.

Esempio di *column-major order*:



Array sovradimensionati

La dimensione dei vettori automatici viene decisa nel codice sorgente, e quindi fissata a *compile time*. Spesso però si vorrebbe che la dimensione effettiva di tali vettori possa essere impostata durante la materiale esecuzione del programma, ossia *run-time*. Esempi:

- dato un vettore di 10 elementi, costruire un secondo vettore contenente soltanto gli elementi pari di quest'ultimo (non si può determinare la lunghezza del secondo vettore se non dopo una scansione del primo vettore a run time del primo)
- acquisire dall'utente un vettore di n interi con n deciso dall'utente (Impossibile sapere la lunghezza del vettore prima del run time in cui l'utente comunica quanti elementi inserire)

In questi casi, le possibilità sono almeno due:

- utilizzo di *allocazione dinamica della memoria* (vedremo poi)
- utilizzare una *stima* della lunghezza

Ad esempio:

- se si vuole produrre un "sottovettore", la lunghezza sarà *al più* pari alla lunghezza del vettore di partenza
- se la dimensione è decisa dall'utente, si può fare una stima di quanti elementi un utente potrebbe inserire (e.g., non più di 100), avvisandolo opportunamente di tale limite.

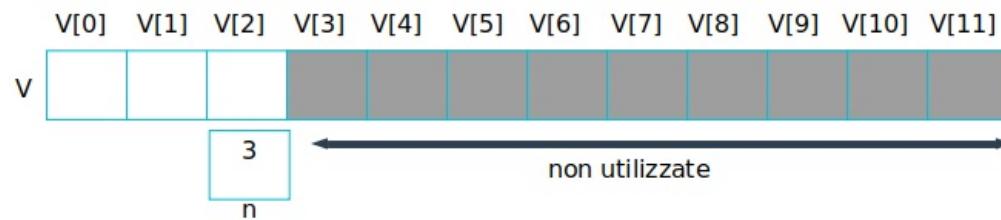
Entrambe queste soluzioni fanno uso di vettori sovradimensionati. Sarà necessaria quindi una ulteriore variabile che contenga quanti siano gli elementi realmente utilizzati.

Esempio:

```
int main()
{
    int V[12]; // dichiaro un vettore di 12 elementi
    int n;      // variabile di supporto contenente il numero di elementi effettivi
    printf("quanti elementi vuoi inserire nel vettore? (NB: MAX 12)");
    scanf("%d",&n);

    for(int i=0; i < n; i++) // l'accesso al vettore avviene soltanto sui primi n elementi
    {
        scanf("%d",&V[i]);
    }

    return 0;
}
```



In alcuni testi, la variabile contenente il numero di elementi effettivi del vettore è detta *riempimento*. Sconsiglio di usare questo termine.

Lo stesso discorso si può generalizzare al caso degli array multidimensionali. Ad esempio, nel caso delle matrici, si può chiedere all'utente di fornire il numero di righe ed il numero di colonne materialmente utilizzate, a fronte di una matrice allocata sovradimensionata.

Potreste aver sentito parlare degli array di lunghezza variabile (*Variable Length Array*, VLA). Non saranno trattati in questo corso.

(Possibili) soluzioni ad (alcuni) esercizi

In [17]:

```
/*
Dato un vettore ed un valore k,
ricerca di k all'interno di un vettore
(versione NON ottimizzata, usare un for (o while) che scorre il vettore fino alla fine)
*/
#include <stdio.h>

int main()
{
    int vett[] = {1,4,8,2};
    int n      = 4;
    int k      = 8;
    int trovato= 0;
    for(int i = 0; i < n; i++)
    {
        if (k == vett[i])
        {

```

```

        trovato = 1;
    }

    if (trovato == 1)
        printf("il valore %d è contenuto nel vettore\n", k);
    else
        printf("il valore %d non è contenuto nel vettore\n", k);
    return 0;
}

```

il valore 8 è contenuto nel vettore

In [20]:

```

/*
Dato un vettore ed un valore k, ricerca di k all'interno di un vettore
(il ciclo si deve fermare quando l'elemento è stato trovato,
USARE UN WHILE NON UN FOR)
*/
#include <stdio.h>

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

int main()
{
    int vett[] = {1,4,8,2};
    int n      = 4;
    int k      = 5;
    int trovato = 0;
    int i      = 0;
    while(i < n && trovato == 0)
    {
        if (k == vett[i])
        {
            trovato = 1;
        }
        i++;
    }
    printf("vettore: ");
    stampa_vett(vett, n);
    printf("cerco il valore %d...\n", k);
    if (trovato == 1)
        printf("il valore %d è contenuto nel vettore\n", k);
    else
        printf("il valore %d non è contenuto nel vettore\n", k);
    return 0;
}

```

vettore: (1, 4, 8, 2)
cerco il valore 5...
il valore 5 non è contenuto nel vettore

In [22]:

```

// versione alternativa
#include <stdio.h>

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

int cerca(int v[], int dim, int key)
{
    int i = 0;
    while(i < dim)
    {
        if (key == v[i])
        {
            break;
        }
        i++;
    }

    if (i < dim)
        return 1;
    else
        return 0;
}

int main()
{
    int vett[] = {1,4,8,2};

```

```

int n      = 4;
int k      = 8;

printf("vettore: ");
stampa_vett(vett, n);

printf("cerco il valore %d...\n", k);

int trovato = cerca(vett, n, k);
if (trovato == 1)
    printf("il valore %d è contenuto nel vettore\n", k);
else
    printf("il valore %d non è contenuto nel vettore\n", k);
return 0;
return 0;
}

```

```

vettore: (1, 4, 8, 2)
cerco il valore 8...
il valore 8 è contenuto nel vettore

```

In [23]:

```

/*
Dato un array ed valore k, contare quante volte appare k nell' array
*/
#include <stdio.h>
```

```

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

int conta(int v[], int dim, int key)
{
    int c = 0;
    int i = 0;
    while(i < dim)
    {
        if (key == v[i])
        {
            c++;
        }
        i++;
    }

    return c;
}
int main()
{
    int vett[] = {1,4,8,2,4};
    int n      = 5;
    int k      = 4;

    printf("vettore: ");
    stampa_vett(vett, n);

    printf("conto quante volte appare il valore %d nel vettore...\n", k);
    int freq = conta(vett, n, k);

    if (freq > 0)
        printf("il valore %d è contenuto nel vettore %d volte\n", k, freq);
    else
        printf("il valore %d non è contenuto nel vettore\n", k);

    return 0;
}
```

```

vettore: (1, 4, 8, 2, 4)
conto quante volte appare il valore 4 nel vettore...
il valore 4 è contenuto nel vettore 2 volte

```

In [29]:

```

/*
determinare il minimo (massimo) presente in un vettore e stamparne a video
sia valore che la posizione in cui si trova.
Esempio: dato `A = 1,3,5,-2,4,0,6` allora stampa `il minimo è -2 e la posizione in cui si trova è 3`
*/

```

```

#include <stdio.h>

void stampa_vett(float vett[], int n)
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%.2f, ", vett[i]);
    printf("\b\b)\n");
}
```

```

int idx_min_array(float v[], int n)
{
    /*
    dato un array di lunghezza n, viene restituito l'indice del valore più piccolo
    */
    int idx = 0;
    for (int i=1; i<n; i++)
        if (v[i] < v[idx])
            idx = i;
    return idx;
}

int idx_max_array(float v[], int n)
{
    /*
    dato un array di lunghezza n, viene restituito l'indice del valore più grande
    */
    int idx = 0;
    for (int i=1; i<n; i++)
        if (v[i] > v[idx])
            idx = i;
    return idx;
}

int main()
{
    float v[] = {3,1,2,4,6,8,9,-5};
    int n     = 8;
    int idx_min = idx_min_array(v,n);
    int idx_max = idx_max_array(v,n);
    printf("vettore: ");
    stampa_vett(v, n);
    printf("min: %.2f    max: %.2f\n", v[idx_min], v[idx_max]);
    printf("idx: [%d]      [%d]\n", idx_min, idx_max);
}

```

```

vettore: (3.00, 1.00, 2.00, 4.00, 6.00, 8.00, 9.00, -5.00)
min: -5.00    max: 9.00
idx: [ 7]      [ 6]

```

```

In [1]: /* Dato un vettore ed un indice di posizione idx,
eliminazione dell'elemento in posizione idx tramite shift a sinistra
*/
/* Dato un vettore ed un indice di posizione idx,
eliminazione dell'elemento in posizione idx tramite shift a destra
*/
#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 shift_sx(int vett[], int* n, int pos)
{
    for(int i = pos; i < (*n)-1; i++)
        vett[i] = vett[i+1];
    (*n)--;
}

void shift_dx(int vett[], int* n, int pos, int ins)
{
    /*
    NB: ovviamente il vettore vett mi aspetto che sia sovrdimensionato di
    almeno una posizione
    */
    for(int i = *n; i > pos; i--) /* parto dalla prima posizione libera
                                    trascinandomi gli elementi all'indietro*/
        vett[i] = vett[i-1];
    vett[pos] = ins;
    (*n)++;
}

int main()
{
    int v[] = {1, 3, 5, 2, 4, 8};
    int n   = 6;
    int k   = 3;
    printf("prima di shift a sx dalla posizione %d:\n", k );
    stampa_vett(v, n);
    shift_sx(v, &n, k);
    printf("dopo lo shift a sx dalla posizione %d:\n", k );
}

```

```

    stampa_vett(v, n);

    int ins=50;
    shift_dx(v, &n, k, ins);
    printf("dopo lo shift a dx dalla posizione %d e inserendo il valore %d:\n",k,ins );
    stampa_vett(v, n);

    return 0;
}

prima di shift a sx dalla posizione 3:
(1,3,5,2,4,8)
dopo lo shift a sx dalla posizione 3:
(1,3,5,4,8)
dopo lo shift a dx dalla posizione 3 e inserendo il valore 50:
(1,3,5,50,4,8)

```

```

In [36]: // Dato un vettore, eliminarne i duplicati tramite shift
#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 shift_sx(int vett[],int* n,  int pos)
{
    for(int i = pos; i < *n; i++)
        vett[i] = vett[i+1];
    (*n)--;
}

int is_presente(int vett[], int n, int val)
{
    int trovato = 0;
    int i = 0;
    while(trovato == 0 && i < n)
    {
        if( vett[i] == val)
            trovato = 1;
        i++;
    }

    return trovato;
}

void elimina_doppioni(int vett[], int* n)
{
    int i = 0;
    while(i < *n)
    {
        int pres = is_presente( vett, i, vett[i] ); // controllo se vett[i] è presente
                                                       // nelle i posizioni precedenti
        if (pres == 1)                            // se è presente
            shift_sx(vett, n, i);                // lo elimino
        else
            i++; /* incremento i solo nel caso in cui non sia stato
                     eliminato il valore i-esimo.
                     Questo perchè, nel caso in cui il valore in posizione di indice i
                     sia stato eliminato,
                     grazie allo spostamento causato dallo shift
                     la posizione di indice i contiene adesso
                     il prossimo valore da esaminare
            */
    }
}

#define MAX_LEN 100
int main()
{
    int A[] = {1,2,3,2,3,8,8,5,5,2,8,6};
    int n_A = 12;
    printf("vettore iniziale: ");
    stampa_vett(A, n_A);

    elimina_doppioni(A, &n_A);

    printf("vettore ripulito: ");
    stampa_vett(A, n_A);
}

```

```
vettore iniziale: (1,2,3,2,3,8,8,5,5,2,8,6)
vettore ripulito: (1,2,3,8,5,6)
```

```
In [30]: /*
Dato un vettore, stamparne un diagramma a barre fatto di `*` .
esempio: vettore: [ 3 6 4] la stampa sarà:
 ***
*****
****

*/
#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 stampa_bar_diagramm(int vett[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for(int k = 0; k < vett[i]; k++)
            printf("*");
        printf("\n", vett[i]);
    }
}

int main()
{
    int v[] = {1, 3, 5, 2, 4, 8};
    int n    = 6;
    printf("vettore:");
    stampa_vett(v, n);
    printf("diagramma a barre:\n");
    stampa_bar_diagramm(v, n);
    return 0;
}
```

```
vettore:(1,3,5,2,4,8)
```

```
diagramma a barre:
```

```
*
```

```
***
```

```
*****
```

```
**
```

```
****
```

```
*****
```

```
In [32]: /*
dati due vettori `X` ed `Y` contenenti le coordinate di una serie di punti,
calcolare una matrice `D` contenente le distanze tra questi punti
*/
#include <stdio.h>
#include <math.h>
#define MAX_COLS 10
#define MAX_ROWS 10
void stampa_vett(float vett[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
        printf("%.2f, ", vett[i]);
    printf("\b\b)\n");
}

void stampa_mat(float mat[][MAX_COLS], 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(" %.2f, ", mat[i][j]);
        printf("\b\b\n");
    }
    printf(")\n");
}

float calcola_distanza_euclidea(float x_1, float y_1, float x_2, float y_2)
{

    float diff_x = x_1 - x_2;
    diff_x      = diff_x * diff_x;
    float diff_y = y_1 - y_2;
    diff_y      = diff_y * diff_y;

    float dist   = sqrt(diff_x + diff_y);
    return dist;
}
```

```

}

void calcola_matrice_distanze(float X[], float Y[], int n, float M[][MAX_COLS])
{
    for( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            M[i][j] = calcola_distanza_euclidea(X[i], Y[i], X[j], Y[j]);
    }
}

int main()
{
    float X[] = {1, 3, 5, 2, 4, 8};
    float Y[] = {2, 5, 9, 7, 2, 2};
    int n   = 6;
    printf("Coordinate:\n X: ");
    stampa_vett(X, n);
    printf(" Y: ");
    stampa_vett(Y,n);

    float M[MAX_ROWS][MAX_COLS];

    calcola_matrice_distanze(X,Y, n, M);
    printf("matrice delle distanze:\n");
    stampa_mat(M, n, n);
    return 0;
}

```

```

Coordinate:
X: (1.00, 3.00, 5.00, 2.00, 4.00, 8.00)
Y: (2.00, 5.00, 9.00, 7.00, 2.00, 2.00)
matrice delle distanze:
(
0.00,  3.61,  8.06,  5.10,  3.00,  7.00
3.61,  0.00,  4.47,  2.24,  3.16,  5.83
8.06,  4.47,  0.00,  3.61,  7.07,  7.62
5.10,  2.24,  3.61,  0.00,  5.39,  7.81
3.00,  3.16,  7.07,  5.39,  0.00,  4.00
7.00,  5.83,  7.62,  7.81,  4.00,  0.00
)
```

```

In [55]: /*
dati due vettori `X` ed `Y` contenenti le coordinate di una serie di punti,
calcolare una matrice `D` contenente le distanze tra questi punti (VERSIONE MIGLIORATA)
*/

/*
sappiamo che:
- gli elementi sulla diagonale principale sono pari a 0
- per ogni coppia (i,j), D(i,j) == D(j,i)
*/
#include <stdio.h>
#include <math.h>
#define MAX_COLS 10
#define MAX_ROWS 10
void stampa_vett(float vett[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
        printf("%.2f,", vett[i]);
    printf("\b)\n");
}

void stampa_mat(float mat[][MAX_COLS], 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(" %.2f,", mat[i][j]);
        printf("\b\n");
    }
    printf(")\n");
}

float calcola_distanza_euclidea(float x_1, float y_1, float x_2, float y_2)
{
    float diff_x = x_1 - x_2;
    diff_x      = diff_x * diff_x;
    float diff_y = y_1 - y_2;
    diff_y      = diff_y * diff_y;

    float dist   = sqrt(diff_x + diff_y);
    return dist;
}
```

```

}

void calcola_matrice_distanze(float X[], float Y[], int n, float M[][MAX_COLS])
{
    for( int i = 0; i < n; i++)
    {
        M[i][i] = 0; // gli elementi sulla diagonale principale sono sicuramente pari a 0
        for (int j = i+1; j < n; j++) // calcolo solo il triangolo superiore
        {
            M[i][j] = calcola_distanza_euclidea(X[i], Y[i], X[j], Y[j]);
            M[j][i] = M[i][j];           // copio il triangolo superiore
                                         // nel triangolo inferiore
        }
    }
}

int main()
{
    float X[] = {1, 3, 5, 2, 4, 8};
    float Y[] = {2, 5, 9, 7, 2, 2};
    int n = 6;
    printf("Coordinate:\n X: ");
    stampa_vett(X, n);
    printf(" Y: ");
    stampa_vett(Y,n);

    float D[MAX_ROWS][MAX_COLS];

    calcola_matrice_distanze(X,Y, n, D);
    printf("Matrice delle distanze:\n");
    stampa_mat(D, n, n);
    return 0;
}

```

```

Coordinate:
X: (1.00,3.00,5.00,2.00,4.00,8.00)
Y: (2.00,5.00,9.00,7.00,2.00,2.00)
matrice delle distanze:
(
0.00, 3.61, 8.06, 5.10, 3.00, 7.00
3.61, 0.00, 4.47, 2.24, 3.16, 5.83
8.06, 4.47, 0.00, 3.61, 7.07, 7.62
5.10, 2.24, 3.61, 0.00, 5.39, 7.81
3.00, 3.16, 7.07, 5.39, 0.00, 4.00
7.00, 5.83, 7.62, 7.81, 4.00, 0.00
)
```

```

In [33]: /*
Costruire un vettore che sia l'unione (in senso insiemistico) di 2 vettori dati
(ossia mettere gli elementi dei due vettori in un terzo vettore senza ripetizioni)
*/
#include <stdio.h>
#include <math.h>
#define MAX_LEN 100
void stampa_vett(int vett[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
        printf("%d, ", vett[i]);
    printf("\b\b)\n");
}

void unione(int vett_1[], int n_1, int vett_2[], int n_2, int vett_out[], int* n_out)
{
    *n_out = 0;
    for(int i = 0; i < n_1; i++)
    {
        int gia_presente = 0;
        for (int k = 0; k < *n_out; k++)
        {
            if (vett_out[k] == vett_1[i])
                gia_presente = 1;
        }

        if (gia_presente == 0)
        {
            vett_out[*n_out] = vett_1[i];
            (*n_out)++;
        }
    }

    for(int i = 0; i < n_2; i++)
    {
        int gia_presente = 0;
        for (int k = 0; k < *n_out; k++)
        
```

```

    {
        if (vett_out[k] == vett_2[i])
            gia_presente = 1;
    }

    if (gia_presente == 0)
    {
        vett_out[*n_out] = vett_2[i];
        (*n_out)++;
    }
}

int main()
{
    int v_1[] = {1,2,3,2,3,8,8,5,5,2,8,6};
    int v_2[] = {2,4,2,8,6,7,6,7};
    int n_1    = 12;
    int n_2    = 8;
    printf("primo vettore:\n");
    stampa_vett(v_1, n_1);
    printf("secondo vettore:\n");
    stampa_vett(v_2, n_2);

    int v_3[MAX_LEN];
    int n_3;
    unione(v_1, n_1, v_2, n_2, v_3, &n_3);
    printf("unione:\n");
    stampa_vett(v_3, n_3);

    return 0;
}

```

```

primo vettore:
(1, 2, 3, 2, 3, 8, 8, 5, 5, 2, 8, 6)
secondo vettore:
(2, 4, 2, 8, 6, 7, 6, 7)
unione:
(1, 2, 3, 8, 5, 6, 4, 7)

```

In [34]:

```

/*
Costruire un vettore che sia l'unione di 2 vettori dati
(ossia mettere gli elementi dei due vettori in un terzo vettore senza ripetizioni)
(versione più modulare)
*/
#include <stdio.h>
#include <math.h>
#define MAX_LEN 100
void stampa_vett(int vett[], int n)
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%d, ", vett[i]);
    printf("\b\b)\n");
}

int is_presente(int vett[], int n, int val)
{
    int trovato = 0;
    int i        = 0;
    while(trovato == 0 && i < n)
    {
        if( vett[i] == val)
            trovato = 1;
        i++;
    }

    return trovato;
}

void unione(int vett_1[], int n_1, int vett_2[], int n_2, int vett_out[], int* n_out)
{
    *n_out = 0;
    for(int i = 0; i < n_1; i++)
    {
        int gia_presente = is_presente(vett_out, *n_out, vett_1[i]);

        if (gia_presente == 0)
        {
            vett_out[*n_out] = vett_1[i];
            (*n_out)++;
        }
    }
}

```

```

        for(int i = 0; i < n_2; i++)
    {
        int gia_presente = is_presente(vett_out, *n_out, vett_2[i]);

        if (gia_presente == 0)
        {
            vett_out[*n_out] = vett_2[i];
            (*n_out)++;
        }
    }

}

int main()
{
    int v_1[] = {1,2,3,2,3,8,8,5,5,2,8,6};
    int v_2[] = {2,4,2,8,6,7,6,7};
    int n_1    = 12;
    int n_2    = 8;
    printf("primo vettore:\n");
    stampa_vett(v_1, n_1);
    printf("secondo vettore:\n");
    stampa_vett(v_2, n_2);

    int v_3[MAX_LEN];
    int n_3;
    unione(v_1, n_1, v_2, n_2, v_3, &n_3);
    printf("unione:\n");
    stampa_vett(v_3, n_3);

    return 0;
}

```

```

primo vettore:
(1, 2, 3, 2, 3, 8, 8, 5, 5, 2, 8, 6)
secondo vettore:
(2, 4, 2, 8, 6, 7, 6, 7)
unione:
(1, 2, 3, 8, 5, 6, 4, 7)

```

```

In [35]: /*
Costruire un vettore che sia l'intersezione di 2 vettori dati
(ossia mettere i soli elementi comuni tra i due vettori
in un terzo vettore senza ripetizioni)
*/
#include <stdio.h>
#define MAX_LEN 100
void stampa_vett(int vett[], int n)
{
    printf("(");
    for(int i=0; i<n; i++)
        printf("%d, ", vett[i]);
    printf("\b\b)\n");
}

int is_presente(int vett[], int n, int val)
{
    int trovato = 0;
    int i       = 0;
    while(trovato == 0 && i < n)
    {
        if( vett[i] == val)
            trovato = 1;
        i++;
    }

    return trovato;
}

void intersezione(int vett_1[], int n_1, int vett_2[], int n_2, int vett_out[], int* n_out)
{
    *n_out = 0;
    for(int i = 0; i < n_1; i++)
    {
        int gia_presente = is_presente(vett_out, *n_out, vett_1[i]);

        if (gia_presente == 0)
        {
            int presente_in_vett_2 = is_presente(vett_2, n_2, vett_1[i]);

            if (presente_in_vett_2 == 1)
            {
                vett_out[*n_out] = vett_1[i];
                (*n_out)++;
            }
        }
    }
}

```

```

        (*n_out)++;}
    }
}

int main()
{
    int v_1[] = {1,2,3,2,3,8,8,5,5,2,8,6};
    int v_2[] = {2,4,2,8,6,7,6,7};
    int n_1 = 12;
    int n_2 = 8;
    printf("primo vettore:\n");
    stampa_vett(v_1, n_1);
    printf("secondo vettore:\n");
    stampa_vett(v_2, n_2);

    int v_3[MAX_LEN];
    int n_3;
    intersezione(v_1, n_1, v_2, n_2, v_3, &n_3);
    printf("intersezione:\n");
    stampa_vett(v_3, n_3);

    return 0;
}

```

```

primo vettore:
(1, 2, 3, 2, 3, 8, 8, 5, 5, 2, 8, 6)
secondo vettore:
(2, 4, 2, 8, 6, 7, 6, 7)
intersezione:
(2, 8, 6)

```

In [37]:

```

/*
Dato un vettore A ed un vettore B di lunghezza <= alla lunghezza di A,
determinare se il vettore B è un sottovettore di A
(ossia se esiste una sequenza di elementi in A uguale
all'*intera* sequenza di elementi in B)
*/
#include <stdio.h>
#include <math.h>
#define MAX_LEN 100

```

```

void stampa_vett_e_idxs(int vett[], int n)
{
    printf(" idx: ");
    for(int i=0; i<n; i++)
        printf("[%d] ", i);
    printf("\n");
    printf("      (");
    for(int i=0; i<n; i++)
        printf("%3d,", vett[i]);
    printf("\b)\n");
}

```

```

int get_start_sottovettore(int vett_1[], int n_1, int vett_2[], int n_2)
{
    /*
    Restituisce l'indice di inizio della sequenza vett_2 in vett_1.
    Nel caso in cui tale sequenza non sia presente, restituisce -1
    */
    if( n_2 > n_1)
        return -1;
    int idx_start_sottovettore = -1;
    for(int i = 0; i <= n_1-n_2; i++)
    {
        int uguali = 1;
        for(int j=0; j < n_2; j++) // per ogni elemento nel secondo vettore
        {
            if (vett_1[i+j] != vett_2[j]) /* verifico se esistono coppie diverse tra vett_1 e vett_2
                                         a partire dalla posizione i-esima di vett_1 */
                uguali = 0;
        }
        if (uguali == 1)
            idx_start_sottovettore = i;
    }
    return idx_start_sottovettore;
}

int main()
{

```

```

int A[] = {1,2,3,2,3,8,8,5,5,2,8,6};
int B[] = {5,2,8,6};
int n_A = 12;
int n_B = 4;
printf("A: \n");
stampa_vett_e_idxs(A, n_A);
printf("B: \n");
stampa_vett_e_idxs(B, n_B);

int idx_start_sottovettore = get_start_sottovettore(A,n_A, B,n_B);

if(idx_start_sottovettore > 0)
    printf("B è sottovettore di A a partire dalla posizione %d.\n", idx_start_sottovettore);
else
    printf("B NON è sottovettore di A.\n");

}

A:
idx: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
( 1, 2, 3, 2, 3, 8, 8, 5, 5, 2, 8, 6)
B:
idx: [0] [1] [2] [3]
( 5, 2, 8, 6)
B è sottovettore di A a partire dalla posizione 8.

```

```

In [38]: /*
Dato un vettore A ed un vettore B di lunghezza <= alla lunghezza di A,
determinare se il vettore B è un sottovettore di A
(ossia se esiste una sequenza di elementi in A uguale all'*intera* sequenza di elementi in B)
(versione migliorata)
*/
#include <stdio.h>
#include <math.h>
#define MAX_LEN 100

void stampa_vett_e_idxs(int vett[], int n)
{

    printf(" idx: ");
    for(int i=0; i<n; i++)
        printf("[%d] ", i);
    printf("\n");
    printf("      (");
    for(int i=0; i<n; i++)
        printf("%3d,", vett[i]);
    printf("\b)\n");
}

int get_start_sottovettore(int vett_1[], int n_1, int vett_2[], int n_2)
{
/*
Restituisce l'indice di inizio della sequenza vett_2 in vett_1.
Nel caso in cui tale sequenza non sia presente, restituisce -1
*/
if( n_2 > n_1)
    return 0;
int idx_start_sottovettore = -1;

int i = 0;
while(i <= n_1-n_2 && idx_start_sottovettore < 0)
{
    int j = 0;
    while(j < n_2 && vett_1[i+j] == vett_2[j])
        j++;
    if (j == n_2) // se tutta la sequenza vett_2 è stata esaminata, allora è presente
        idx_start_sottovettore = i;
    i++;
}
return idx_start_sottovettore;
}

int main()
{
    int A[] = {1,2,3,2,3,8,8,5,5,2,8,6};
    int B[] = {5,2,8,6};
    int n_A = 12;
    int n_B = 4;
    printf("A: \n");
    stampa_vett_e_idxs(A, n_A);
    printf("B: \n");
    stampa_vett_e_idxs(B, n_B);

    int idx_start_sottovettore = get_start_sottovettore(A,n_A, B,n_B);
}

```

```
    if(idx_start_sottovettore > 0)
        printf("B è sottovettore di A ed inizia dalla posizione %d.\n", idx_start_sottovettore);
    else
        printf("B NON è sottovettore di A.\n");

}
```

A:

```
idx: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
( 1, 2, 3, 2, 3, 8, 8, 5, 5, 2, 8, 6)
```

B:

```
idx: [0] [1] [2] [3]
( 5, 2, 8, 6)
```

B è sottovettore di A ed inizia dalla posizione 8.

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

TIME

```
#include <time.h>
```

Header per sfruttare funzioni di libreria atte al fornire informazioni riguardo al "tempo".

Funzioni principali:

- `time_t time(time_t* arg);` restituisce un object di tipo `time_t` contenente la data e l'ora attuali. Generalmente, `time_t` corrisponde ad un semplice intero positivo contenente il numero di secondi trascorsi dal 1 gennaio 1970 fino ad oggi. L'argomento `arg` è un puntatore che, al termine della funzione, conterrà lo stesso valore restituito da quest'ultima (può quindi essere ignorato passandogli `NULL`);

```
In [12]: #include <stdio.h>
#include <time.h>
int main()
{
    int sec_dal_1970 = time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %d\n", sec_dal_1970);
    printf("numero di anni dal 1 gennaio 1970: %f\n", sec_dal_1970/60.0/60.0/24.0/365.0);
    return 0;
}
```

numero di secondi trascorsi dal 1 gennaio 1970: 1649163282
numero di anni dal 1 gennaio 1970: 52.294625

```
In [ ]:
```

- `struct tm* gmtime(const time_t *timer);` Dato in input un numero di secondi trascorsi, restituisce un puntatore ad una `struct` contenente la data (UTC) distribuiti su vari campi.

```
struct tm {  
</table>};
```

```
In [4]: #include <stdio.h>
#include <time.h>
int main()
{
    time_t sec_dal_1970 = time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %d\n", sec_dal_1970);
    struct tm* data = gmtime(&sec_dal_1970);
    printf("data (UTC): giorno:%d mese:%d anno:%d ore:%d minuti:%d secondi:%d\n",
           data->tm_mday, data->tm_mon+1, 1900+data->tm_year,
           data->tm_hour, data->tm_min, data->tm_sec);

    return 0;
}
```

numero di secondi trascorsi dal 1 gennaio 1970: 1649245950
data (UTC): giorno:6 mese:4 anno:2022 ore:11 minuti:52 secondi:30

- `struct tm *localtime (const time_t *timer);`

come `gmtime`, ma la data viene espressa secondo il fuso orario locale

- `char* asctime(const struct tm *timeptr);`

Dato l'indirizzo di una struct di tipo `tm`, restituisce una stringa contenente la data già formattata

```
In [3]: #include <stdio.h>
#include <time.h>
int main()
{
    time_t sec_dal_1970 = time(NULL);
    printf("numero di secondi trascorsi dal 1 gennaio 1970: %d\n", sec_dal_1970);
    struct tm* data = localtime(&sec_dal_1970);
    char* string_data = asctime(data);
    printf("data (locale): %s\n", string_data );

    return 0;
}
```

numero di secondi trascorsi dal 1 gennaio 1970: 1649245935
 data (locale): Wed Apr 6 13:52:15 2022

Random numbers

La funzione `rand()` restituisce un numero *pseudo-casuale* nell'intervallo tra 0 e `RAND_MAX`.

`RAND_MAX` è una costante il cui valore può essere diverso in base alle diverse implementazioni, ma sicuramente è ≥ 32767

```
In [15]: #include <stdio.h>
#include <time.h>
int main()
{

    printf("RAND_MAX: %lld", RAND_MAX);
    return 0;
}
```

RAND_MAX: 2147483647

```
In [16]: #include <stdio.h>
#include <time.h>
int main()
{
    long int r = rand();
    printf("%lld\n", r);
    return 0;
}
```

1804289383
 846930886
 1681692777
 1714636915

Diverse esecuzioni della funzione `rand()` daranno sempre lo stesso numero! Questo perchè, essendo *pseudo-casuali* (e non casuali) sono generati sempre in funzione del valore precedente ottenuto *durante l'esecuzione*. In altri termini, il t -esimo numero pseudocasuale N_t cambia in funzione del numero casuale precedente, i.e. $N_t = f(N_{t-1})$. Il primo di questi valori, N_0 , dipende da un valore iniziale detto *seme* N_0 , tipicamente pari a 1.

Un modo per avere valori diversi è quello di inizializzare diversamente il valore seme della sequenza (in altri termini N_0) la funzione

`void srand(unsigned seed);`

imposta il seme al valore `seed`.

L'ideale sarebbe avere un seme diverso ad ogni esecuzione → si può usare il numero di secondi restituiti da `time(...)`.

```
In [17]: #include <stdio.h>
#include <time.h>
int main()
{
    srand(time(NULL));
    long int r = rand();
    printf("%lld\n", r);
    r = rand();
    printf("%lld\n", r);
```

```

r = rand();
printf("%lld\n", r);
r = rand();
printf("%lld\n", r);
}

```

```

1572501703
2021161100
1915516915
624023057

```

Numero intero random in intervallo $\{0, 1, \dots, b\}$

Esempio: voglio un numero casuale tra 0 e 5 (estremi compresi).

Possibile ragionamento: sfruttare l'operatore modulo. Si ricorda che, dati due numeri interi d e u , $mod(d, u)$ sarà un numero intero tale che $0 \leq mod(d, u) \leq u - 1$.

$mod(1, 5) = 1$, $mod(2, 5) = 2$, $mod(3, 5) = 3$, $mod(4, 5) = 4$, $mod(5, 5) = 0$, $mod(6, 5) = 1$, $mod(7, 5) = 2$, $mod(8, 5) = 3$, ...

Quindi, per avere un numero tra 0 e 5, nell'operazione $mod(d, u)$ basta impostare $u = 5 + 1 = 6$ mentre d può essere qualsiasi numero (pseudo)casuale, i.e.

$$0 \leq mod(rand(), b + 1) \leq b$$

```
In [18]: #include <stdio.h>
#include <time.h>

long int random_int_in_max(int b)
{
    return rand() % (b + 1);
}

int main()
{
    int max_number = 5;
    srand( time(NULL) );
    for (int i = 0; i < 50; i++)
    {
        long int r = random_int_in_max(max_number);
        printf("%lld ", r);
    }
}
```

```
5 0 1 4 3 2 2 0 4 5 2 3 4 2 1 3 2 5 5 5 0 3 0 0 4 0 0 5 5 5 3 4 3 2 0 1 4 2 5 0 1 0 2 4 2
3 1 4 2 0
```

Numero intero random in intervallo $\{a, a + 1, \dots, b\}$

Esempio: voglio un numero casuale tra 2 e 5 (estremi compresi).

Possibile ragionamento: innanzitutto devo sapere *quanti* sono i numeri effettivi nell'intervallo desiderato. In questo caso, il numero desiderato può essere uno nell'insieme $\{2, 3, 4, 5\}$, per un totale di $w = 4$ numeri. Generalizzando, $w = b + 1 - a$.

$c = mod(rand(), w)$ mi darà un numero casuale intero il cui valore sarà uno tra 0 e $w - 1$.

Sommando al valore c il valore a , si avrà quindi un numero casuale compreso tra $0 + a = a$ e $w - 1 + a = (b + 1 - a) - 1 + a = b$.

```
In [5]: #include <stdio.h>
#include <time.h>

long int random_int_in_range(int a, int b)
{
    return rand() % (b + 1 - a) + a;
}

int main()
{
    srand( time(NULL) );
    for(int i = 0; i<30; i++)
    {
        long int r = random_int_in_range(2,5);
        printf("%lld ", r);
    }
}
```

```

    }
    return 0;
}

5 4 3 5 5 5 5 4 4 2 5 3 3 2 5 5 3 4 2 2 3 4 4 2 2 5 2 3 2

```

Numero reale random in intervallo [0,1]

$$0 \leq \text{rand}() \leq \text{RAND_MAX} \Rightarrow 0 \leq \frac{\text{rand}()}{\text{RAND_MAX}} \leq 1$$

```
In [20]: #include <stdio.h>
#include <time.h>

double random_real_in_0_1()
{
    double x = (float)rand() / RAND_MAX;
    return x;
}

int main()
{
    srand( time(NULL) );
    for(int i = 0; i<20; i++)
    {
        double r = random_real_in_0_1();
        printf("%lf ", r);
    }
    return 0;
}
```

0.310926 0.685900 0.302752 0.254879 0.047773 0.926345 0.087862 0.852664 0.104581 0.785533
0.016734 0.426548 0.722991 0.011384 0.020128 0.640147 0.550135 0.839033 0.135204 0.944068

Numero reale random in intervallo [0,b]

```
In [21]: #include <stdio.h>
#include <time.h>

double random_real_in_max(int b)
{
    double x = ((float)rand() / RAND_MAX) * b;
    return x;
}

int main()
{
    srand( time(NULL) );
    for(int i = 0; i<20; i++)
    {
        double r = random_real_in_max(5);
        printf("%lf ", r);
    }
    return 0;
}
```

3.362904 0.500493 0.544300 0.395331 2.416048 0.755194 2.269607 0.747441 2.071917 1.881912
2.642678 2.966139 3.159104 2.831061 2.585112 3.044364 3.678777 0.545692 4.196876 3.055914

Numero reale random in intervallo [a,b]

mi calcolo la distanza tra b ed a , ossia $w = b - a$.

$$\text{Dato che } 0 \leq \frac{\text{rand}()}{\text{RAND_MAX}} \leq 1, \text{ allora } w \cdot 0 \leq w \cdot \frac{\text{rand}()}{\text{RAND_MAX}} \leq w \cdot 1.$$

Sommando a :

$$a + 0 \leq a + w \cdot \frac{\text{rand}()}{\text{RAND_MAX}} \leq a + w = a + b - a = b.$$

$$\text{Quindi: } a \leq a + (b - a) \cdot \frac{\text{rand}()}{\text{RAND_MAX}} \leq b$$

```
In [6]: #include <stdio.h>
#include <time.h>

double random_real_in_range(int a, int b)
{
```

```

        double x = ((b - a) * ((double)rand() / RAND_MAX)) + a;
        return x;
    }

int main()
{
    srand( time(NULL) );
    for(int i = 0; i<100; i++)
    {
        double r = random_real_in_range(1,2);
        printf("%lf ", r);
    }
    return 0;
}

```

```

1.571983 1.036639 1.856773 1.002382 1.501812 1.947896 1.944476 1.561045 1.482949 1.864293
1.837613 1.474696 1.919647 1.270162 1.042264 1.890886 1.126113 1.479243 1.303184 1.307271
1.036848 1.014611 1.182290 1.875430 1.805373 1.309830 1.572245 1.653529 1.573379 1.587789
1.830877 1.145362 1.624427 1.687650 1.147744 1.126239 1.635547 1.092220 1.687285 1.118496
1.956513 1.524897 1.593192 1.876160 1.795059 1.635456 1.767046 1.921173 1.114699 1.070229
1.228443 1.151547 1.084840 1.410733 1.026977 1.890213 1.720563 1.599222 1.543742 1.293942
1.187010 1.374620 1.439304 1.811438 1.062270 1.587048 1.937677 1.697817 1.679268 1.624962
1.816312 1.635782 1.149859 1.409504 1.511942 1.944918 1.044960 1.278987 1.866091 1.159660
1.349216 1.094534 1.311207 1.434056 1.505267 1.338184 1.324269 1.225830 1.937406 1.868012
1.519772 1.124416 1.242631 1.959076 1.935854 1.304901 1.546124 1.873532 1.002718 1.225392

```

int tm_sec	seconds after the minute;
int tm_min	minutes after the hour;
int tm_hour	hours since midnight; [0, 23]
int tm_mday	day of the month; [1, 31]
int tm_mon	months since January; [0, 11]
int tm_year	years since 1900;
int tm_wday	days since Sunday; [0, 6]
int tm_yday	days since January 1; [0, 365]

Processing math: 100%

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Verificare se un vettore è ordinato

Scrivere un algoritmo che, dato in input un array, restituisca

- 1 se tale array è ordinato
- 0 altrimenti

```
In [3]: // versione 1
#include <stdio.h>
int is_sorted(float vett[], int dim)
{
    int sorted = 1;
    for(int i = 1; i < dim; i++)
        if( vett[i-1] > vett[i])
            sorted = 0;
    return sorted;
}

int main()
{
    float v[] = {1, 4, 6, 8, 2}; //{3,1,2,4,6,8,9,-5};
    int n      = 5; //8;
    int ord   = is_sorted(v,n);

    if(ord == 0)
        printf("il vettore non è ordinato\n");
    else
        printf("il vettore è ordinato\n");
}
```

il vettore non è ordinato

```
In [4]: // versione 2
#include <stdio.h>
int is_sorted(float vett[], int dim)
{
    int sorted = 1;
    int i      = 1;
    while(i < dim && sorted == 1)
    {
        if( vett[i-1] > vett[i])
            sorted = 0;
        i++;
    }
    return sorted;
}

int main()
{
    float v[] = {1, 4, 6, 8}; //{3,1,2,4,6,8,9,-5};
    int n      = 4; //8;
    int ord   = is_sorted(v,n);

    if(ord == 0)
        printf("il vettore non è ordinato\n");
    else
        printf("il vettore è ordinato\n");
}
```

il vettore è ordinato

Ordinamento per selezione

```
In [5]: #include <stdio.h>
```

```

void stampa_vett(float v[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
        printf("%.2f, ", v[i]);
    printf("\b\b)\n");
}

float idx_min_array(float v[], int n, int start)
{
    int idx_min = start;
    float min   = v[start];
    // Loop interno
    for (int i=start+1; i<n; i++)
        if (v[i] < min)
    {
        min     = v[i];
        idx_min = i;
    }
    return idx_min;
}

void selection_sort(float v[], int n)
{
    int idx_min;
    // Loop esterno
    for(int i=0; i<n-1; i++) // n-1 in quanto ultimo elemento sarà già nella posizione corretta grazie
                                // agli spostamenti precedenti
    {
        idx_min     = idx_min_array(v, n, i);
        float tmp   = v[i];
        v[i]        = v[idx_min];
        v[idx_min] = tmp;
    }
}

```

```

int main()
{
    float vett[] = {8,4,2,1,5,7,9,3,6};
    int n       = 9;
    printf("prima: ");
    stampa_vett(vett,n);
    selection_sort(vett,n);
    printf("dopo: ");
    stampa_vett(vett,n);

    return 0;
}

```

prima: (8.00, 4.00, 2.00, 1.00, 5.00, 7.00, 9.00, 3.00, 6.00)
dopo: (1.00, 2.00, 3.00, 4.00, 5.00, 6.00, 7.00, 8.00, 9.00)

Il tempo impiegato dalla funzione `selection_sort(...)` dipende dal tempo t_{loop} impiegato da ogni iterazione del loop esterno.

t_{loop} , a sua volta, non è altro che la somma tra:

- il tempo $t_{min}(k)$ speso per eseguire la funzione `idx_min_array(...)`. Tale funzione non è altro che una ricerca del minimo su un (sotto)array di v di lunghezza k , con k che diventa, ad ogni iterazione del loop esterno, più piccolo di 1;
 - il tempo t_{swap} speso per fare uno scambio tra due elementi (3 operazioni elementari, indipendente dalla lunghezza dell' array)

Quindi $t_{loop} = t_{min}(k) + t_{swap}$

Inoltre, sappiamo già che $t_{min}(k)$ è **lineare** sulla dimensione dell' input meno 1, cioè $k - 1$. Possiamo quindi esprimere come $t_{min}(k) = a(k - 1) + b$, con a e b coefficienti della parte rispettivamente dipendente e indipendente dalla dimensione dell' input k .

L'algoritmo di Selection Sort può essere visto come $n - 1$ ricerche del minimo, ognuna di queste effettuata su un array di dimensioni sempre più piccola, con relativo scambio.

- 1° iterazione: ricerca del minimo su un array di n elementi + scambio
 - 2° iterazione: ricerca del minimo su un array di $n - 1$ elementi + scambio
 - 3° iterazione: ricerca del minimo su un array di $n - 2$ elementi + scambio
- ⋮
- $(n-1)^{\circ}$ iterazione: ricerca del minimo su un array di 2 elementi + scambio

La complessità complessiva dell'algoritmo sarà data dalla somma de:

- i tempi per le ricerche dei minimi t_{min} .

- il tempo per effettuare tutti gli scambi, che è facilmente calcolabile come $(n - 1) \cdot t_{swap}$

Concentriamoci adesso sul tempo complessivo per le ricerche dei minimi:

$$\begin{aligned}
 t_{min}(n) + t_{min}(n-1) + t_{min}(n-2) + \dots + t_{min}(2) &= \\
 &= \sum_{k=2}^n t_{min}(k) \\
 &= \sum_{k=2}^n (a(k-1) + b) = \sum_{k=1}^{n-1} (a \cdot k + b) = a \sum_{k=1}^{n-1} k + (n-1)b
 \end{aligned}$$

La prima sommatoria è la somma dei primi $n - 1$ numeri interi, quindi si può usare la formula di Gauss ($\sum_{i=1}^n i = n(n+1)/2$):

$$\sum_{k=1}^{n-1} k = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$$

Il tempo complessivo dell'implementazione proposta di Selection Sort è dato da

$$a \frac{n^2 - n}{2} + (b + t_{swap})(n - 1)$$

Il termine "dominante" di questa espressione è n^2 , quindi il tempo necessario al completamento dell'algoritmo varia in maniera *quadratica* al variare della dimensione dell'input.

Ricerca binaria

```
In [6]: #include <stdio.h>
void stampa_vett_part(int v[], int inizio_idx, int fine_idx)
{
    printf("(");
    if( fine_idx < inizio_idx )
        printf(" ");
    for(int i = inizio_idx; i <= fine_idx; i++)
        printf("%d, ", v[i]);
    printf("\b\b)");
}

void stampa_vett(int v[], int n)
{
    stampa_vett_part(v, 0, n-1);
}

int ricerca_binaria(int v[], int n,  int key)
{
    int idx_inizio    = 0;
    int idx_fine     = n-1;

    int idx_trovato  = -1;

    int it = 0; // utile solo per visualizzazione
    while(idx_inizio <= idx_fine && idx_trovato == -1)
    {
        int idx_centrale = (idx_inizio + idx_fine) / 2;
        /*
        printf("it:%d sottovett. sx: ", it+1);
        stampa_vett_part(v, idx_inizio, idx_centrale-1);
        printf(" centrale: |%d| ", v[idx_centrale]);
        printf("sottovett. dx: ");
        stampa_vett_part(v, idx_centrale+1, idx_fine);
        printf("\n");
        */
        if (v[idx_centrale] == key) // cerco al centro
            idx_trovato = idx_centrale;
        else if(key > v[idx_centrale]) // cerco nel lato destro
            idx_inizio    = idx_centrale + 1;
        else
            idx_fine     = idx_centrale - 1; // cerco nel lato sinistro
        it++; // utile solo per visualizzazione
    }
    return idx_trovato;
}

int main()
{
    int vett[]    = {1,4,6,8,9,10,11};
    int n         = 7;
```

```

int k      = 11;
printf("vettore: ");
stampa_vett(vett,n);
printf("\n");
int idx_k = ricerca_binaria(vett,n,k);
if(idx_k == -1)
    printf("il valore %d non si trova nel vettore\n", k);
else
    printf("il valore %d si trova nel vettore in posizione %d.\n", k, idx_k);
return 0;
}

```

vettore: (1, 4, 6, 8, 9, 10, 11)
il valore 11 si trova nel vettore in posizione 6.

Il corpo del ciclo è composto da poche istruzioni elementari. Definiamo t_{loop} il tempo per eseguire una singola iterazione.

Bisogna quindi valutare quante volte viene ripetuto il corpo del ciclo.

Caso migliore: l'elemento da cercare si trova proprio al centro dell'array \Rightarrow tempo indipendente dalla dimensione dell'array

Caso peggiore: vengono effettuate tutte le iterazioni possibili (ad esempio, nel caso l'elemento non è presente oppure se l'elemento si trova in uno dei due estremi dell'array). In questi casi, l'array viene ripetutamente spezzato in due parti (una parte sx ed una parte dx). La parte in esame quindi diventa sempre più piccola di un fattore 2 ad ogni iterazione, fino a raggiungere la dimensione di 1 elemento.

- 1° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento l'intero array
 - 2° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento una metà ($\frac{n}{2}$) dell'array
 - 3° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento una metà della metà ($\frac{n}{4}$) dell'array
 - 4° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento $\frac{n}{8}$ dell'array
- ⋮
- k° iterazione: il blocco nel ciclo viene eseguito avendo come riferimento $\frac{n}{2^{k-1}}$ dell'array

il ciclo si fermerà quando si arriverà al caso di un array di lunghezza 1, i.e. quando $\frac{n}{2^{k-1}} = 1 \Rightarrow 2^{k-1} = n \Rightarrow k-1 = \log_2 n \Rightarrow k = \log_2 n + 1$

l'algoritmo quindi nel caso peggiore terminerà dopo $k = \log_2 n + 1$ iterazioni.

Tempo di esecuzione varia quindi in maniera **logaritmica** al variare della dimensione dell'input!

Esempio: per un array di lunghezza 64, sono necessarie al più appena $\log_2 64 = 6$ iterazioni del ciclo per completare la ricerca.

\Rightarrow in generale più veloce della ricerca lineare!

Inserimento ordinato in array

```

In [7]: #include <stdio.h>
#define MAXDIM 100
void stampa_vett_part(int v[], int inizio_idx, int fine_idx)
{
    printf("(");
    if( fine_idx < inizio_idx)
        printf(" ");
    for(int i = inizio_idx; i <= fine_idx; i++)
        printf("%d, ", v[i]);
    printf("\b\b)");
}
void stampa_vett(int v[], int n)
{
    stampa_vett_part(v, 0, n-1);
}

void inserisci_ordinato(int vett[], int dim, int val)
{
    int i = 0;

    // scorro finchè non trovo un elemento >= val
    while(i < dim && vett[i] < val)
        i++;
    /* una volta trovato (se esiste) sposto a dx di 1
       tutti gli elementi alla sua dx
    */
    for(int j=dim; j > i; j--)
        vett[j] = vett[j-1];
    // inserisco val nella posizione "liberata"
    vett[i] = val;
}

```

```
int main()
{
    int n;
    int v[MAXDIM];
    printf("quanti valori vuoi inserire?");
    scanf("%d", &n);
    for(int i = 0; i < n; i++)
    {
        int val;
        printf("Inserisci il prossimo valore:");
        scanf("%d", &val);
        inserisci_ordinato(v, i, val);
    }
    stampa_vett(v, n);
    return 0;
}
```

quanti valori vuoi inserire?

Inserisci il prossimo valore:

Inserisci il prossimo valore:

Inserisci il prossimo valore:

(1, 5, 6)

L'inserimento ordinato funziona solo se il vettore di partenza è a sua volta ordinato \Rightarrow è necessario utilizzare **sempre** la funzione di inserimento ordinato (fin dal primo elemento) per garantire che quest'ultimo rimanga ordinato.

Domanda: dato che il vettore è ordinato, si può sfruttare la ricerca binaria per cercare la posizione in cui inserire l'elemento?

Risposta: Sì, ma poi comunque, una volta trovata la posizione di inserimento, avrei dovuto spostare gli elementi per far spazio al nuovo elemento, vanificando in parte i benefici della ricerca binaria.

In []:

Processing math: 100%

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Fusione di due vettori ordinati

Supponiamo che abbia due Vettori **ordinati**:

$$\mathbf{v}_1 = (2, 9, 1)$$

$$\mathbf{v}_2 = (5, 7, 4)$$

Si vuole ottenere un vettore \mathbf{v}_{out} che contenga:

- gli stessi elementi di \mathbf{v}_1 e \mathbf{v}_2
- che sia ordinato

$$\mathbf{v}_{\text{out}} = (2, 5, 7, 9, 14, 21)$$

Versione ingenua:

1. concateno i due vettori in un unico vettore
2. ordino il vettore concatenato

Funziona, ma il tempo di esecuzione è dominato dal tempo necessario all'ordinamento dell'array di lunghezza $n_1 + n_2$ (e.g., *quadratico* nel caso di selection sort).

Si può fare di meglio?

```
In [17]: #include <stdio.h>
void stampa_vett_part(int v[], int inizio_idx, int fine_idx)
{
    printf("(");
    if( fine_idx < inizio_idx )
        printf(" ");
    for(int i = inizio_idx; i <= fine_idx; i++)
        printf("%d, ", v[i]);
    printf("\b\b)");
}

void stampa_vett(int v[], int n)
{
    stampa_vett_part(v, 0, n-1);
}

void fondi(int v1[], int n1, int v2[], int n2, int v_out[], int* n_out)
{
    int idx_v1    = 0;
    int idx_v2    = 0;
    int idx_v_out = 0;

    // while
    while(idx_v1 < n1 && idx_v2 < n2)
    {
        if (v1[idx_v1] < v2[idx_v2])
        {
            v_out[idx_v_out] = v1[idx_v1];
            idx_v1++;
        }
        else
        {
            v_out[idx_v_out] = v2[idx_v2];
            idx_v2++;
        }
        idx_v_out++;
    }
}
```

```

if(idx_v1 < n1)
{
    // For1
    for(int i=idx_v1; i < n1; i++)
    {
        v_out[idx_v_out] = v1[i];
        idx_v_out++;
    }
}

if(idx_v2 < n2)
{
    // For2
    for(int i = idx_v2; i < n2; i++)
    {
        v_out[idx_v_out] = v2[i];
        idx_v_out++;
    }
}
*n_out = idx_v_out;
}

int main()
{
    int vett1[] = {0,1,3,6,7,9};
    int vett2[] = {-1,0,2,4,8,10,12,14};
    int n1 = 6;
    int n2 = 8;
    int vett_out[100];
    int n_out;
    printf("primo vett: ");
    stampa_vett(vett1,n1);
    printf("\nsecondo vett: ");
    stampa_vett(vett2,n2);
    fondi(vett1, n1, vett2, n2, vett_out, &n_out);
    printf("\nfusion vett: ");
    stampa_vett(vett_out,n_out);

    return 0;
}

```

```

primo vett: (0, 1, 3, 6, 7, 9)
secondo vett: (-1, 0, 2, 4, 8, 10, 12, 14)
fusion vett: (-1, 0, 0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 12, 14)

```

Il tempo di esecuzione nel caso peggiore, trascurando eventuali tempi di esecuzione costanti (e.g., inizializzazioni ecc.), è dato dal tempo di 3 componenti:

1. il tempo di ogni iterazione del 1° ciclo t_{while} , ripetuto il numero di iterazioni necessarie
2. il tempo di ogni iterazione del 2° ciclo t_{for_1} , ripetuto il numero di iterazioni necessarie
3. il tempo di ogni iterazione del 3° ciclo t_{for_2} , ripetuto il numero di iterazioni necessarie

Se i due array sono della stessa lunghezza, i passaggi 2. e 3. non vengono eseguiti (tranne che per un eventuale ultimo elemento rimasto in uno dei due array, ossia per una singola iterazione il cui tempo può considerarsi trascurabile).

Da notare che i due cicli `for`

- sono mutuamente esclusivi (o viene eseguito uno o l'altro, mai entrambi)
- eseguono le stesse operazioni (anche se su array diversi)

$$\Rightarrow t_{for_1} = t_{for_2}.$$

- il corpo del `while` esegue le stesse operazioni di uno dei due cicli `for` (in maniera esclusiva), quindi a sua volta impiega il loro stesso tempo

$$\Rightarrow t_{while} = t_{for_1} = t_{for_2} = t_*.$$

Analizziamo adesso i tempi dei 3 passaggi:

1. t_{while} terminerà quando o v_1 o v_2 sarà stato interamente scansionato. Da notare che gli indici di scansionamento sono diversi. Quindi, nel caso peggiore, i due indici possono procedere in maniera alternata $\Rightarrow t_{while}$ sarà ripetuto 2 min

Esempio: $v_1 = (2,4,6)$, $v_2 = (1,5,7,8)$.

E' evidente dall'algoritmo che, in questo caso, i due array verranno scansionati in maniera alternata, quindi il ciclo sarà composto da $6 = 2 \cdot \min(3,4)$ iterazioni.

Se $n_1=n_2=n$, i.e., i due vettori sono della stessa lunghezza, il tempo necessario per l'algoritmo di selection sort sarà di $t_* \cdot 2n$ (più il tempo per sistemare l'elemento residuo, che come abbiamo detto consideriamo ininfluente).

Altrimenti:

1. se $n_1 > n_2$, sarà eseguito il ciclo `For1`, che prenderà un tempo $t_{\{for_1\}}$ moltiplicato per gli elementi rimanenti in v_1 , ossia $n_1 - n_2 \Rightarrow t_{\{for_1\}}$ sarà ripetuto $n_1 - n_2$ volte.

2. se $n_1 < n_2$, sarà eseguito `For2`, con ragionamento è analogo al caso precedente $\Rightarrow t_{\{for_2\}}$ sarà ripetuto $n_2 - n_1$.

Dato che 2. e 3. sono mutuamente esclusivi ed uguali in termini di tempo richiesto, analizziamo il tempo totale prendendo il caso 2. In tal caso il tempo dell'algoritmo sarà dato da

$$2 \cdot n_2 \cdot t_{\{\text{while}\}} + (n_1 - n_2) \cdot t_{\{\text{for_1}\}} = n_2 \cdot t_* + n_2 \cdot t_* + (n_1 - n_2) \cdot t_* = (n_1 + n_2) \cdot t_*$$

il tempo di esecuzione dell'algoritmo di merge varia quindi in modo *lineare* al variare della *somma* delle dimensioni dei due array.

Può essere utilizzata per implementare un nuovo algoritmo di ordinamento: **merge sort** [trattato nelle prossime lezioni]

Partizione di un array

Dicesi *partizione* di un array $\mathbf{v} = (v_1, v_2, \dots, v_n)$ rispetto ad un suo valore k , un array avente gli stessi elementi di \mathbf{v} tale che:

1. tutti i valori $< k$ si trovino prima di k
2. tutti i valori $\geq k$ si trovino dopo k .

Esempio: $\mathbf{v} = (1, 4, 5, 2, 3, 6, 8)$, $\text{idx_k}=3$ (i.e. $k=5$) \Rightarrow

`part(v,3)=(1, 4, 2, 3, 5, 6, 8)`

ma anche

`part(v,3)=(1, 4, 2, 3, 5, 8, 6)`

oppure

`part(v,3)=(4, 2, 1, 3, 5, 8, 6)`

- Possono esistere più partizioni di uno stesso array rispetto allo stesso valore k
- Esempio di Partizione banale: array ordinato \Rightarrow per ogni elemento k , tutti i valori $< k$ staranno a sinistra di k , tutti gli altri elementi staranno a destra di k . \Rightarrow `part(v,1)` è `part(v,2)`, ma anche `part(v,3)`, ..., ed anche `part(v,n)`.
- Esiste un algoritmo in grado di restituirmi una partizione di un vettore \mathbf{v} rispetto ad un valore k senza la necessità di ordinarlo?
 - Sì. Esistono diverse strategie, ad esempio la seguente:

In [1]:

```
#include <stdio.h>

// stampa di una array di n elementi, assieme agli indici e due "marcatori"
// utile solo per capire cosa sta succedendo
void stampa_vett(int v[], int n, int idx, int idx2)
{
    printf("idxs: [ ");
    for(int i = 0; i < n; i++)
        printf("%d, ", i);
    printf("\b\b]\nvals: ( ");
    for(int i = 0; i < n; i++)
        printf("%.d, ", v[i]);
    printf("\b\b)\n");
    if (idx >= 0)
    {
        printf("idx1: ");
        for(int i = 1; i <= idx; i++)
            printf("   ");
        printf(" ^\n");
    }

    if (idx2 >= 0)
    {
        printf("idx2: ");
        for(int i = 1; i <= idx2; i++)
            printf("   ");
        printf(" *\n");
    }
}

void swap(int* a, int* b)
```

```

{
    int tmp = *a;
    *a      = *b;
    *b      = tmp;
}

void partition(int v[], int n, int idx_k) // versione di Lomuto
{
    // scambio l'ultimo elemento con quello in posizione idx_k
    printf("preparazione: scambio posizione %d con posizione %d\n", idx_k, n-1);
    // scambio l'ultimo elemento con quello in posizione idx_k
    swap(&v[idx_k], &v[n-1]); // adesso l'ultima posizione di v conterrà il valore k

    stampa_vett(v,n, -1, -1);

    int idx_store = 0; // indice del primo elemento >= di k
    for(int i = 0; i < n-1; i++)
    {
        printf("iterazione: %d\n", i+1);
        stampa_vett(v,n, idx_store, i);
        if (v[i] < v[n-1])
        {

            swap(&v[i], &v[idx_store]); // dopo lo scambio, v[idx_store] sarà < di k,
                                         // quindi si ipotizza che il successivo sia >=k
            idx_store++;
        }
    }
    swap(&v[n-1], &v[idx_store]);
}

int main()
{
    int v[] = {1,4,5,2,6,3,8};
    int n   = 7;
    printf("stato iniziale: \n");
    stampa_vett(v,n,-1,-1);
    int idx_val = 2;
    printf("inizio partizione rispetto a : %d (posizione %d)\n",v[idx_val], idx_val);
    partition(v,n, idx_val);
    printf("fine partizione. \n");
    printf("stato finale: \n");
    stampa_vett(v,n, -1,-1);
}

```

stato iniziale:
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 5, 2, 6, 3, 8)
 inizio partizione rispetto a : 5 (posizione 2)
 preparazione: scambio posizione 2 con posizione 6
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 8, 2, 6, 3, 5)
 iterazione: 1
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 8, 2, 6, 3, 5)
 idx1: ^
 idx2: *
 iterazione: 2
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 8, 2, 6, 3, 5)
 idx1: ^
 idx2: *
 iterazione: 3
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 8, 2, 6, 3, 5)
 idx1: ^
 idx2: *
 iterazione: 4
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 8, 2, 6, 3, 5)
 idx1: ^
 idx2: *
 iterazione: 5
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 2, 8, 6, 3, 5)
 idx1: ^
 idx2: *
 iterazione: 6
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 2, 8, 6, 3, 5)
 idx1: ^
 idx2: *
 fine partizione.
 stato finale:
 idxs: [0, 1, 2, 3, 4, 5, 6]
 vals: (1, 4, 2, 3, 5, 8, 6)

Cerchiamo di capire meglio il blocco principale dell'algoritmo:

```

if (v[i] < v[n-1])
{
    swap(&v[i], &v[idx_store]); // dopo lo scambio, v[idx_store] sarà < di k,
                                // quindi si ipotizza che il successivo sia >=k
    idx_store++;
}

idx_score inizialmente è = 0 . La variabile i scorre l'array partendo da 0.

```

Vediamo cosa succede nei diversi casi:

- `v[i] < v[n-1]` : viene scambiato `v[i]` con `v[idx_store]`. Notare che, se `i==idx_store`, lo scambio è ininfluente (viene scambiato l'elemento con se stesso). L'importante è che viene incrementato `idx_score`. `i` viene ovviamente incrementata.
- `v[i] >= v[n-1]` : nessuno scambio viene effettuato. `i` viene ovviamente incrementata.

Quindi abbiamo che:

- `idx_score` si fermerà quando `v[idx_score]>=k` (in quanto, se `v[idx_score]` fosse stato minore di `k`, sarebbe stato scambiato con `v[i]` e `idx_score` incrementato). Se tale valore non esiste (quindi tutti i valori sono `<k`), allora il vettore è già una partizione rispetto a `k`.
- la variabile `i` scorre il vettore finché non trova un valore `v[i]<k` da scambiare col valore contenuto in `v[idx_score]`.

Ricordando ciò che abbiamo detto prima riguardo il rapporto tra partizioni e array ordinati, si potrebbe pensare al seguente algoritmo di ordinamento:

```
In [15]: // un possibile algoritmo di ordinamento usando l'algoritmo di partizione

#include <stdio.h>

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

void swap(int* a, int* b)
{
    int tmp = *a;
    *a      = *b;
    *b      = tmp;
}

void partition(int v[], int n, int idx_k)
{
    // scambio l'ultimo elemento con quello in posizione idx_k
    swap(&v[idx_k], &v[n-1]); // adesso l'ultimo elemento conterrà il valore k

    int idx_store = 0; // indice del primo elemento >= di k. Ipotizzo sia 0
    for(int i = 0; i < n-1; i++)
    {
        if (v[i] < v[n-1])
        {
            swap(&v[i], &v[idx_store]);
            idx_store++; // dopo lo scambio, v[idx_store] sarà < di k,
                           // quindi si ipotizza che il successivo sia >=k
        }
    }

    // infine, metto il valore k nella posizione corretta
    swap(&v[n-1], &v[idx_store]);
}

void sort(int v[], int n)
{
    for(int i = 0; i < n; i++)
        partition(v, n, i);
}

int main()
{
    int v[] = {7,4,5,2,3,8,6};
    int n   = 7;
    printf("prima: \n");
    stampa_vett(v,n);
    sort(v,n);
    printf("dopo: \n");
    stampa_vett(v,n);
}
```

```
prima:  
(7, 4, 5, 2, 3, 8, 6)  
dopo:  
(2, 3, 4, 5, 6, 7, 8)
```

NON efficiente! Esistono però algoritmi migliori, in termini di tempo necessario al completamento, che sfruttano la funzione di partizionamento: **Quicksort** [non trattato in questo corso].

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Argomenti da linea di comando

Il C mette a disposizione la possibilità di inserire input *allo start del programma*. Esempio:

Supponiamo di avere un codice che, data una sequenza di numeri, vogliamo che restituisca il massimo. Solitamente, per eseguire questo programma, dobbiamo:

- (ovviamente compilarlo)
- eseguirlo
- attendere che il programma ci chieda i numeri da inserire
- inserire i numeri
- attendere che il programma ci comunichi il massimo

esempio di codice, file `massimo.c` :

```
#include <stdio.h>
#define MAX_N 100
int main()
{
    int v[MAX_N];
    int n;

    printf("quanti valori? ");
    scanf("%d", &n);

    printf("inserisci i valori:\n");
    for(int i = 0; i < n; i++)
        scanf("%d", &v[i]);

    int m = v[0];
    for(int i = 1; i < n; i++)
        if(v[i] > m)
            m = v[i];

    printf("il massimo è %d\n", m);

    return 0;
}
```

l'esecuzione di questo programma prevede i seguenti comandi:

```
gcc massimo.c
./a.out
quanti valori? 3
inserisci i valori:
4
1
5
il massimo è 5
```

Sarebbe più rapido, per l'utente, inserire i valori all'atto stesso dell'invocazione del programma, ossia:

```
gcc massimo.c
./a.out 4 1 5
il massimo è 5
```

Questo è possibile attraverso gli **argomenti da linea di comando**.

E' possibile infatti definire il `main(...)` in due modi:

- definizione classica: `int main()`
- definizione con parametri: `int main(int argc, char** argv)`

La seconda definizione prevede quindi 2 parametri che **devono** essere rispettivamente di tipo `int` e `char**`, e solitamente chiamati `argc` ed `argv` rispettivamente.

- il primo parametro, `argc`, contiene *quanti* argomenti sono stati inseriti all'atto dell'invocazione, compreso il nome del programma (che quindi viene visto come argomento)
- il secondo parametro, `argv`, contiene *quali* argomenti sono stati inseriti all'atto dell'invocazione, compreso il nome del programma (che quindi viene visto come argomento), sottoforma di stringhe.

Esempio, file main_args.c :

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("argomenti:\n");
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);

    return 0;
}
```

Esecuzione:

```
gcc main_args.c
./a.out pippo pluto paperino
argomenti:
./a.out
pippo
pluto
paperino
```

Da notare che

- c'è sempre almeno un argomento $\Rightarrow \text{argc} \geq 1$ che è il nome del programma. Tale argomento è contenuto in `argv[0]`. In altri termini, l'esecuzione `./a.out` avrebbe prodotto come output argomenti: `./a.out`
- gli argomenti sono sempre stringhe (NB: DA GESTIRE!)

```
#include <stdio.h>
int main(int argc, char** argv)
{

    int m = argv[1]; // ERRORE
    for(int i = 2; i < argc; i++)
        if(argv[i] > m) // ERRORE
            m = argv[i]; // ERRORE

    printf("il massimo è %d\n", m);

    return 0;
}
```

Questo programma è errato in quanto gli argomenti di `argv` sono stringhe, e come tali devono essere trattati.

Ad esempio, se ho la certezza che i miei input sono tutti numeri interi, posso utilizzare la funzione `atoi(...)` (dichiarata in `stdlib.h`) per convertire una stringa in intero.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv)
{

    int m = atoi(argv[1]);
    for(int i = 2; i < argc; i++)
    {
        int num = atoi(argv[i]);
        if(num > m)
            m = num;
    }
    printf("il massimo è %d\n", m);

    return 0;
}
```

Posso quindi:

```
gcc massimo.c
./a.out 1 3 5
il massimo è 5
```

altre funzioni simili per la conversione di stringhe in numeri:

- `atol(...)`
- `atoll(...)`
- `atof(...)`
- ...

Domanda: In quale segmento di memoria sono memorizzati gli argomenti da linea di comando?

Risposta: Lo standard non lo specifica, quindi dipende dall'implementazione. Generalmente, vengono memorizzati nello stack di esecuzione.

Domanda: è possibile inserire stringhe contenenti spazi come argomento unico? **Risposta:** Sì, utilizzando i doppi apici " . . . "

Esempio:

```
gcc main_args.c
./a.out pippo pluto paperino "mickey mouse"
argomenti:
./a.out
pippo
pluto
paperino
mickey mouse
```

In []:

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Funzioni che invocano altre funzioni

Una funzione può a sua volta essere composta dall'invocazione di più funzioni.

Esempio: *dati 3 valori, calcolarne la media aritmetica*

Posso vedere la media come la composizione di due funzioni: una funzione che effettua la somma ed un'altra che effettua la divisione

funzioni ausiliarie	main()
<pre>float somma(float a, float b, float c) { float s = a + b + c; return s; } float media(float a, float b, float c) { float s = somma(a,b,c); float d = s / 3; return d; }</pre>	<pre>int main() { float a = 5; float b = 3; float c = 4; float s = media(a,b,c); printf("la media e' %f\n", s); }</pre>

cosa succederebbe se definissi prima `media(...)` e dopo `

`somma(...)`?`

```
In [1]: # include <stdio.h>
float media(float a, float b, float c)
{
    float s = somma(a,b,c); // invocazione funzione
                            // non ancora dichiarata
    float d = s / 3;
    return d;
}

float somma(float a, float b, float c)
{
    float s = a + b + c;
    return s;
}

int main()
{
    float a = 5;
    float b = 3;
    float c = 4;
    float s = media(a,b,c);
    printf("la media e' %f\n", s);
}

/tmp/tmpikc4a6wl.c: In function 'media':
/tmp/tmpikc4a6wl.c:4:15: warning: implicit declaration of function 'somma' [-Wimplicit-function-declaration]
  4 |     float s = somma(a,b,c); // invocazione funzione
     |             ^
/tmp/tmpikc4a6wl.c: At top level:
/tmp/tmpikc4a6wl.c:11:7: error: conflicting types for 'somma'
 11 |     float somma(float a, float b, float c)
     |             ^
/tmp/tmpikc4a6wl.c:4:15: note: previous implicit declaration of 'somma' was here
  4 |     float s = somma(a,b,c); // invocazione funzione
     |             ^
[C kernel] GCC exited with code 1, the executable will not be executed
```

Il compilatore compila il codice sequenziale sfruttando i simboli dichirati *in precedenza*. Durante la compilazione della funzione

`media(...)`, la *dichiarazione* del nome simbolico `somma(...)` non è stata ancora affettuata. Il compilatore non "sa" quindi della sua esistenza, non permettendo la corretta compilazione.

⇒
⇒

separare la *dichiarazione* (chi è) del metodo dalla sua *definizione* (cosa fa)

In [25]:

```
#include <stdio.h>

// Dichiara funzioni media(...) e somma...
float media(float, float, float);
float somma(float, float, float);
//
int main()
{
    float a = 5;
    float b = 3;
    float c = 4;
    float s = media(a,b,c);
    printf("la media e' %f\n", s);

}

// Definisce funzioni media(...) e somma...
float media(float a, float b, float c)
{
    float s = somma(a,b,c);
    float d = s / 3;
    return d;
}

float somma(float a, float b, float c)
{
    float s = a + b + c;
    return s;
}
```

la media e' 4.000000

Prototipi

`float media(float, float, float); // prototipo della funzione media`

e

`float somma(float, float, float); // prototipo della funzione somma`

definiscono le caratteristiche formali delle singole funzioni, ossia:

- il nome simbolico
- quanti e quali tipi di parametri accettano in ingresso
- e quale tipo di valore restituiscono in uscita

Tali informazioni sono sufficienti al compilatore per compilare eventuali invocazioni della funzione

e.g., `float s = somma(a,b,c);`

verificando che **formalmente** sia tutto corretto

e.g., i tipi dei parametri attuali `a, b, c` sono compatibili con i tipi dei parametri formali di `somma(...)`? Il tipo restituito è compatibile con l'espressione `float s = ...`?

Spetterà poi al *linker* associare ogni invocazione alla *definizione* corretta (*binding*).

Nota: all'interno dei prototipi, i nomi dei parametri formali non sono necessari. La loro presenza è facoltativa e non deve necessariamente corrispondere ai nomi reali dei parametri formali effettivamente usati nella definizione

e.g., `float media(float, float, float);`

≡ `float media(float a, float b, float c);`

≡ `float media(float pippo, float pluto, float dartagnan);`

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Funzioni e soluzioni ricorsive

Qualche dettaglio in più sulle invocazioni di funzioni

```
In [ ]: # include <stdio.h>

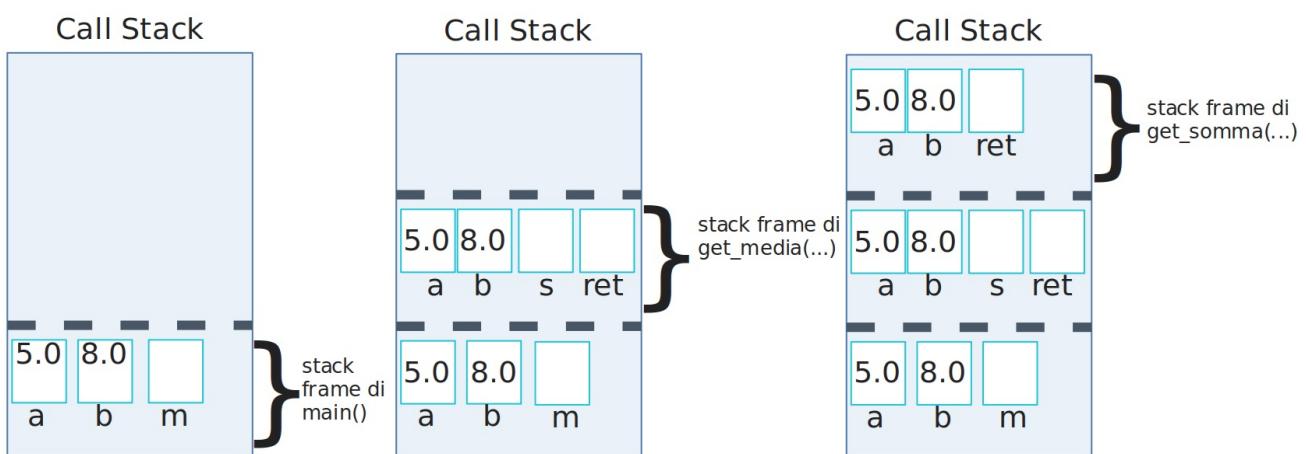
double get_somma(double a, double b)
{
    double ret = a + b;
    return ret;
}

double get_media(double a, double b)
{
    double s;
    double ret;
    s      = get_somma(a, b);
    ret   = s/2.0;
    return ret;
}

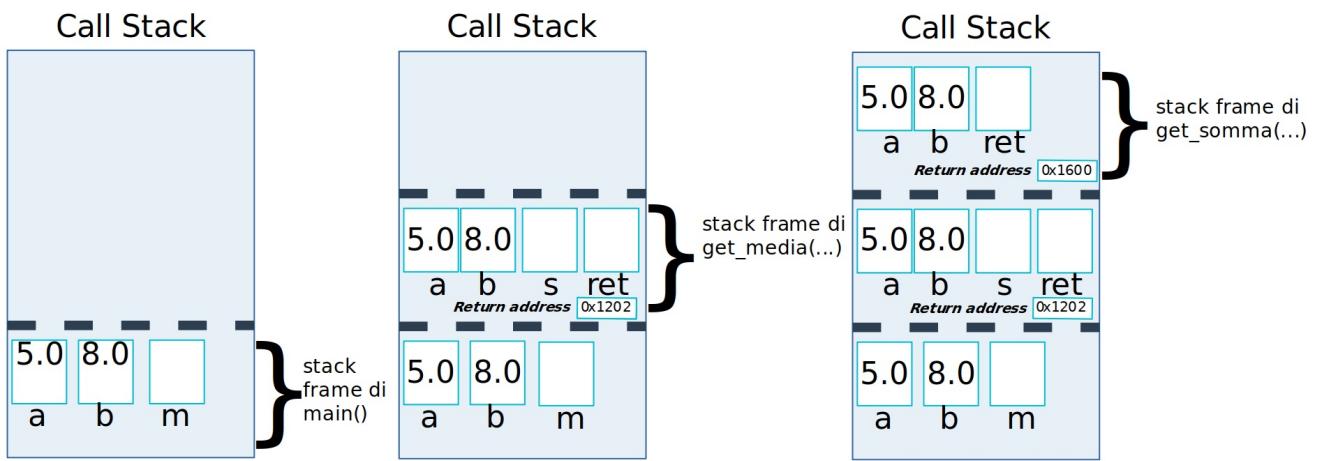
int main()
{
    double a = 5;
    double b = 8;
    double m;
    m      = get_media(a,b);
    printf("la media tra %.2lf e %.2lf è %.2lf\n",a,b,m);

    return 0;
}
```

All'esecuzione di questo programma, il call stack al suo apice sarà rappresentato nel modo seguente:



In realtà questa rappresentazione è semplificata, in quanto ogni invocazione di funzione, oltre ai parametri attuali, memorizza nello stack anche il *return address* della funzione (assieme ad altre informazioni non direttamente accessibili).



Attraverso il return address, il programma può tornare al punto esatto in cui si era interrotto e proseguire l'esecuzione dell'invocante. In altri termini, data una invocazione di funzione, il suo return address è l'indirizzo, nella funzione invocante, dell'istruzione immediatamente successiva all'invocazione.

Nel esempio di sopra, `0x1202` sarà l'indirizzo dell'istruzione immediatamente successiva all'invocazione `get_media(a, b)`; nella funzione `main(...)`, mentre `0x1600` sarà l'indirizzo dell'istruzione immediatamente successiva all'invocazione `get_somma(a, b)`; nella funzione `get_media(a, b)`;

Ovviamente, termine di ogni funzione l'intero stack frame verrà eliminato, compreso l'indirizzo di ritorno.

⇒

⇒

ogni invocazione di funzione, anche se non porta parametri, costa in termini di spazio sul call stack.

Funzione ricorsiva

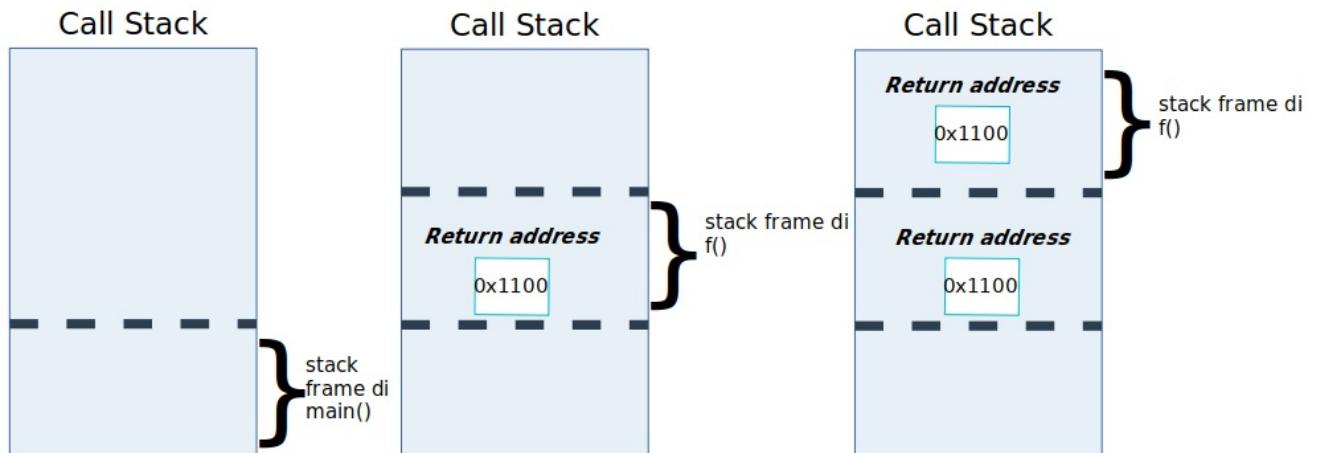
Funzione ricorsiva (programmazione): strumento messo a disposizione da tanti (ma non tutti) i linguaggi di programmazione (e.g., FORTRAN 77 non supporta la ricorsione) che permette ad una funzione di "invocare se stessa"

Esempio:

```
In [ ]: #include <stdio.h>

void f()
{
    printf("ciao\n");
    f();
}

int main()
{
    f();
    return 0;
}
```



Nel caso precedente, la funzione `f()` invoca se stessa (teoricamente) all'infinito. Ad ogni invocazione, la funzione invocante:

- salva nello stack di esecuzione il punto in cui si trova (così come qualsiasi altra funzione)
- invoca sé stessa (quindi ricomincia dall'inizio)
- una volta terminata l'esecuzione, ritorna al punto in cui l'invocante si era interrotto

⇒

⇒

necessario un criterio di stop!

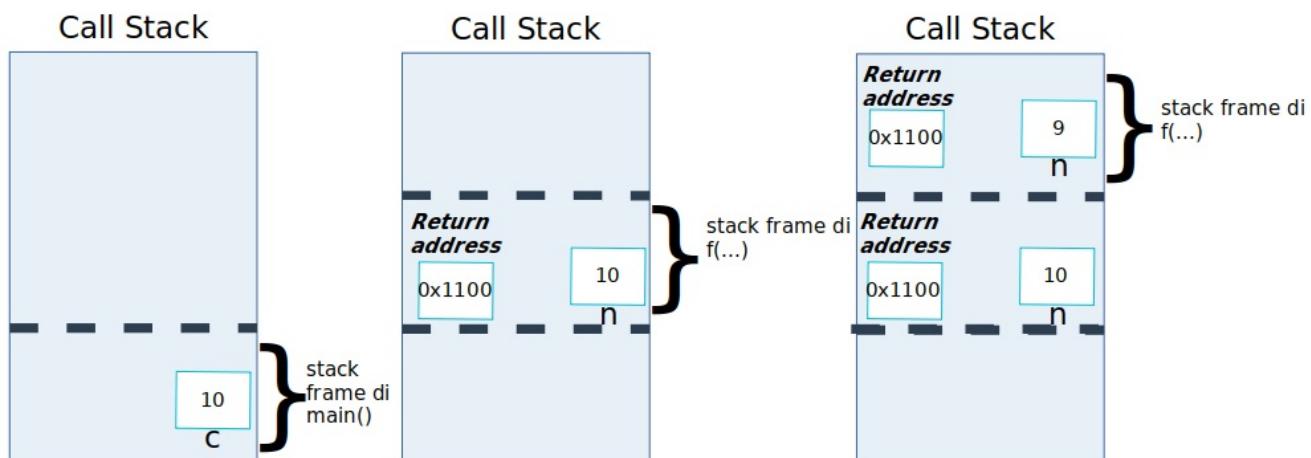
In [1]:

```
#include <stdio.h>

void f(int n)
{
    if(n > 0)
    {
        printf("ciao\n");
        f(n-1);
    }
}

int main()
{
    int c = 2;
    f(c);
    return 0;
}
```

ciao
ciao



In [8]:

```
#include <stdio.h>

void f(int n)
{
    while(n > 0)
    {
        printf("ciao\n");
        n--;
    }
}

int main()
{
    f(10);
    return 0;
}
```

ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao
ciao

A livello logico, il codice ricorsivo ed il codice iterativo sopra riportati sono equivalenti. La differenza sostanziale è che **il codice ricorsivo impegnerà più risorse**, in quanto, ad ogni invocazione, occuperà spazio nel call stack attraverso un nuovo stack frame.

Piccola parentesi:

- Da notare che anche il `main` è una funzione...quindi niente vieta di...

In []:

```
#include <stdio.h>
int main()
{
    printf("ciao\n");
```

```
    main();
}
```

Teoricamente, ogni codice iterativo può essere convertito in codice ricorsivo e viceversa.

Esempio:

Problema: Data la seguente funzione per la ricerca del massimo in un vettore

```
int max_array(int v[], int n)
{
    int max = v[0];
    for(int i = 1; i < n; i++)
        if(v[i] > max)
            max = v[i];
    return max;
}
```

Definirne una versione ricorsiva

```
In [17]: // ricerca del massimo in un vettore (funzione ricorsiva)
#include <stdio.h>
#define N 10

void max_array(int v[], int n, int i, int* max)
{
    if(i==n)
        return;
    if(i==0)
        *max = v[0];

    if(v[i] > *max)
        *max = v[i];
    max_array(v, n, i+1, max);
}

int main()
{
    int v[] = {1,5,7,2,9,4,2};
    int n = 7;
    int max;
    max_array(v,n,0, &max);
    printf("il massimo è %d\n", max);
    return 0;
}

il massimo è 9
```

```
In [16]: // ricerca del massimo in un vettore (funzione ricorsiva, alternativa)
#include <stdio.h>
#define N 10

int max_array(int v[], int n, int i, int max)
{
    if(i==n)
        return max;
    if(i==0)
        max = v[0];

    if( v[i] > max)
        max = v[i];
    return max_array(v, n, i+1, max);
}

int main()
{
    int v[] = {1,5,7,2,9,4,2};
    int n = 7;
    int max;
    max = max_array(v,n,0, 0);
    printf("il massimo è %d\n", max);
    return 0;
}

il massimo è 9
```

I due programmi di cui sopra sono logicamente equivalenti alla versione iterativa in quanto producono le stesse soluzioni.

Da notare che, ad ogni invocazione, la funzione invocante deve inviare tutte le variabili di cui necessita l'invocata, ossia:

- dati che specificano su quale/i elemento/i lavorare (in questo caso, variabile `i`)
- dati necessari per effettuare i calcoli (in questo caso, `max`)

- dati per determinare se devono essere effettuate altre autoinvocazioni oppure no (in questo caso, i ed n)

Problema: Scrivere una funzione ricorsiva per il calcolo del fattoriale di un numero.

Si ricorda che il fattoriale $n!$

$$! \\ \text{di un numero } n$$

è

- $n! = 1$
 $! = 1$
 se $n = 0$
 $= 0$
 $,$
- altrimenti, $n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$
 $! = \prod_{=1}^n = 1 \cdot 2 \cdot 3 \cdot \dots$

Una possibile soluzione iterativa al calcolo del fattoriale in un numero n

è la seguente:

```
int fattoriale(int n)
{
    int i    = 1;
    int fatt = 1;
    while(i <= n)
    {
        fatt = fatt * i;
        i++;
    }
    return fatt;
}
```

una possibile conversione della funzione precedente in funzione ricorsiva è la seguente:

```
In [26]: // Calcolo del fattoriale di un numero (funzione ricorsiva)

#include <stdio.h>
#define N 10

int fattoriale(int n, int i, int fatt)
{
    if(i > n)
        return fatt;

    return fattoriale(n, i+1, fatt*i);
}

int main()
{
    int n    = 5;
    int fatt;
    fatt= fattoriale(n,1,1);
    printf("il fattoriale è %d\n", fatt);
    return 0;
}
```

il fattoriale è 120

- ogni funzione ricorsiva deve disporre di almeno due sezioni di codice:
 - sezione dedicata al caso base: necessaria per la terminazione della funzione
 - sezione dedicata al caso ricorsivo: dove materialmente viene re-invocata la funzione
- In generale, ogni frammento di codice iterativo può essere convertito in una funzione ricorsiva, e viceversa.
- La trasformazione da iterativo a ricorsivo non sempre è intuitiva, e spesso necessita di strutture dati di appoggio. Ad esempio in certi casi è necessaria un'area di memoria con politica di accesso Pila che simuli l'execution stack)

Funzione Ricorsiva Vs. Soluzione Ricorsiva

Funzione ricorsiva (programmazione): strumento messo a disposizione da tanti (ma non tutti) i linguaggi di programmazione (e.g., FORTRAN 77 non supporta la ricorsione) che permette ad una funzione di "invocare se stessa"

E' importante distinguere tra una funzione ricorsiva ed una soluzione ricorsiva:

Soluzione ricorsiva: soluzione ad un problema che sfrutta una possibile suddivisione di quest'ultimo in sottoproblemi più semplici da risolvere (generalmente di dimensione ridotta)

NB: L'utilizzo di una funzione ricorsiva non implica che si stia fornendo una soluzione ricorsiva

COROLLARIO: Se vi si chiede una soluzione ricorsiva ad un problema e voi presentate un funzione ricorsiva che non implementa una soluzione ricorsiva, non state presentando quanto richiesto.

Esempio: voglio una soluzione ricorsiva al calcolo del fattoriale di un numero n

Per costruire una soluzione ricorsiva ad un problema una strategia può essere quella di porsi le seguenti domande:

1. esistono dei casi in cui il mio problema è facilmente risolvibile (ossia che posso considerare banale)?
2. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?
3. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?
4. posso generalizzare per i casi rimanenti?

Nel caso specifico, esistono dei valori di n

per cui il calcolo del fattoriale è particolarmente semplice da calcolare?

La risposta è sì: sappiamo infatti che per $n = 1$

$$= 1$$

(ed anche per $n = 0$

$$= 0$$

) il calcolo del fattoriale è banale, ossia che $0! = 1$

$$0! = 1$$

$$\text{ed } 1! = 1$$

$$1! = 1$$

In realtà, sapendo che $0! = 1$

$$0! = 1$$

, posso anche scrivere $1! = 1 \cdot 0!$

$$1! = 1 \cdot 0!$$

. Quindi, il caso $1!$

$$1!$$

può essere risolto sfruttando la soluzione di $0!$

$$0!$$

1. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un problema un po' più complesso?

vediamo: se $n = 2$

$$= 2$$

il fattoriale sarà $2! = 2 \cdot 1 = 2$

$$2! = 2 \cdot 1 = 2$$

$$\Rightarrow 2! = 2 \cdot 1 = 2 \cdot (2 - 1)!$$

$$\Rightarrow 2! = 2 \cdot 1 = 2 \cdot (2 - 1)!$$

Quindi ho utilizzato la soluzione $1!$

$$1!$$

per risolvere il problema $2!$

$$2!$$

1. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

vediamo: se $n = 3$

$$= 3$$

il fattoriale sarà $3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2! = 3 \cdot (3 - 1)!$

$$3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2! = 3 \cdot (3 - 1)!$$

Quindi ho utilizzato la soluzione $2!$ per risolvere il problema $3!$

se $n = 4$ il fattoriale sarà $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3! = 4 \cdot (4 - 1)!$.

1. posso generalizzare?

In generale,

$$\prod_i^n i = n \cdot \prod_i^{n-1} i$$

quindi

$$\forall n \geq 1, n! = n \cdot (n-1)!$$

Posso quindi riscrivere il fattoriale di un numero come:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Tale soluzione è implementabile in maniera (quasi) immediata attraverso una funzione ricorsiva:

```
int fattoriale(int n)
{
    if(n == 0)
        return 1;
    return n * fattoriale(n-1);
}
```

```
In [1]: // Calcolo del fattoriale di un numero (soluzione ricorsiva)
#include <stdio.h>
#define N 10

int fattoriale(int n)
{
    if(n == 0)
        return 1;
    return n * fattoriale(n-1);
}

int main()
{
    int n = 5;
    int fatt;
    fatt = fattoriale(n);
    printf("il fattoriale di %d è %d\n", n, fatt);
    return 0;
}
```

il fattoriale di 5 è 120

Da notare che la funzione, ad ogni invocazione, *necessita soltanto dei dati su cui lavorare, che saranno di volta in volta più semplici* (in questo caso, l'unico dato su cui la funzione dovrà lavorare sarà la variabile `n` che diventa sempre più piccola).

Problema: Dato un insieme di numeri S , determinarne il massimo. Proporre una *soluzione* ricorsiva

soluzione iterativa: scorro attraverso un ciclo gli elementi del mio insieme, confrontando ogni elemento con il massimo locale trovato fino a quel momento.

implementazione iterativa:

```
int A[] = {7, 6, 8, 3}; //NB: in generale, si ricorda che un vettore non è un insieme
int n    = 4;
int m    = A[0];
for(int i=1; i < n; i=i+1)
{
    if(A[i] > m)
        m = A[i];
}
```

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

Il problema è cercare il massimo in un insieme di numeri, ossia si vuole trovare una funzione $\max_set(S)$ che restituisca il più grande da un insieme di numeri S dato in input.

Intuitivamente, più l'insieme S è piccolo, più il problema è semplice.

In particolare, il problema risulta particolarmente semplice se $|S| = 1$ oppure $|S| = 2$.

- se S ha un solo elemento, i.e. $S = \{s\}$, allora il più grande è banalmente s . Posso dire che:

se $S = \{s\}$, $\max_set(S) = s$

- se S ha due elementi, i.e. $S = \{a, b\}$, allora il più grande elemento nell'insieme è il massimo tra i due elementi, che posso calcolare facilmente.

se $S = \{a, b\}$, $\max_set(S) = \max(a, b)$

con $\max(\cdot, \cdot)$ funzione di appoggio definita come $\max(a, b) = \begin{cases} a & \text{se } a \geq b \\ b & \text{altrimenti} \end{cases}$

NB. Ho quindi due funzioni:

I. $\max(\cdot, \cdot)$, che accetta in input due *numeri* e ne restituisce il massimo

II. $\max_set(\cdot)$, che accetta in input un *insieme di numeri*

1. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Il caso immediatamente più complesso è quello avente $|S| = 3$. Dato che so risolvere il problema per $|S| = 1$ ed $|S| = 2$, posso sfruttare queste soluzioni per risolvere il caso $|S| = 3$?

Il massimo tra 3 numeri S lo posso vedere come il massimo tra due numeri qualsiasi in S ed il numero rimanente, ossia:

$$\text{se } S = \{a, b, c\}, \max_set(S) = \max(c, \max(a, b))$$

Dato che so che $\max(a, b) = \max_set(\{a, b\})$,

possiamo quindi riscrivere la soluzione al problema come:

$$\text{se } S = \{a, b, c\}, \max_set(S) = \max(c, \max_set(\{a, b\}))$$

1. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

se $|S| = 4$, posso ripetere il ragionamento di prima e sfruttare la soluzione avuta con $|S| = 3$, quindi

$$\text{se } S = \{a, b, c, d\}, \max_set(S) = \max(d, \max_set(\{a, b, c\}))$$

1. posso generalizzare?

In generale,

$$\forall |S| > 2, \max_set(S) = \max(s \in S, \max_set(S \setminus \{s\}))$$

Posso quindi definire la seguente funzione ricorsiva:

$$\max_set(S = \{s_1, s_2, \dots, s_n\}) = \begin{cases} s, & \text{se } S = \{s\} \text{ (i.e. } |S|=1) \\ \max(s_1, s_2), & \text{se } S = \{s_1, s_2\} \\ \max(s \in S, \max_set(S \setminus \{s\})), & \text{se } |S| > 2 \end{cases}$$

Che può essere ulteriormente semplificata come:

$$\max_set(S = \{s_1, s_2, \dots, s_n\}) = \begin{cases} s, & \text{se } S = \{s\} \text{ (i.e. } |S|=1) \\ \max(s \in S, \max_set(S \setminus \{s\})), & \text{se } |S| \geq 2 \end{cases}$$

- (Un') Implementazione ricorsiva:

```
int max(int a, int b)
{
    if (a >= b)
        return a;
    return b;
}

int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
    // invoco max su S con un elemento in meno (per semplicità, tolgo quello in posizione 0).
    // invoco quindi la funzione passandogli (l'indirizzo del) vettore a partire dal secondo
    // elemento
    //int m = max_set(&S[1], n-1); // <=> max_set(S+1, n-1);

    //return max(S[0], m);
    return max(S[0], max_set(S+1, n-1));
}
```

Posso anche pensare di "inglobare" la funzione `max(...)` all'interno della funzione `max_set(...)`, ottenendo la seguente implementazione alternativa:

```
int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
    int m = max_set(&S[1], n-1); // <=> max(S+1, n-1);
    if (m > S[0])
        return m;
    return S[0];
```

```
}
```

```
In [31]: #include <stdio.h>
int max(int a,int b)
{
    if (a >= b)
        return a;
    return b;
}

int max_set(int v[], int n)
{
    if (n == 1)
        return v[0];
    return max(v[0], max_set(v+1, n-1));
}

int main()
{
    int v[] = {1,3,2,-1,5,7,4,2,3};
    int n = 9;
    printf("il massimo è %d\n", max_set(v,n));
}
```

```
il massimo è 7
```

```
In [29]: #include <stdio.h>
int max_set(int S[], int n)
{
    if (n == 1)
        return S[0];
    int m = max_set(&S[1], n-1); // <=> max(S+1, n-1);
    // invoco max su S con un elemento in meno (per semplicità, tolgo quello in posizione 0).
    // Passo quindi alla funzione (l'indirizzo del) vettore a partire dal secondo elemento
    if (m > S[0])
        return m;
    return S[0];
}

int main()
{
    int S[] = {7,6,8,3};
    int m = max_set(S, 4);
    printf("il massimo è %d\n",m);
    return 0;
}
```

```
il massimo è 8
```

Ricerca del massimo in un insieme: un altro approccio ricorsivo

Posso pensare di dividere la ricerca del massimo in un insieme come la ricerca ricorsiva dei massimi locali su due metà dell'insieme, e prendere quindi il valore più grande tra i due.

In altri termini:

- partiziono l'insieme in due metà
- calcolo il massimo m_1 della prima metà
- calcolo il massimo m_2 della seconda metà
- calcolo il massimo tra m_1 ed m_2

Per comodità, vedo l'insieme come un array v ed il numero di elementi che lo compone com $\#v$.

Posso quindi definire la funzione massimo in questo modo:

$$max_set(v) = \begin{cases} v_1 & \text{se } \#v = 1, \\ \max(max_set(v_1, \dots, \lfloor \frac{\#v}{2} \rfloor), max_set(v_{\lfloor \frac{\#v}{2} \rfloor + 1}, \dots, \#v)) & \text{se } \#v > 1 \end{cases}$$

In questo modo, ho *due* invocazioni ricorsive nella stessa funzione.

```
In [32]: #include <stdio.h>
int max_set(int v[], int n)
{
    if(n == 1)
        return v[0];

    // cerco il max sulla prima metà di v
    int m1 = max_set(v, n/2);
    // cerco il max sulla seconda metà di v (più eventuale elemento rimanente)
    int m2 = max_set(v+n/2, n/2+n%2);
```

```

    if (m1 >= m2)
        return m1;
    return m2;
}

int main()
{
    int v[] = {7,6,8,3,9,11,2,4,13,21,24,1,3};
    int m = max_set(v, 13);
    printf("il massimo è %d\n",m);
    return 0;
}

```

il massimo è 24

In generale, dato un problema P , un insieme di dati D_t con t "indice di complessità" dell'insieme di dati (i.e., $t+1 > t \Rightarrow$ problema su D_{t+1} più complesso che su D_t), una soluzione S del problema P è ricorsiva se può essere definita come:

$$S(D_t) = \begin{cases} S_0 & \text{se } D_t = D_0, \\ f(S(D_j), D_t) & \text{se } D_t \neq D_0 \wedge j < t \end{cases}$$

Dove D_0 è un insieme di dati su cui il problema ha una soluzione S_0 banale/nota ed f una funzione in grado di sfruttare una soluzione su un insieme di dati più semplice e, se necessario, i dati con complessità D_t .

Repetita iuvant

L'utilizzo di una funzione ricorsiva non implica che si stia fornendo una soluzione ricorsiva

```

In [1]: // Dato un insieme di numeri S, determinarne il massimo.
// Una soluzione che sembra ricorsiva, ma che ricorsiva NON è!

#include <stdio.h>
void max(int S[], int n, int idx, int* m)
{
    if(idx < n-1)
        max(S,n,idx+1,m); /* nonostante la funzione sia ricorsiva,
                           non lavora su un sottoproblema più semplice,
                           ma sullo stesso problema.
                           Cambia solo il punto in esame attraverso idx
                           */
    if(S[idx] > *m)
        *m = S[idx];
}

int main()
{
    int S[] = {7,6,8,3};

    int m = S[0];
    max(S, 4, 0, &m);
    printf("il massimo è %d\n",m);

}

// Mera conversione di una soluzione iterativa utilizzando lo strumento ricorsivo

```

il massimo è 8

Problema: ricerca di un elemento k in un insieme (e.g. $S = \{7, 6, 2, 8, 4\}$, $k = 8$). Si desidera una funzione che restituisca un valore logico di

- *true* (e.g., 1), se $k \in S$,
- *false* (e.g., 0) altrimenti.

Soluzione iterativa: scorro l'insieme finché non trovo l'elemento selezionato. Se non lo trovo, l'elemento non è presente. Restituisco 1 se è presente, 0 altrimenti.

(Un') implementazione iterativa:

```

int S[]     = { 7,6,2,8,4 };
int n      = 5;
int k      = 8;
int trovato = 0;
for(int i = 0; i < n; i=i+1)
{
    if (S[i] == k)
    {
        trovato = 1;
        break;
    }
}

```

}

Soluzione ricorsiva:

Voglio definire una funzione $\text{find_in_set}(S, k)$, con S insieme e k valore da cercare.

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

- Se il mio insieme è vuoto, ovviamente l'elemento non è presente $\Rightarrow \forall k, \text{find_in_set}(\emptyset, k) = \text{false}$
- Se l'insieme è composto da un solo elemento, allora:

$$\text{find_in_set}(S = \{s\}, k) = \begin{cases} \text{false} & \text{se } s \neq k \\ \text{true} & \text{altrimenti.} \end{cases}$$

1. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

Se $S = \{s_1, s_2\}$, posso risolvere il problema sui due sottoinsiemi $\{s_1\}$ ed $\{s_2\}$ che, avendo cardinalità 1, so risolvere. Se almeno uno dei due mi ha dato esito *true*, allora l'elemento k è presente in S , altrimenti no.

$$\text{find_in_set}(S = \{s_1, s_2\}, k) = \text{find_in_set}(\{s_1\}, k) \vee \text{find_in_set}(\{s_2\}, k)$$

1. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

Posso sfruttare una soluzione di $|S| \leq 2$ per $|S| = 3$? Ripetendo il ragionamento precedente, posso dividere S in due sottoinsiemi:

$$\{s_1\} \text{ e } \{s_2, s_3\}$$

per ognuno dei quali so trovare la soluzione. Anche in questo caso, la soluzione del problema con $|S| = 3$ sarà data dall'*or* logico tra le soluzioni dei due sottoinsiemi.

$$\text{find_in_set}(S = \{s_1, s_2, s_3\}, k) = \text{find_in_set}(\{s_1\}, k) \vee \text{find_in_set}(\{s_2, s_3\}, k)$$

1. posso generalizzare?

$$\forall |S| > 1, \text{find_in_set}(S, k) = \text{find_in_set}(\{s \in S\}, k) \vee \text{find_in_set}(S \setminus \{s\}, k)$$

Che dà vita alla funzione ricorsiva:

$$\text{find_in_set}(S, k) = \begin{cases} \text{false}, & \text{se } S = \emptyset \vee (S = \{s\} \wedge k \neq s) \\ \text{true}, & \text{se } S = \{k\} \\ \text{find_in_set}(\{s \in S\}, k) \vee \text{find_in_set}(S \setminus \{s\}, k), & \text{altrimenti} \end{cases}$$

(Un') implementazione ricorsiva:

```
int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
    {
        if (k == S[0])
            return 1;
        return 0;
    }

    return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);
}
```

oppure più semplicemente:

```
int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;

    if (k==S[0])
        return 1;

    return find_in_set(S+1, k, n-1);
}
```

Alcuni tipi di ricorsione

In base al numero di invocazioni ricorsive che si trovano all'interno di una funzione, si distingue tra:

- ricorsione *lineare*: è sufficiente una singola chiamata ricorsiva per definire il valore restituito

```
        return find_in_set(S+1, k, n-1);
```

- ricorsione *non lineare*: sono necessarie più chiamate ricorsive per definire il valore restituito

```
        return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);
```

In [34]:

```
#include <stdio.h>

int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;

    if (k==S[0])
        return 1;

    return find_in_set(&S[1], k, n-1);
}

/*
oppure...

int find_in_set(int S[], int k, int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
    {
        if (k == S[0])
            return 1;
        return 0;
    }

    return find_in_set(&S[0], k, 1) || find_in_set(S+1, k, n-1);
}
*/
```

```
int main()
{
    int S[] = {7,6,2,8,4};
    int n   = 5;
    int k   = 9;
    int trovato = find_in_set(S,k,n);
    if(trovato == 1)
        printf("trovato\n");
    else
        printf("non trovato\n");
}
```

```
non trovato
```

Ordinamento per selezione

dato un vettore v di reali, ordinare il vettore implementando una soluzione ricorsiva che sfrutti l'ordinamento per selezione (Una) versione iterativa:

```
float idx_min_array(float v[], int n, int start)
{
    int idx_min = start;
    float min   = v[start];
    for (int i=start+1; i<n; i++)
        if (v[i] < min)
        {
            min   = v[i];
            idx_min = i;
        }
    return idx_min;
}

void selection_sort(float v[], int n)
{
    int idx_min;
    for(int i=0; i<n-1; i++) // n-1, in quanto ultimo elemento,
                                // sarà già nella posizione corretta
                                // grazie agli spostamenti precedenti
    {
        idx_min   = idx_min_array(v, n, i);
        float tmp  = v[i];
        v[i]      = v[idx_min];
        v[idx_min] = tmp;
    }
}
```

Voglio provare a trovare una soluzione ricorsiva.

1. esistono dei casi in cui il mio problema è facilmente risolvibile?

- Se \mathbf{v} ha un solo elemento, la versione ordinata del vettore è banalmente \mathbf{v} stesso. Quindi,
 - se $\#\mathbf{v} = 1$, $\text{selection_sort}(\mathbf{v}) = \mathbf{v}$
- Se \mathbf{v} ha due elementi, i.e. $\mathbf{v} = (v_1, v_2)$, la versione ordinata di \mathbf{v} sarà:
 - se $\min(v_1, v_2) = v_1$, $\mathbf{v}^{\text{sorted}} = (v_1, v_2)$
 - se $\min(v_1, v_2) = v_2$, $\mathbf{v}^{\text{sorted}} = (v_2, v_1)$.

Posso quindi definire la funzione `selection_sort` per un vettore di due elementi come:

- $\text{selection_sort}(\mathbf{v}) = (v_{\text{idx_min}}, v_j)$ con $v_{\text{idx_min}} \leq v_j$.

1. posso sfruttare le soluzioni ai casi banali individuati per costruire soluzioni ad un caso un po' più complesso?

- analizziamo il caso in cui $\mathbf{v} = (v_1, v_2, v_3)$.

In questo caso, se si vuole sfruttare il principio di base del Selection Sort (ossia la ricerca del minimo), la versione ordinata del vettore la posso esprimere come il vettore composto dalla concatenazione di:

- $v_{\text{idx_min}}$, con idx_min indice del valore minimo in \mathbf{v}
- l'ordinamento del sottoarray formato dalle 2 componenti rimanenti, che so calcolarlo.

Indicando con $\mathbf{v}^{i \neq t}$ il vettore composto da tutte le componenti di \mathbf{v} tranne la t -esima (ossia $\mathbf{v}^{i \neq t} = (v_1, v_2, \dots, v_{t-1}, v_{t+1}, \dots, v_{\#\mathbf{v}})$) posso riscrivere la funzione `selection_sort(v)` su array di dimensione 3 come:

$$\text{selection_sort}(\mathbf{v}) = \text{concat}(v_{\text{idx_min}}, \text{selection_sort}(\mathbf{v}^{i \neq \text{idx_min}}))$$

con `concat(...)` funzione che restituisce la concatenazione dei due vettori dati in input.

1. posso sfruttare la mia nuova soluzione per costruire una soluzione ad un caso ancora più complesso?

Posso ripetere il ragionamento di sopra per array di dimensione 4.

1. posso generalizzare?

Sì, posso definire quindi la funzione Selection Sort per un array di lunghezza arbitraria:

`selection_sort(v):`

$$\mathbf{v}^{\text{sorted}} = \begin{cases} v_1, & \text{se } \#\mathbf{v} = 1 \\ \text{concat}(v_{\text{idx_min}}, \text{selection_sort}(\mathbf{v}_1, \dots, \text{idx_min}-1, \text{idx_min}+1, \dots, \#\mathbf{v})), & \text{se } \#\mathbf{v} > 1 \end{cases}$$

In []:

In [3]: // soluzione NON ricorsiva con funzioni ricorsive
#include <stdio.h>
#define MAX_LEN 15

```
void selection_sort(int v[], int n, int i, int k, int tmin)
{
    if( k == n)
        return;
    if(i == n)
    {
        int t = v[k];
        v[k] = v[tmin];
        v[tmin] = t;

        k++;
        i = k;
        tmin = k;

    }
    if (v[i] < v[tmin])
    {
        tmin = i;
    }
    selection_sort(v, n, ++i, k, tmin);
}

void stampa_vett(int v[], int n)
{
    printf("(");
    for(int i = 0; i < n; i++)
    {
```

```

        printf("%2d, ",v[i]);
    }
    printf("\b\b)\n");
}

int main()
{
    int v[MAX_LEN] = {5,4,1,2,3,9,7,8,10,4,6,0,9,11,15};
    selection_sort(v,MAX_LEN,0,0,0);
    stampa_vett(v,MAX_LEN);
    return 0;
}
( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 10, 11, 15)

```

```

In [4]: // soluzione ricorsiva
#include <stdio.h>
#define MAX_LEN 15

int idx_min(int v[], int n)
{
    int idx = 0;
    for(int i=1; i < n; i++)
        if (v[i] < v[idx])
            idx = i;
    return idx;
}

void swap(int* a, int* b)
{
    int t = *a;
    *a    = *b;
    *b    = t;
}

void selection_sort(int v[], int n)
{
    if(n==0 || n==1)
        return;

    int idx = idx_min(v,n);

    swap(&v[0], &v[idx]);
    selection_sort(v+1, n-1);
}

void stampa_vett(int v[], int n)
{
    printf("(" );
    for(int i = 0; i < n; i++)
    {
        printf("%2d, ",v[i]);
    }
    printf("\b\b)\n");
}

int main()
{
    int v[MAX_LEN] = {5,4,1,2,3,9,7,8,10,4,6,0,9,11,15};
    selection_sort(v,MAX_LEN);
    stampa_vett(v,MAX_LEN);
    return 0;
}
( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 10, 11, 15)

```

Merge sort

Ricordiamo la funzione di fusione tra due vettori ordinati:

```

void fondi(int v1[], int n1,
            int v2[], int n2,
            int v_out[],int* n_out)
{
    int idx_v1    = 0;
    int idx_v2    = 0;
    int idx_v_out = 0;
    while(idx_v1 < n1 && idx_v2 < n2)
    {
        if (v1[idx_v1] < v2[idx_v2])
        {
            v_out[idx_v_out] = v1[idx_v1];
            idx_v1++;
        }
        else

```

```

    {
        v_out[idx_v_out] = v2[idx_v2];
        idx_v2++;
    }
    idx_v_out++;
}

if(idx_v1 < n1)
{
    for(int i=idx_v1; i < n1; i++)
    {
        v_out[idx_v_out] = v1[i];
        idx_v_out++;
    }
}

if(idx_v2 < n2)
{
    for(int i = idx_v2; i < n2; i++)
    {
        v_out[idx_v_out] = v2[i];
        idx_v_out++;
    }
}
*n_out = idx_v_out;
}

```

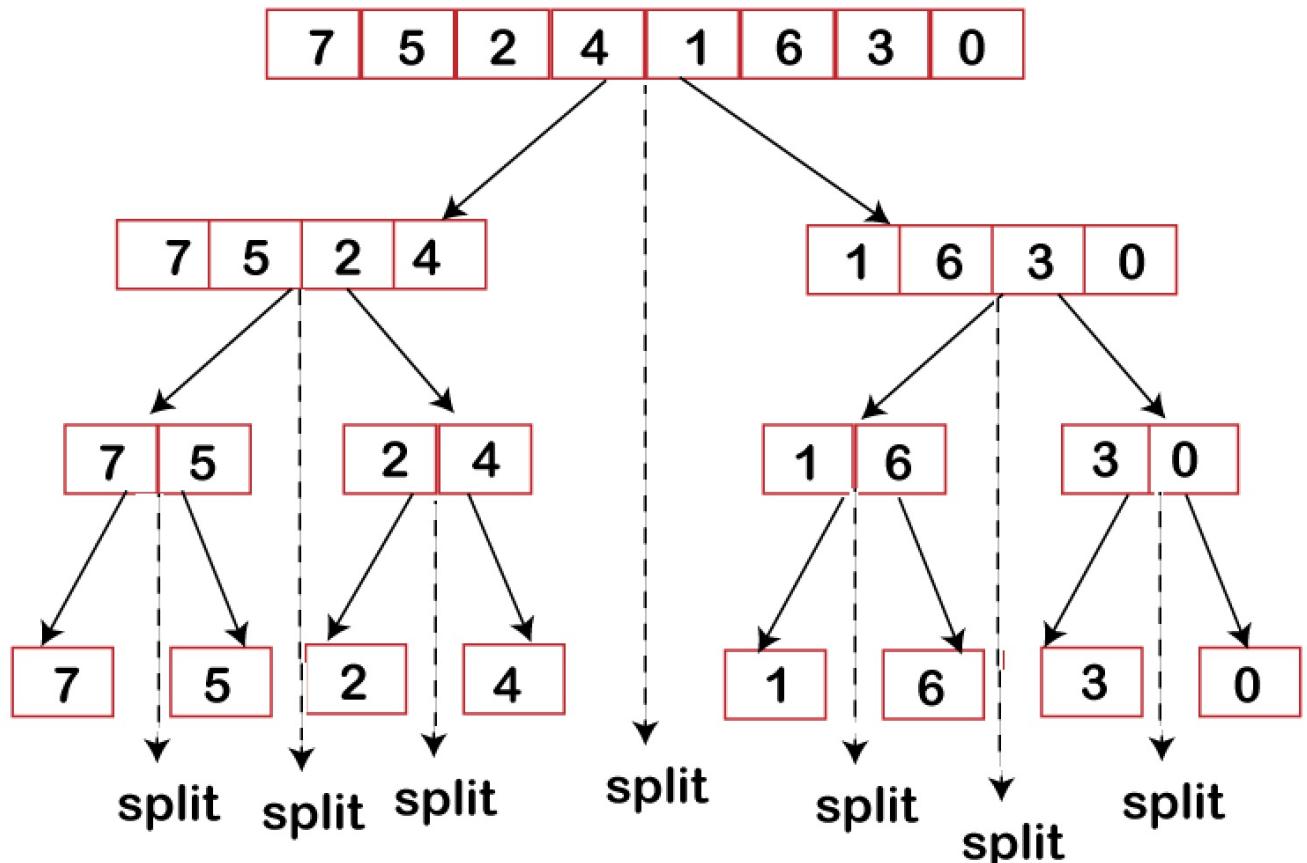
- Un vettore di un solo elemento è ordinato \implies la funzione di merge di vettori ordinati è definita anche se gli input sono due array di un solo elemento ciascuno

Princípio:

1. divido il vettore in due parti in maniera ricorsiva
2. ordino le due parti
3. fondo le due parti ordinate

mergeSort(v):

$$v_{sorted} = \begin{cases} v & \text{se } \#v = 1 \\ \text{fondi}(mergeSort(v_1 \dots \lfloor v/2 \rfloor), mergeSort(mergeSort(v_{\lfloor v/2 \rfloor +1}, \dots \#v))) & \text{altrimenti} \end{cases}$$



In [1]: #include <stdio.h>

```

#define MAX_LEN 100

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

void fondi(int v1[], int n1,
           int v2[],int n2,
           int v_out[],int* n_out)
{
    int idx_v1      = 0;
    int idx_v2      = 0;
    int idx_v_out = 0;
    while(idx_v1 < n1 && idx_v2 < n2)
    {
        if (v1[idx_v1] < v2[idx_v2])
        {
            v_out[idx_v_out] = v1[idx_v1];
            idx_v1++;
        }
        else
        {
            v_out[idx_v_out] = v2[idx_v2];
            idx_v2++;
        }
        idx_v_out++;
    }

    if(idx_v1 < n1)
    {
        for(int i=idx_v1; i < n1; i++)
        {
            v_out[idx_v_out] = v1[i];
            idx_v_out++;
        }
    }

    if(idx_v2 < n2)
    {
        for(int i = idx_v2; i < n2; i++)
        {
            v_out[idx_v_out] = v2[i];
            idx_v_out++;
        }
    }
    *n_out = idx_v_out;
}

void array_copy(int v[], int n,
               int out[])
{
    for(int i=0; i < n; i++)
        out[i] = v[i];
}

void merge_sort(int v[], int n)
{
    if (n == 1)
        return;
    merge_sort(v, n/2);
    merge_sort(v+n/2, n/2+n%2);

    int out[MAX_LEN];
    int n_out;
    fondi(v, n/2, v+n/2, n/2+n%2,
          out, &n_out);
    array_copy(out, n_out, v);
    n_out = n;
}

int main()
{
    int v[] = {10,9,8,7,6,5,4,3,2,1,0}; //{5,10,4,2,1,6,8,7,9,0,3};
    int n   = 11;
    printf("prima: ");
    stampa_vett(v, n);

    merge_sort(v, n);
}

```

```
    printf("dopo:  ");
    stampa_vett(v, n);
}

prima: ( 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0)
dopo:  ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10)
```

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

I file Header

E' buona prassi:

- inserire i prototipi in file con estensione `.h`
- inserire le definizioni delle funzioni in uno o più file `.c`

tramite la direttiva `#include` è possibile importare i *prototipi* soltanto nei file in cui le funzioni dichiarate sono materialmente utilizzate.

<code>funzioni.h</code>	<code>funzioni.c</code>	<code>principale.c</code>
<pre>float somma(float, float, float); float media(float, float, float);</pre>	<pre>#include "funzioni.h" float somma(float a, float b, float c) { float s = a + b + c; return s; } float media(float a, float b, float c) { float s = somma(a,b,c); float d = s / 3; return d; }</pre>	<pre>#include < stdio.h > #include "funzioni.h" int main() { float a = 5; float b = 3; float c = 4; float s = media(a,b,c); printf("la media e' %f\n", s); }</pre>

Dato che i file `.h` contengono solo dichiarazioni e non definizioni di funzioni, sono detti file **header** e **NON** di libreria!!!!

da notare l'utilizzo di `"..."` invece di `<...>`. In GCC, ,la differenza sta nel segnalare dove iniziare la ricerca del file di interesse.

- con le parentesi angolari `<...>` si indica che il file deve essere cercato in uno dei percorsi di default
- con i doppi apici `"..."` la ricerca inizia nella directory locale, altrimenti si cercherà negli altri percorsi

I path di default dove GCC cerca gli header sono:

```
/usr/local/include
libdir/gcc/target/version/include
/usr/target/include
/usr/include
```

Perchè la compilazione separata

- Distribuire le definizioni su più file `.c` permette di compilare le parti di un programma in maniera separata ed indipendente
- dover ricompilare una parte di programma in seguito a modifiche non implica il dover ricompilare anche le altre!

Esempio:

```
gcc -c funzioni.c → produce funzioni.o
```

```
gcc -c principale.c → produce principale.o
```

```
gcc principale.o funzioni.o -o programma → linking di principale.o e funzioni.o nell'eseguibile programma
```

Supponiamo che modifichi `principale.c` ma non `funzioni.c`:

```
gcc -c principale.c → ricompilo e sovrascrivo principale.o
```

```
gcc principale.o funzioni.o -o programma → sovrascrive eseguibile programma
```

Il nuovo eseguibile è stato generato senza la necessità di aver ricompilato `funzioni.c` che è rimasto inalterato.

Risparmio di tempo di compilazione (che per programmi molto grandi può essere anche di diverse ore)!

Cosa c'è di sbagliato in questo programma?

funzioni.h	principale.c
<pre>float somma(float, float, float); float media(float, float, float); float somma(float a, float b, float c) { float s = a + b + c; return s; } float media(float a, float b, float c) { float s = somma(a,b,c); float d = s / 3; return d; }</pre>	<pre>#include < stdio.h > #include "funzioni.h" int main() { float a = 5; float b = 3; float c = 4; float s = media(a,b,c); printf("la media e' %f\n", s); }</pre>

PESSIMA PRATICA! Definire le funzioni all'interno di un file header annulla i vantaggi della compilazione separata!

Include Guard

supponiamo di avere la seguente situazione:

structures.h	functions.h	principale.c
<pre>struct Paziente { int eta; float altezza; };</pre>	<pre>#include "structures.h" float calcolaIMC(struct Paziente);</pre>	<pre>#include < stdio.h > #include "structures.h" #include "functions.h" int main() { struct Paziente p; p.eta = 50; p.altezza = 1.70; float imc = calcolaIMC(p); printf("IMC: %f\n", imc); return 0; }</pre>

Se provo a compilare con `gcc -c principale.c`:

```
structures.h:1:8: error: redefinition of 'struct Paziente'
  1 | struct Paziente
     | ^~~~~~
In file included from principale.c:2:
structures.h:1:8: note: originally defined here
  1 | struct Paziente
     | ^~~~~~
```

Si ricorda che la direttiva `#include` viene eseguita dal *preprocessor*, prima della compilazione. Il *preprocessor* prende il contenuto del file incluso e lo inserisce nel file da compilare. Al termine della fase di preprocessing, il codice effettivo che il compilatore proverà a compilare sarà quindi diventa quindi:

```
// per praticità si è omesso il risultato di #include &lt; stdio.h &gt;

/* risultato di #include "structures.h" nel file principale.c: */
struct Paziente
{
    int eta;
    float altezza;
};

/************/

/* risultato di #include "functions.h" nel file principale.c: */

/********** risultato di #include "structures.h" nel file functions.h */
struct Paziente
{
    int eta;
    float altezza;
};
/**********/
```

```

float calcolaIMC(struct Paziente);
/*************/

int main()
{
    struct Paziente p;
    p_eta = 50;
    p_altezza = 1.70;
    float imc = calcolaIMC(p);
    printf("IMC: %f\n", imc);

    return 0;
}

```

Il fatto di includere `structures.h` sia in `principale.c` che in `functions.h` ha come risultato la `struct Paziente` viene definita due volte. In generale, dichiarazioni multiple in C sono consentite (purchè compatibili) ma **NON** sono consentite le *definizioni* multiple.

Possibili soluzioni:

1. rimuovo `#include "structures.h"` da `principale.c`, lasciando solo `#include "functions.h"`. Questo però presuppone che io sappia che `functions.h` già include `structures.h` che quindi sarà inserito "a cascata". In teoria, non sarei tenuto a sapere ogni file quali altri file include, soprattutto in un progetto di grandi dimensioni
2. utilizzo delle *include guard*.

Direttiva `ifndef`

la direttiva `#ifndef` controlla se è stata definita una costante (o più genericamente una macro attraverso la direttiva `#define`). Se non è stata definita, viene considerato il codice fino alla direttiva `#endif`, altrimenti tali righe vengono ignorate.

Esempio:

```
In [2]: #include <stdio.h>
#define MAX 10
int main()
{
    #ifndef MAX
        printf("ciao\n");
    #endif
    printf("fine\n");
}
```

ciao
fine

il codice nel "blocco" `#ifndef MAX / #endif` viene passato al compilatore solo se è definito il simbolo `MAX`.

Volendo, è possibile anche usare la direttiva `#else`.

```
In [4]: #include <stdio.h>
#define MAX 10
int main()
{
    #ifndef MAX
        printf("ciao\n");
    #else
        printf("salve\n");
    #endif
    printf("fine\n");
}
```

salve
fine

Attraverso tale direttiva posso quindi selezionare se considerare o meno per la compilazione una sezione di codice.

Da notare che, in questo caso, il valore di `MAX` non ha alcuna influenza, quindi può anche essere omesso (il controllo è fatto sul nome del simbolo).

```
In [5]: #include <stdio.h>
#define MAX
int main()
{
    #ifndef MAX
        printf("ciao\n");
    #else
        printf("salve\n");
    #endif
    printf("fine\n");
}
```

```
salve  
fine
```

Posso adottare questa direttiva, assieme alla `#define`, per specificare un meccanismo di controllo sul codice da includere

supponiamo di avere la seguente situazione:

structures.h	functions.h	principale.c
<pre>#ifndef CONTROLLO #define CONTROLLO struct Paziente { int eta; float altezza; }; #endif</pre>	<pre>#include "structures.h" float calcolaIMC(struct Paziente);</pre>	<pre>#include < stdio.h > #include "structures.h" #include "functions.h" int main() { struct Paziente p; p.eta = 50; p.altezza = 1.70; float imc = calcolaIMC(p); printf("IMC: %f\n", imc); return 0; }</pre>

Se non è stata ancora definita la macro `CONTROLLO`, definiscila e considera il codice. Se invece è già stata definita in precedenza (ad esempio da un `#include` già eseguito, non considerare quel codice (quindi materialmente non viene incluso più di una volta).

Problema: se ho più file `.h` ho bisogno di un nome diverso per ogni macro.

Soluzione: usare una qualche convenzione. Ad esempio, è prassi definire le macro delle include guard col nome del file da "proteggere"

structures.h	functions.h	principale.c
<pre>#ifndef STRUCTURES_H #define STRUCTURES_H struct Paziente { int eta; float altezza; }; #endif</pre>	<pre>#ifndef FUNCTIONS_H #define FUNCTIONS_H #include "structures.h" float calcolaIMC(struct Paziente);</pre>	<pre>#include < stdio.h > #include "structures.h" #include "functions.h" int main() { struct Paziente p; p.eta = 50; p.altezza = 1.70; float imc = calcolaIMC(p); printf("IMC: %f\n", imc); return 0; }</pre>

Esempio

f.h	f1.c	f2.c	principale.c
<pre>#ifndef F_H #define F_H void f1(); void f2(); #endif</pre>	<pre>#include < stdio.h > #include "f.h" void f1() { printf("f1 invocata.\n"); }</pre>	<pre>#include < stdio.h > #include "f.h" void f2() { printf("f2 invocata.\n"); }</pre>	<pre>#include < stdio.h > #include "f.h" int main() { f1(); f2(); return 0; }</pre>

Compilazione separata:

```
gcc -c f1.c
gcc -c f2.c
gcc -c principale.c
```

Linking:

```
gcc f1.o f2.o principale.o
```

Esecuzione:

```
./a.out
f1 invocata.
f2 invocata.
```

Attraverso il meccanismo del linking, se ho più versioni della stessa funzione (i.e., stesso prototipo), posso "scegliere" quale funzione "collegare" materialmente al programma invocante.

Esempio:

```
// file f1_alternativo.c
#include < stdio.h >
```

```
#include "f.h"
void f1()
{
    printf("f1 alternativo invocata.\n");
}
```

Compilazione:

```
gcc -c f1_alternativo.c
```

linking:

```
gcc principale.c f1_alternativo.o f2.o
```

esecuzione:

```
./a.out
f1 alternativo invocata.
f2 invocata.
```

Librerie in C

Posso raggruppare assieme più file `.o` generando una libreria.

Due tipi di libreria:

1. librerie statiche (estensione `.a`)
2. librerie a collegamento dinamico (o condivise, estensione `.so`, o `.dll` su sistemi windows) [non trattate in questo corso]

Librerie statiche:

- sono degli archivi di file `.o`.
- hanno generalmente estensione `.a`.
- Il nome inizia con la radice `lib`.

esempio: voglio generare una libreria statica di nome `libfun.a` contenente i file `f1.o` ed `f2.o`

```
ar -rc libfun.a f1.o f2.o
```

`-rc` stanno per *crea e sostituisce i file all'interno se già presenti.*

Dato un file `.a`, per verificare tale file quali file a sua volta contiene:

```
ar -t libfun.a
```

```
f1.o
f2.o
```

Per collegare il file di libreria all'eseguibile, almeno 2 modi:

a) metodo classico

```
gcc principale.o libfun.a
```

b) ho bisogno di utilizzare due specifici flag di gcc:

1. `-L` : dice *dove* (in termini di directory) cercare i/il file di libreria;
2. `-l` : specifica *quale* file di libreria usare (omettere la radice `lib` del nome del file e l'estensione)

```
gcc principale.o -L./ -l:libfun
```

oppure, se non si vuole omettere la radice `lib` o se il file ha un'altra radice

```
gcc principale.o -L./ -l:libfun.a
```

il file generato (in questo caso `a.out`) sarà identico, in termini di input/output, al precedente.

Se il file di libreria si trova in uno dei percorsi di ricerca di default di `gcc`, il flag `-L` può essere omesso.

Ad esempio, se il programma utilizza una delle funzioni i cui prototipi sono contenuti nel file header `math.h` e deve essere collegato al noto file di libreria `libm.a` (o `libm.so`), allora la sintassi sarà più semplicemente

```
gcc programma.c -lm
```

L'ordine conta!

f.h

power_fun.c

principale.c

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

```

#ifndef F_H
#define F_H
void print_powers(int n, int max_exp);
#endif

#include "f.h"
void print_powers(int n, int max_exp)
{
    for(int i=0; i <= max_exp; i++)
    {
        printf("n^i = %lf\n", pow(n, i));
    }
}

#include <stdio.h>
#include "f.h"
int main()
{
    print_powers(3,5);
    return 0;
}

```

compilazione:

```
gcc -c power_fun.c
gcc -c principale.c
```

linking:

dato che utilizzo la funzione `pow(...)`, necessito della libreria `libm.a` (oppure `libm.so`).

```
gcc main.o power_fun.o -lm
```

DOMANDA: E' lo stesso se inverti l'ordine dei flag ?

i.e.

```
gcc main.o -lm power_fun.o è lo stesso di gcc main.o power_fun.o -lm ???
```

output di `gcc main.o -lm power_fun.o`:

```
/usr/bin/ld: power_fun.o: in function `print_powers':
power_fun.c:(.text+0x26): undefined reference to `pow'
collect2: error: ld returned 1 exit status
```

Il linker GNU tiene traccia delle funzioni invocate durante il processo di linking. Se, all'atto del linking di una data libreria, ci sono funzioni nella libreria che non sono state *ancora* effettivamente invocate, tali funzioni vengono eliminate dal processo di linking. Il risultato è che, se viene effettuato il linking troppo presto, allora le funzioni in quella libreria non sono più disponibili nei file successivi.

Quindi è necessario mettere prima le librerie *dipendenti*, e poi le librerie *che soddisfano* tali dipendenze.

```
gcc main.o -lm power_fun.o
```

In questo caso, `main.o` non dipende in alcun modo da `libm`, in quanto nessuna funzione di `libm` viene direttamente invocata. Quindi, quando viene effettuato il linking con `libm`, **nessuna funzione ivi contenuta viene mantenuta nel processo**. Quando si cercherà di effettuare il linking di `power_fun.o`, la funzione `pow(...)` non verrà trovata.

invece, in

```
gcc main.o power_fun.o -lm
```

si analizza *prima* `power_fun.o`. Verrà tenuta traccia delle dipendenza non soddisfatta a `pow(...)` che verrà cercata nei file di libreria specificati *successivamente* (in questo caso `libm`).

In []:

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

Politiche di accesso alle strutture dati

Con **politica di accesso** intenderemo l'insieme di regole utilizzate per accedere ad una struttura dati (e.g., array o liste).

Solitamente, una struttura dati viene fornita con *accesso libero*, nel senso che, nel corso del programma, l'utilizzatore può accedere liberamente a *qualsiasi* porzione della struttura dati (ad esempio, in un array, è possibile accedere a qualsiasi posizione di lettura/scrittura attraverso l'operatore `[]`), quindi senza alcuna regola che me ne limiti l'accesso.

Esempio:

```
int main()
{
    int V[10];
    V[3] = 5;
    V[1] = 2;
    V[7] = 44;

    return 0;
}
```

In questo esempio, non ho nessun problema ad accedere a *qualsiasi* posizione dell'array nel corso del programma. Posso accedere alla prima come posso accedere all'ultima come posso accedere alla posizione centrale e così via in maniera libera.

Tuttavia, strutture dati con politiche di accesso "libere" come gli array non sempre sono la scelta migliore. In generale, esistono strutture dati con politiche di accesso più ristrette, ossia che seguono determinate regole. Esempi:

- Politica di accesso **Coda (Queue)**:
 - è possibile accedere *in lettura* soltanto al primo elemento della struttura dati. L'eventuale rimozione del primo elemento permetterà di leggere (ed eventualmente rimuovere) il successivo, e così via.
 - detta anche modalità di accesso *FIFO (First In, First Out)*
- Politica di accesso **Pila (Stack)**:
 - è possibile accedere *in lettura* soltanto al primo elemento della struttura dati. L'eventuale rimozione del primo elemento permetterà di leggere (ed eventualmente rimuovere) il successivo, e così via.
 - detta anche modalità di accesso *LIFO (Last In, First Out)*

Il C non mette a disposizione in maniera nativa strutture dati che implementano simili politiche di accesso.

E' però possibile *costruirle*, utilizzando ciò che abbiamo (e dandoci noi qualche regola).

CODE

In []:

```
// una prima implementazione
#include <stdio.h>
#define N_CODA 4

#define OK 0

#define ERR 1
struct Coda
{
    int deposito[N_CODA];
```

```

int idx_prima_libera;

};

void init_queue (struct Coda* c)
{
    c->idx_prima_libera = 0;
}

int insert_in_queue(struct Coda* c, int elemento)
{
    if (c->idx_prima_libera >= N_CODA)
    {
        return ERR;
    }
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera++;
    return OK;
}

int read_from_queue(struct Coda* c, int* p_val)
{
    if(c->idx_prima_libera == 0)
    {
        return ERR;
    }
    *p_val = c->deposito[0];
    return 0;
}

int get_from_queue(struct Coda* c, int* p_val)
{
    if(read_from_queue(c, p_val) != 0)
        return ERR;
    int n_elementi      = c->idx_prima_libera;
    //scorro a sinistra
    for(int i=0; i<n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }

    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}

int main()
{
    struct Coda c;
    init_queue(&c);

    int scelta = -1;
    while(scelta != 0)
    {
        printf("digitare:\n");
        printf("1) inserire in coda\n");
        printf("2) leggere dalla coda\n");
        printf("3) leggere e rimuovere dalla coda\n");
        printf("0) uscire\n");
        scanf("%d", &scelta);

        int val, err;
        switch(scelta)
        {
            case 1:
                printf("Inserisci valore: ");
                scanf("%d", &val);
                err = insert_in_queue(&c, val);
                if(err == OK)
                    printf("valore inserito\n");
                else
                    printf("coda piena!\n");
                break;

            case 2:
                err = read_from_queue(&c, &val);

                if(err == OK)
                    printf("il valore in testa è %d.\n", val);
        }
    }
}

```

```

        else
            printf("coda vuota!\n");
            break;

    case 3:
        err = get_from_queue(&c, &val);
        if(err == OK)
            printf("il valore in testa è %d ; tale valore è stato rimosso.\n", val);
        else
            printf("coda vuota!\n");
        break;

    case 0:
        printf("ciao!\n");
        break;

    default:
        printf("scelta errata!\n");
    }
}

return 0;
}

```

Una struttura dati che adotta una particolare politica di accesso deve quindi essere composta da:

- una struttura dati d'appoggio che conterrà materialmente i dati (esempio: un array)
- un insieme di **funzioni** ad-hoc che permettono di accedere alla struttura dati *rispettando le regole della politica di accesso*

L'accesso a tale struttura deve quindi essere effettuato unicamente tramite le funzioni definite. Accedervi in qualsiasi altro modo è una violazione della politica d'accesso!

Esempio:

```

int main()
{
    struct Coda c;
    c->deposito[3] = 34; /* accesso diretto alle caratteristiche interne della struttura
                           senza passare per le funzioni specifiche!
                           VIOLAZIONE POLITICA FIFO
                           */
    c->deposito[5] = 92; // VIOLAZIONE POLITICA FIFO
    printf("%d\n", c->deposito[5]); // VIOLAZIONE POLITICA FIFO
}

```

Il C non mette a disposizione dei modi per evitare l'accesso a determinati campi di una `struct`. Quindi, se conosco i dettagli della struttura dati, posso praticamente accedervi come ho appena fatto senza passare per le funzioni ad-hoc. **Ma ciò fa decadere la proprietà di essere una Coda alla struttura `c`.**

L'utilizzo delle sole funzioni ad-hoc per l'utilizzo della struttura mi garantisce che la struttura dati allocata `c` sia effettivamente una Coda, e che quindi al suo interno troverò sempre i dati disposti nella maniera regolata dalla politica scelta.

Ricapitolando, per definire una struttura dati con una certa politica d'accesso:

- definisco la struttura dati d'appoggio
- definisco le **funzioni** per accedervi

Le funzioni per accedervi possono a loro volta essere:

- fondamentali: richieste per l'accesso materiale alla struttura
- ausiliarie: non indispensabili, ma aiutano l'accesso alla struttura

Ad esempio, una migliore implementazione di una coda potrebbe prevedere le seguenti funzioni:

- fondamentali:
 - `init_queue(...)`
 - `insert_in_queue(...)`, o anche `enqueue(...)` o anche `append(...)`
 - `get_from_queue(...)`, o anche `dequeue(...)`
- ausiliarie:
 - `is_queue_empty(...)` : restituisce un valore che indica se la coda è vuota o meno
 - `is_queue_full(...)` : restituisce un valore che indica se la coda è piena o meno
 - `print_queue(...)` : stampa il contenuto attuale della coda
 - `read_from_queue(...)` : legge l'elemento in testa *ma senza eliminarlo*

In []:

```
// CODA: Una migliore implementazione
```

```

#include <stdio.h>
#define N_CODA 4
#define FULL -1
#define EMPTY -2
#define OK 0

struct Coda
{
    int deposito[N_CODA];
    int idx_prima_libera;
    int idx_prima_occupata;
    int n_occupati;

};

void init_queue (struct Coda* c)
{
    c->idx_prima_libera = 0; // prima posizione libera (ossia posizione in cui inserire)
    c->idx_prima_occupata = 0; // prima posizione occupata (ossia posizione da cui leggere)
    c->n_occupati = 0; // numero di elementi presenti
}

int is_queue_empty(struct Coda* c)
{
    if(c->n_occupati == 0)
    {
        return 1;
    }
    return 0;
}

int is_queue_full(struct Coda* c)
{
    if (c->n_occupati >= N_CODA)
    {
        return 1;
    }
    return 0;
}

int insert_in_queue(struct Coda* c, int elemento)
{
    if(is_queue_full(c) == 1)
        return FULL;
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera = (c->idx_prima_libera+1) % N_CODA;

    c->n_occupati++;
    return OK;
}

int read_from_queue(struct Coda* c, int* p_val)
{
    if(is_queue_empty(c) == 1)
        return EMPTY;

    *p_val = c->deposito[c->idx_prima_occupata];
    return OK;
}

int get_from_queue(struct Coda* c, int* p_val)
{
    if(is_queue_empty(c) == 1)
        return EMPTY;
    read_from_queue(c, p_val);
    c->idx_prima_occupata = (c->idx_prima_occupata+1) % N_CODA;
    c->n_occupati--;
    return OK;
}

int main()
{
    struct Coda c;
    init_queue(&c);

    int scelta = -1;
    while(scelta != 0)

```

```

{
    printf("digitare:\n");
    printf("1) inserire in coda\n");
    printf("2) leggere dalla coda\n");
    printf("3) leggere e rimuovere dalla coda\n");
    printf("0) uscire\n");
    scanf("%d", &scelta);

    int val, err;
    switch(scelta)
    {
        case 1:
            printf("Inserisci valore: ");
            scanf("%d", &val);
            err = insert_in_queue(&c, val);
            if(err == OK)
                printf("valore inserito\n");
            else
                printf("coda piena!\n");
            break;

        case 2:
            err = read_from_queue(&c, &val);

            if(err == OK)
                printf("il valore in testa è %d.\n", val);
            else
                printf("coda vuota!\n");
            break;

        case 3:
            err = get_from_queue(&c, &val);
            if(err == OK)
                printf("il valore in testa è %d ; tale valore è stato rimosso.\n", val);
            else
                printf("coda vuota!\n");
            break;

        case 0:
            printf("ciao!\n");
            break;

        default:
            printf("scelta errata!\n");
    }
}
}

```

In quest'ultima implementazione, non è necessario scorrere il vettore ogni volta che un elemento viene rimosso.

Tale implementazione è detta *circolare*.

PILE

Funzioni di accesso:

- fondamentali:
 - `init_stack(...)`
 - `insert_in_stack(...)`, o anche `push(...)`
 - `get_from_stack(...)`, o anche `pop(...)`
- ausiliarie:
 - `is_stack_empty(...)` : restituisce un valore che indica se la pila è vuota o meno
 - `is_stack_full(...)` : restituisce un valore che indica se la pila è piena o meno
 - `print_stack(...)` : stampa il contenuto attuale della pila
 - `read_from_stack(...)` : legge l'elemento in testa *ma senza eliminarlo*

In []:

```
// una prima implementazione ingenua
#include <stdio.h>
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
struct Pila
{
    int deposito[N_PILA];
    int n_occupati;
};

void init_stack (struct Pila* pila)
{
    pila->n_occupati = 0;
}

int pop(struct Pila* p, int* val)
{
    if(p->n_occupati <= 0)
        return EMPTY;
    *val = p->deposito[0];
    for(int i = 0; i < p->n_occupati-1; i++)
        p->deposito[i] = p->deposito[i+1];
    p->n_occupati--;
    return OK;
}

int push(struct Pila* p, int val)
{
    if(p->n_occupati >= N_PILA)
        return FULL;
    for(int i = p->n_occupati; i > 0; i--)
        p->deposito[i] = p->deposito[i-1];
    p->deposito[0] = val;
    p->n_occupati++;
    return OK;
}
```

In []:

```
// una migliore implementazione

#include <stdio.h>
#define N_PILA 4

#define FULL -1
#define EMPTY -2
#define OK 0

struct Pila
{
    int deposito[N_PILA];
    int n_occupati;
};

void init_stack (struct Pila* p)
{
    p->n_occupati = 0;
}

int pop(struct Pila* p, int* val)
{
    if(p->n_occupati <= 0)
        return EMPTY;
    *val = p->deposito[p->n_occupati-1];
    p->n_occupati--;
    return OK;
}

int push(struct Pila* p, int val)
{
    if(p->n_occupati >= N_PILA)
        return FULL;
    p->deposito[p->n_occupati] = val;
    p->n_occupati++;
    return OK;
}
```

L'utilizzo delle funzioni ad-hoc mi permette di evitare di dover conoscere (o ricordare) i dettagli implementativi della struttura.

Mi è sufficiente conoscere i prototipi delle funzioni di accesso (e.g., `push(...)` e `pop(...)`) per poter accedere alla struttura, *senza dover sapere materialmente come è implementata la mia struttura dati*. E.g., se l'implementatore ha utilizzato un array o, come vedremo, una lista concatenata, per l'utilizzatore della struttura *non ha alcuna importanza*, dato che le funzioni di accesso *mi nascondono* i dettagli implementativi, permettendo comunque l'accesso alla struttura da parte dell'utilizzatore.

Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2021/22

A. Apicella

C: Qualche consiglio pratico - 1

Funzione `scanf(...)`

caso 1

Raramente, ma può succedere:

```
# include<stdio.h>

int main(){
    int a,b;

    printf("inserisci interi:");
    scanf("%d%d",&a,&b); // <---- mancanza di spazio tra %d
    printf("i valori inseriti sono: %d %d", a, b);
    return 0;
}
```

(possibile) output:

```
inserisci interi:1
i valori inseriti sono: 1 32754
```

La mancanza di uno spazio tra `%d%d`, in alcuni compilatori, *potrebbe* dare problemi (anche se nella maggior parte dei casi non ne dà, in quanto in teoria non necessario). In generale, consiglio sempre di separare gli speciatori di formato da uno spazio, anche per una questione di leggibilità.

Versione migliore:

caso 2

```
# include<stdio.h>

int main(){
    int a,b;

    printf("inserisci interi:");
    scanf("%d %d",&a,&b); //<--- conviene lasciare uno spazio tra gli specicatori di formato in
    //caso di acquisizioni multiple
    printf("i valori inseriti sono: %d %d", a, b);
    return 0;
}
```

output:

```
inserisci interi:1
2
i valori inseriti sono: 1 2
```

Ad ogni modo, almeno all'inizio e nel caso di acquisizioni semplici, consiglio di evitare l'acquisizione di più valori utilizzando la stessa invocazione di `scanf(...)` (a meno che non sia espressamente richiesto).

```
int main(){
    int a,b;

    printf("inserisci intero:");
    scanf("%d ",&a); // <---- notare la presenza dello spazio dopo %d
```

```

    printf("inserisci intero:");
    scanf("%d ",&b); // ----- notare la presenza dello spazio dopo %d
    printf("i valori inseriti sono: %d %d\n",a, b);
    return 0;
}

```

output:

```

inserisci intero:1
2
inserisci intero:3
i valori inseriti sono: 1 2

```

Nella `scanf(...)` far seguire uno spazio ad uno specificatore di formato può creare problemi. Questo perché la `scanf(...)`, quando incontra uno spazio nella sua stringa di formato, non termina finché non viene inserito un carattere diverso da `space` o `newline`. E' necessario quindi inserire un utelriore carattere (e.g., `2`) per far terminare la `scanf(...)`. Il carattere `2` inserito per far terminare la `scanf("%d ",&a)` non viene però "consumato", rimanendo nel buffer. La seconda chiamata `scanf("%d ",&b)` "consumerà" il carattere `2`, ponendolo nella variabile `b` (al posto del carattere `3`, che rimane nel buffer, e viene utilizzato solo per far terminare la `scanf("%d ",&b)`, in quanto anch'essa contiene uno spazio dopo lo specificatore).

Lo stesso comportamento si ha facendo seguire `%d` da un `\n`.

```

int main(){
    int a,b;

    printf("inserisci intero:");
    scanf("%d\n",&a); // ----- notare la presenza del \n dopo %d

    printf("inserisci intero:");
    scanf("%d\n",&b); // ----- notare la presenza del \n dopo %d
    printf("i valori inseriti sono: %d %d\n",a, b);
    return 0;
}

```

Possibile soluzione: evitare gli spazi e `\n`.

```

int main(){
    int a,b;

    printf("inserisci intero:");
    scanf("%d",&a);

    printf("inserisci intero:");
    scanf("%d",&b);
    printf("i valori inseriti sono: %d %d\n",a, b);
    return 0;
}

```

output:

```

inserisci intero:1
inserisci intero:3
i valori inseriti sono: 1 3

```

caso 3

```

#include<stdio.h>

int main(){
    int a;

    char car;

    printf("inserisci intero:");
    scanf("%d",&a);
    printf("inserisci carattere:");
    scanf("%c",&car);
    printf("Hai inserito il carattere %c\n", car);
    return 0;
}

```

In questo caso, abbiamo un carattere acquisito dopo un numero.

Problema: l'invocazione `scanf(" %d ",&a)` acquisisce e mette nella variabile `a` il valore inserito da tastiera. Tuttavia, la conferma del valore viene data con il tasto `\n` che corrisponde ad un carattere di `\n` che però non viene consumato, aspettando la prossima acquisizione.

La successiva acquisizione è `scanf ("%c", &car);`, ossia l'acquisizione di un carattere (`%c`). Tale acquisizione troverà nel buffer il precedente carattere `\n`, considerandolo come carattere da mettere nella variabile `car`, e quindi impedendo all'utente di fare l'acquisizione effettiva.

soluzione 1:

Inserire uno spazio prima di `%c`

```
#include<stdio.h>

int main(){
    int a;

    char car;

    printf("inserisci intero:");
    scanf("%d",&a);
    printf("inserisci carattere:");
    scanf(" %c",&car); // notare il carattere spazio prima di %c
    printf("Hai inserito il carattere %c\n", car);
    return 0;
}
```

Inserendo uno spazio prima di `%c` dirà alla `scanf(...)` di ignorare eventuali spazi (o `\n`) inseriti (e quindi anche l'eventuale `\n` già presente).

soluzione 2:

- fare un'acquisizione a vuoto per consumare il carattere rimanente
 - `getchar();` oppure `getch(stdin);` nel caso di un solo carattere

```
#include<stdio.h>

int main(){
    int a;

    char car;

    printf("inserisci intero:");
    scanf("%d",&a);
    printf("inserisci carattere:");
    getchar();
    scanf("%c",&car);
    printf("Hai inserito il carattere %c\n", car);
    return 0;
}
```

soluzione 3:

Scrivere ed invocare una funzione dedicata alla pulizia del buffer di input:

```
#include<stdio.h>
void clean_stdin(void)
{
    int c;
    do
    {
        c = getchar();
    }
    while (c != '\n' && c != EOF);
}

int main(){
    int a;

    char car;

    printf("inserisci intero:");
    scanf("%d",&a);
    printf("inserisci carattere:");
    clean_stdin();
    scanf("%c",&car);
    printf("Hai inserito il carattere %c\n", car);
    return 0;
}
```

Una **non**-soluzione:

Su alcune fonti, viene riportata come soluzione l'utilizzo della funzione `fflush(stdin)`. Tuttavia, tale soluzione potrebbe avere

comportamento indefinito in alcune implementazioni quando applicata su buffer di input (come nel caso, appunto, di `stdin`). Il comportamento potrebbe quindi essere dipendente dall'ambiente di esecuzione, non garantendomi che venga effettivamente pulito il buffer (e che quindi venga risolto il problema). **Sconsigliato!**

Funzione `printf(...)`

Consideriamo il seguente codice (sbagliato):

```
#include<stdio.h>

int main(){
    int* p = NULL;
    *p    = 50;
    printf("il valore in *p è %d\n",*p);

    return 0;
}
```

Mandandolo in esecuzione avremo:

```
Errore di segmentazione (core dump creato)
```

Ovviamente la riga errata è `*p = 50`, ma supponiamo di non saperlo e che vogliamo scoprirlo. Supponiamo che, per qualche oscura ragione, decidiate di non usare il debugger per analizzare il vostro programma e scoprire dov'è il problema, preferendo riempire il vostro codice di `printf(...)` per scoprire la/le fonte/i del problema :

```
#include<stdio.h>

int main(){
    int* p = NULL;
    printf("riga 1 eseguita");
    *p    = 50;
    printf("riga 2 eseguita");
    printf("il valore in *p è %d\n",*p);
    printf("riga 3 eseguita");
    return 0;
}
```

dato che il problema è alla riga 2, mi aspetterei che, una volta in esecuzione, il programma stampi almeno `riga 1 eseguita`. Tuttavia, l'output è ancora:

```
Errore di segmentazione (core dump creato)
```

Non leggendo `riga 1 eseguita`, potrei essere portato a pensare erroneamente che l'errore sia causato dalla riga 1 (ossia `int* p = NULL;`) che, invece, non ha nulla di strano.

Questo succede perché lo standard output `stdout` è *line buffered*, ossia non consuma l'output (che rimane nel buffer di uscita senza essere consumato e quindi, in questo caso, senza essere mostrato a video) finché non viene raggiunta la fine della riga.

Soluzione 1:

aggiungere uno `\n` così che la riga venga considerata terminata

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

```
int main(){
    int* p = NULL;
    printf("riga 1 eseguita\n");
    *p    = 50;
    printf("riga 2 eseguita\n");
    printf("il valore in *p è %d\n",*p);
    printf("riga 3 eseguita\n");
    return 0;
}
```

output:

```
riga 1 eseguita
```

```
Errore di segmentazione (core dump creato)
```

Soluzione 2:

Usare la funzione `fflush(stdout)`. In questo caso la funzione `fflush(...)` va bene, in quanto nasce proprio per i flussi di output (a differenza del caso `stdin`, che è un flusso di input).

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

```
int main(){
    int* p = NULL;
    printf("riga 1 eseguita");
    fflush(stdout);
    *p    = 50;
    printf("riga 2 eseguita");
    fflush(stdout);
    printf("il valore in *p è %d\n",*p);
    printf("riga 3 eseguita");
    fflush(stdout);
    return 0;
}

riga 1 eseguita
```

(da notare che il tutto viene stampato sulla stessa riga).

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

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

Tempo di esecuzione di un algoritmo

Dato un algoritmo, vogliamo avere una stima di quanto *tempo* ci mette ad essere completato su una data macchina.

Esempio 1: ipotizziamo di avere un algoritmo in grado di effettuare la somma dei primi n numeri naturali, con n dato in input, e di voler sapere quanto tempo impiega per essere eseguito su una data macchina.

ipotizziamo che le istruzioni

- aritmetiche
- logiche
- scrittura/lettura di memoria

siano considerate elementari ed impieghino tutte lo stesso tempo (e.g. 1 ms) per essere eseguite su una data architettura.

Date queste premesse, supponiamo di voler sapere quanto tempo impiega il seguente codice per essere eseguito.

```
In [ ]: // algoritmo 1.1
int main()
{
    int n;
    printf("inserire numero: ");
    scanf("%d", &n);

    int s = 0;
    for(int i = 1; i<=n; i++)
        s = s + i;

    printf("la somma dei numeri da 1 a %d è %d\n", n, s);

    return 0;
}
```

Supponendo di inserire 5 come valore di `n`, il seguente algoritmo effettuerà:

- 2 inizializzazioni `int s = 0; int i = 1;`
- 6 confronti `i<=5`
- 5 incrementi `i++`, che possono essere visti come 5 assegnazioni + 5 operazioni aritmetiche
- 5 somme `s + i`
- 5 assegnazioni `s = ...`

totale: $2 + 6 + 2 \cdot 5 + 5 + 5 = 28$ operazioni da 1 ms, per un totale di 28 ms (Nota: solo per semplicità, abbiamo trascurato il tempo per le operazioni di lettura dalla memoria).

Generalizzando ad un n qualsiasi:

$$2 + n + 1 + 2n + n + n = 5n + 3 \text{ ms.}$$

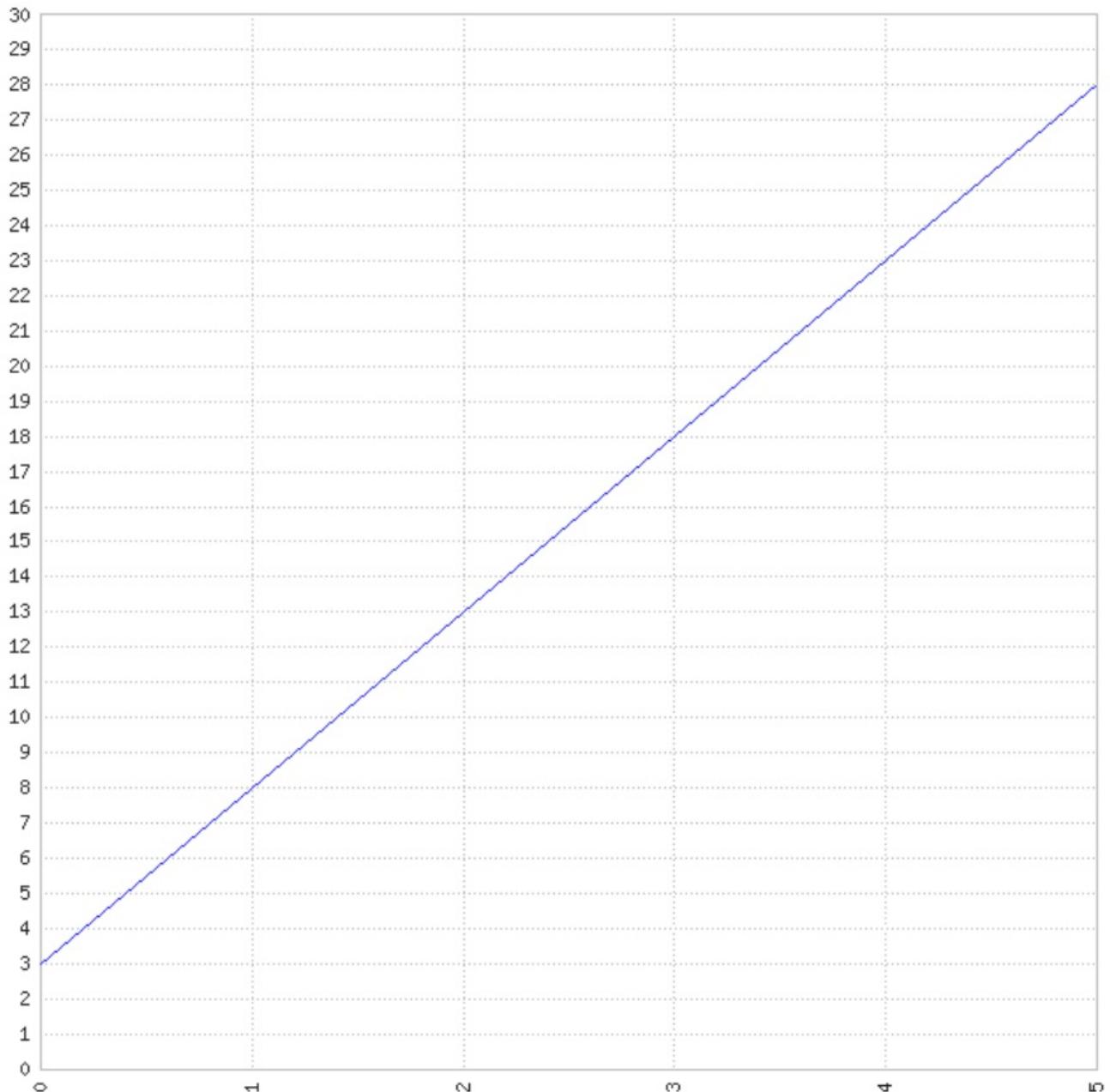
Notiamo che:

- il tempo t_{alg} di esecuzione di questo algoritmo dipende da quanto è grande la variabile n data in input.
- Tale tempo può essere "spezzato" in due parti:
 1. una parte di tempo t_{iter} , costituita dal tempo impiegato per una singola iterazione del ciclo, moltiplicata per il numero di volte in cui si ripete il ciclo;
 2. una parte di tempo t_{cost} , indipendente dal numero di volte in cui si ripete il ciclo.

- Dato che il numero di volte che viene ripetuto t_{iter} corrisponde ad n , il tempo complessivo dell'algoritmo si può quindi esprimere come $t_{alg} = t_{iter} \cdot n + t_{cost}$
- al crescere di n , il termine $t_{iter} \cdot n = 5n$ diventa preponderante sul termine $t_{cost} = 3$, che quindi può essere considerato trascurabile
- indipendentemente dal fatto che l'architettura di riferimento impieghi 1ms o 1μs o 5 s o 10 ns per ogni istruzione elementare, l'espressione per il calcolo del tempo sarà sempre la stessa (quindi posso evitare di esprimere l'unità di tempo)
- il tempo complessivo $t_{alg} = t_{iter}n + t_{cost}$ posso vederla come $t_{alg} = a \cdot n + b$ con $a, b \in \mathbb{R}$, $\Rightarrow t_{alg}$ è una funzione **lineare** sulla variabile di input n , quindi si dice che tale algoritmo, per essere completato, impiega un tempo **lineare** rispetto all'input.

Possiamo quindi fare un grafico per vedere come varia il tempo al variare del valore di n

(ossia avendo sulle y: tempo t_{alg} e sulle x: n)



Possiamo fare di meglio? In altri termini, esiste un algoritmo che mi permette di risolvere lo stesso problema ma in tempo minore?

In questo caso specifico, sì.

Ricordandoci la formula di Gauss per il calcolo della somma dei primi n numeri naturali:

$$\frac{n(n+1)}{2}$$

possiamo implementare il seguente algoritmo:

```
In [ ]: // algoritmo 1.2
int main()
{
```

```

int n;
printf("inserire numero: ");
scanf("%d", &n);

int s = n*(n+1)/2;

printf("la somma dei numeri da 1 a %d è %d\n", n, s);

return 0;
}

```

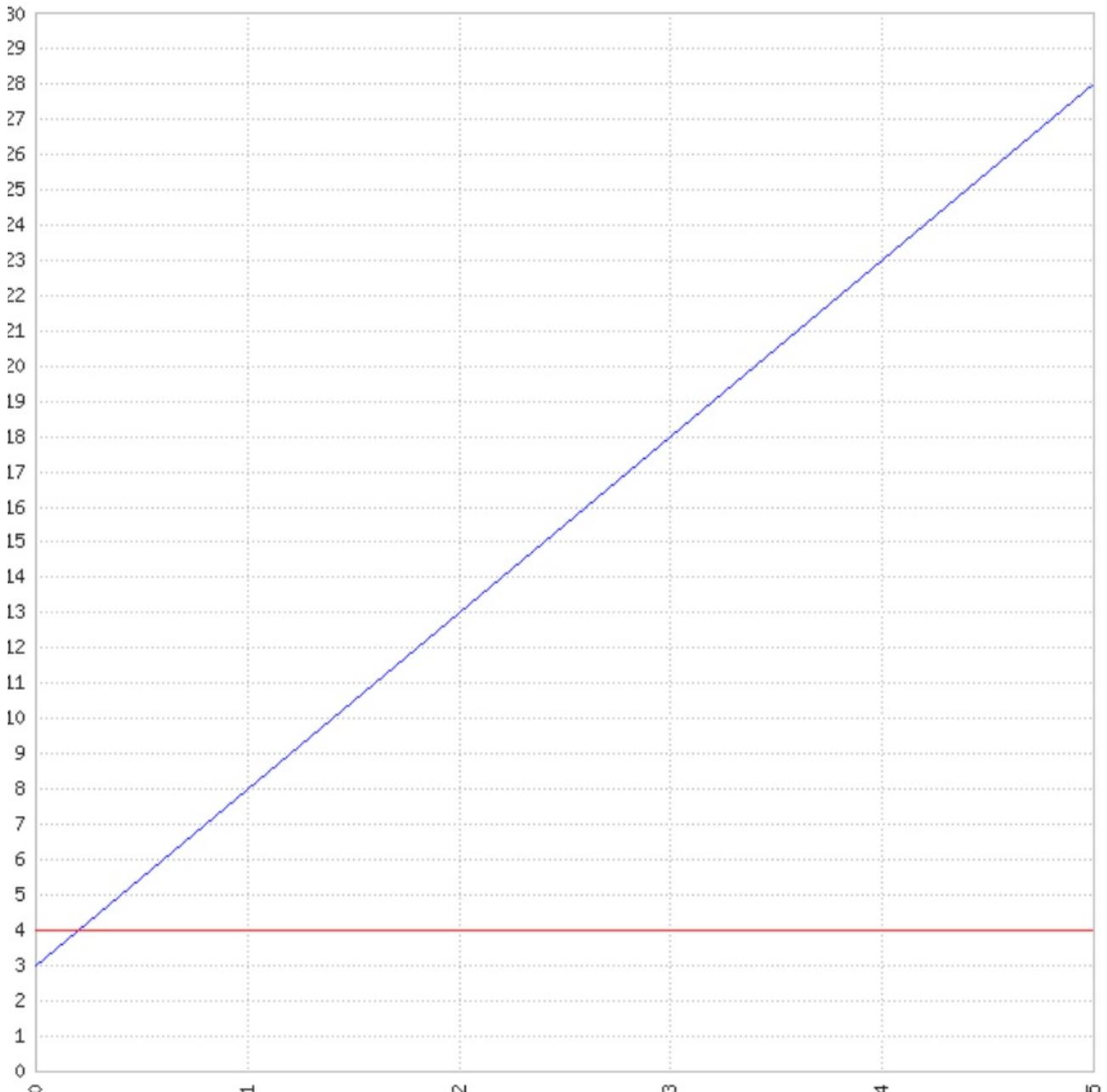
in questo caso, il tempo di esecuzione di questo algoritmo sarà dato da:

- 1 assegnazione
- 1 somma
- 1 moltiplicazione
- 1 divisione

totale: 4 operazioni elementari

In questo caso il tempo necessario all'algoritmo non dipende dalla dimensione della variabile n di input, ma sarà sempre pari a 4 **indipendentemente** dal valore assegnato ad n .

Si dirà quindi che tale algoritmo impiega un tempo *costante* rispetto all'input (ossia il tempo necessario ad essere completato non dipende dall'input).



Osserviamo il seguente grafico, avente come ascisse il valore n e come ordinate il tempo t di esecuzione:

La linea blu rappresenta il tempo dell'algoritmo 1.1 in funzione di n , ossia $5n + 3$, la linea rossa invece il tempo dell'algoritmo 1.2, ossia 4 a prescindere da n .

Ovviamente, il secondo algoritmo è preferibile al primo in termini di tempo, in quanto indipendente dall'input.

Esempio 2: ipotizziamo di voler sapere quanto tempo impiega ad essere completato il seguente algoritmo che restituisce la somma di tutti gli elementi contenuti in un vettore di lunghezza n

```
In [ ]: int somma_vett(int v[], int n)
{
    int s = 0;
    for(int i = 0; i < n; i++)
        s = s + v[i];

    return s;
}
```

Si hanno:

- 2 inizializzazioni $s=0$ e $i=0$
- $n+1$ confronti $i < n$
- n incrementi $i++$ (ognuno dei quali vale 2)
- n addizioni $s + \dots$
- n assegnazioni $s = \dots$

totale: $2 + n + 1 + 2n + n + n = 5n + 3$.

Anche in questo caso il tempo può essere espresso come $t_{iter}n + t_{cost}$

⇒ l'algoritmo impiega un tempo *lineare* rispetto alla dimensione n dell'array per essere completato.

Esempio 3: ipotizziamo di voler realizzare un algoritmo che, dato un array v di lunghezza n ed un valore k , restituisca l'indice in cui si trova k .

```
In [ ]: // algoritmo 3.1
int cerca_vett(int v[], int n, int k)
{
    int idx_k = -1;
    for(int i = 0; i < n; i++)
        if(k == v[i])
            idx_k = i;
    return idx_k;
}
```

Anche in questo caso il tempo di esecuzione dell'algoritmo sarà composto da una parte costante t_{cost} che è pari al tempo per effettuare l'inizializzazione $idx_k = -1$, e da una parte variabile t_{loop} che dipende dal numero di iterazioni di un ciclo, quindi $t_{alg} = t_{cost} + t_{loop}$.

Calcoliamo momentaneamente il tempo impiegato da *una singola* iterazione (quindi t_{iter}):

- 1 confronti $i < n$
- 1 confronto $k == v[i]$
- 1 assegnazione (solo se il confronto $k == v[i]$ da esito positivo)
- 1 incremento $i++$ (che vale 2)

tal iterazione sarà ripetuta n volte, a cui bisogna aggiungere il tempo dell'assegnazione $idx_k = -1$.

Ogni singola iterazione avrà quindi tempo 4 oppure 5 in base al fatto che il confronto $k == v[i]$ dia esito positivo o meno.

Semplifichiamo considerando due casi:

1. il valore k non è presente nell'array (ossia 0 volte): in questo caso ogni iterazione avrà durata $t_{iter}^0 = 4 \Rightarrow$ il tempo complessivo del ciclo sarà $t_{loop} = t_{iter}^0 n = 4n$
2. il valore k è presente in ogni posizione dell'array (ossia n volte): in questo caso ogni iterazione avrà durata $t_{iter}^n = 5 \Rightarrow$ il tempo complessivo del ciclo sarà $t_{loop} = t_{iter}^n n = 5n$

è facile vedere che in tutti gli altri casi t_{iter}^i , $0 < i < n$, la durata complessiva del ciclo sarà $4n < t_{loop} < 5n$.

Quindi in generale l'algoritmo avrà tempo di esecuzione $t_{alg} = t_{cost} + t_{loop} = 1 + w \cdot t_{loop}$, con $4 \leq w \leq 5$.

Si vede quindi che, indipendentemente dal valore effettivo di w , il tempo necessario all'algoritmo per convergere al risultato finale è **lineare** rispetto alla dimensione dell'array.

Si può fare di meglio?

```
In [ ]: // algoritmo 3.2
int cerca_vett(int v[], int n, int k)
{
```

```

int idx_k = -1;
int i = 0;
while(i < n && idx_k == -1)
{
    if( k == v[i])
    {
        idx_k = i;
    }
    i++;
}
return idx_k;
}

```

Tralasciamo per ora il tempo t_{cost} impiegato dalle istruzioni fuori dal ciclo.

Calcoliamo momentaneamente il tempo impiegato da *una singola* iterazione (quindi t_{iter}):

- 2 confronti `i < n`, `idx_k == -1`
- 1 and logico `&&`
- 1 confronto `k == v[i]`
- 1 assegnazione (solo se il confronto `k == v[i]` da esito positivo)
- 1 incremento `i++` (che vale 2)

totale: una singola iterazione impiega un tempo pari a $t_{iter} = 6$ unità per essere completato, o 7 se `k == v[i]`, ma in quel caso il ciclo terminerebbe subito dopo.

Semplifichiamo considerando soltanto $t_{iter} = 6$, tanto come abbiamo visto non cambia molto.

Vediamo adesso quante volte il blocco con tempo t_{iter} viene eseguito:

- se il valore `k` si trova nella prima posizione di `v`, il loop sarà eseguito 1 sola volta, in quanto il corpo del ciclo sarà eseguito una sola volta
- se il valore `k` si trova nella seconda posizione di `v`, il loop sarà eseguito 2 volte $\Rightarrow 2 \cdot t_{iter}$
- \vdots
- se il valore `k` si trova nell'ultima posizione di `v` (o non esiste in `v`), il loop impiegherà $n \cdot t_{iter}$ tempo per essere completato

Facciamo una distinzione su cosa succede nei diversi casi, ossia:

- *caso migliore*: il valore da cercare è in prima posizione \Rightarrow trascurando t_{cost} , il tempo necessario per completare l'algoritmo è pari a $t_{alg} = t_{iter}$ (quindi indipendente dal numero dei dati)
- *caso peggiore*: il valore da cercare non è presente in `v` oppure è in ultima posizione \Rightarrow trascurando t_{cost} , il tempo necessario per completare l'algoritmo è pari a $t_{alg} = n \cdot t_{iter}$
- in tutti gli altri casi: trascurando t_{cost} , il tempo necessario sarà compreso tra t_{iter} ed $n \cdot t_{iter}$. Solitamente il tempo di questi casi viene espresso come tempo necessario nel *caso medio*, che tiene conto di come l'algoritmo si comporta *in media* al variare degli input [non trattato in questo corso].

Ricapitolando, questo algoritmo impiega un tempo:

- *nel caso migliore, indipendente dalla dimensione dell'input* (ossia dal numero di dati in input), quindi ***costante***
- *nel caso peggiore, lineare** sulla dimensione dell'input*

Ciò è comunque meglio dell'avere un algoritmo che impiega un tempo lineare sulla dimensione dell'input *in ogni caso*.

Ipotizziamo di avere un array con un milione di elementi. Se abbiamo la fortuna che l'elemento `k` si trovi in una posizione $< n$ (magari la prima!), allora il tempo da attendere sarà $t_{alg} < 1000000 \cdot t_{iter}$, mentre nel caso del primo algoritmo sarà sempre $t_{alg} = 1000000 \cdot t_{iter}$ indipendentemente da dove si trovi il valore di `k`.

Esempio 4: dato un vettore `v` di lunghezza `n`, vogliamo calcolare quanto tempo impiega per essere completata la seguente funzione che restituisce l'indice di posizione del suo valore minimo

```

In [ ]: float idx_min_array(float v[], int n)
{
    int idx_min = 0;
    float min    = v[0];
    for (int i=1; i < n; i++)
        if (v[i] < min)
        {
            min    = v[i];
            idx_min = i;
        }
    return idx_min;
}

```

trascriviamo con t_{alg} il tempo necessario di completamento dell'algoritmo riportato, i.e. il tempo necessario per la ricerca del minimo in un array al variare della lunghezza n .

In generale, la ricerca del minimo su un array di dimensione n impiegherà un tempo complesivo pari alla somma de:

- il tempo per eseguire 3 assegnazioni iniziali: `idx_min = 0`, `min = v[0]`, `i=0` (parte di tempo costante t_{cost})
- il tempo t_{iter} per eseguire il corpo del ciclo, costituito da 2 confronti (`i < n` e `v[i] < min`) e 2 eventuali assegnazioni (solo se `v[i] < min`).
- il ciclo `for` va da 1 ad n escluso, quindi viene eseguito $n - 1$ volte.

Il tempo impiegato da questo algoritmo sarà quindi, nel caso peggiore, $t_{alg} = t_{iter} \cdot (n - 1) + t_{cost} = 4(n - 1) + 3$.

Quindi il tempo necessario per la ricerca del minimo è *lineare* sulla dimensione dell'input -1.

Esempio 5: ipotizziamo di voler vedere quanto tempo impiega un algoritmo che, dato un array `v` di lunghezza `n`, restituisca un vettore `freq` contenente, in ogni posizione `i`-esima, la frequenza di ogni valore `v[i]` in `v` (ossia quante volte ogni `v[i]` è contenuto in `v`).

```
In [ ]: void freq(int v[], int n, int freq[])
{
    // ciclo esterno
    for(int i = 0; i < n; i++)
    {
        freq[i] = 0;
        // ciclo interno
        for(int j = 0; j < n; j++)
        {
            if(v[j] == v[i])
                freq[i]++;
        }
    }
}
```

Analizziamo il tempo di esecuzione del ciclo interno:

Trascurando le inizializzazioni che darebbero luogo a t_{cost} , il corpo del ciclo più interno è composto composto da:

- 2 confronti: `v[j] == v[i]` e `j < n`
- 2 incrementi: `freq[i]++` e `j++` (ognuno dei quali vale 2)

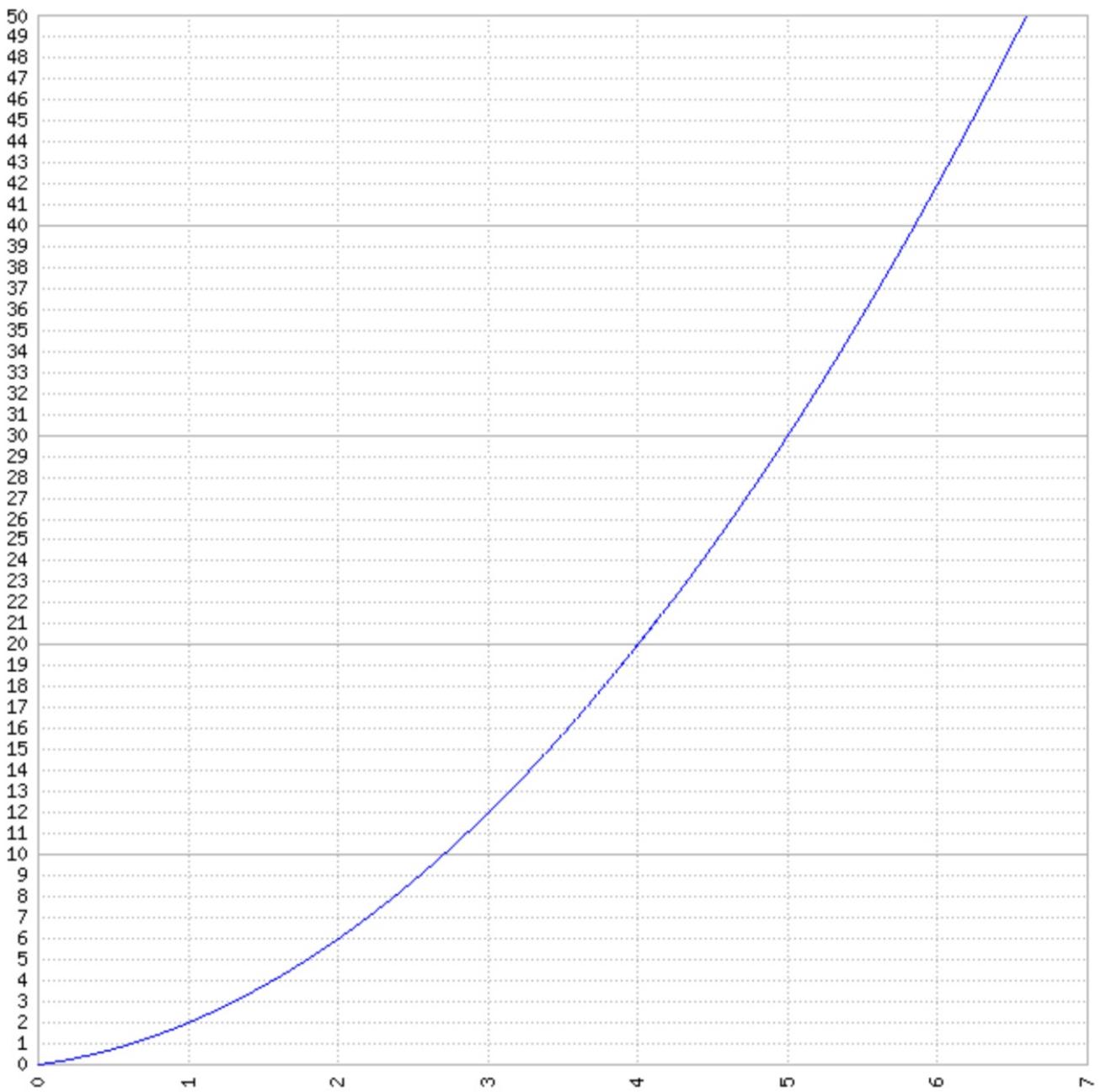
totale: $t_{iter} = 6$ operazioni elementari.

- Tali 6 operazioni vengono ripetute n volte $\Rightarrow 6n$, o anche $t_{iter} \cdot n$ (per semplificare, trascuriamo il tempo del primo confronto `j < n` di ingresso nel ciclo).
- Tali $t_{iter} \cdot n$ operazioni vengono ripetute n volte nel ciclo esterno, più un'assegnazione `freq[i] = 0`

$$\Rightarrow t_{alg} = n \cdot (1 + t_{iter}n) = t_{iter}n^2 + n$$

Per questo algoritmo, il tempo di esecuzione necessario essere completato varia in maniera *quadratica* rispetto alla dimensione dell'input.

\Rightarrow Tempo abbastanza alto anche per vettori non troppo grandi.



Esempio, per $n = 10$ il tempo necessario è dato da $t_{iter}n^2 + n = t_{iter} \cdot 100 + 10$, per $n = 1000$ il tempo necessario diventa $t_{iter} \cdot 1000000 + 1000$!

Considerazioni

- Si nota che la parte davvero preponderante (soprattutto all'aumentare di n) del tempo necessario per un algoritmo è generalmente **funzione dell'input n** , motivo per cui il tempo utilizzato t_{cost} per la parte rimanente (inizializzazioni, ecc.) può essere trascurato.
- Esistono ovviamente anche algoritmi con tempi di esecuzione molto più alti, ad esempio *cubico* sulla dimensione dell'input, (n^3) o, addirittura, *esponenziale* sulla dimensione degli input (2^n). Un algoritmo con tempi alti come l'esponenziale sono generalmente considerati inattuabili nella pratica, in quanto anche per piccoli valori di n , risultano impiegare troppo tempo per avere risultati in tempi utili. Esempio: per $n = 30$, $2^{30} = 1073741824!!!$
- Con il termine **complessità di tempo** intendiamo quanto tempo un dato algoritmo impiega per essere completato *al variare degli input*.
- Con il termine **complessità di spazio** intendiamo quanto spazio in memoria un dato algoritmo necessita per essere completato *al variare degli input* [non trattata in questo corso].
- Ogni algoritmo ha le propria complessità. In teoria, dato un problema, si cerca l'algoritmo risolutivo che abbia la complessità di tempo e/o di spazio più bassa/e possibile.