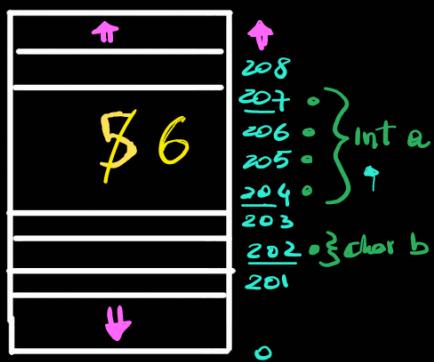


## Memoria (RAM)



## Pointers in C



Ognuno ha  
UN INDIRIZZO

consideriamo  
sia 4 byte  
di memoria

- quando dichiariamo
- int  $a;$  → inizia dall'Indirizzo 204 ↑
- char  $b;$  → 202
- $a = 5;$
- $a++;$  → aggiorna  $a$  da 5 a 6

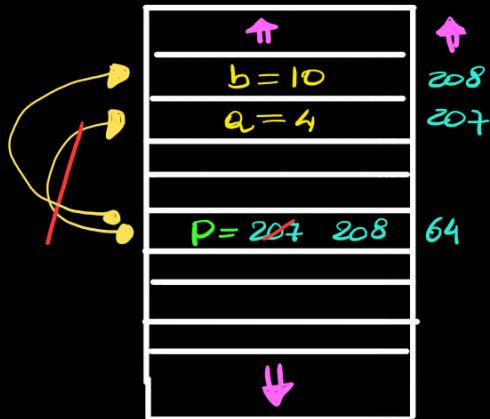
- int 4 byte
- char 1 byte
- float 4 byte

come possiamo operare con  
l'indirizzo di queste  
variabili?

grazie ai  
PUNTATORI

variabili che memorizzano  
l'indirizzo di un'altra variabile

## Memoria (RAM)



- int  $a;$
- int  $*p;$  → dichiara  $p$  come un puntatore a int\*
- $p = &a;$  → restituisce l'indirizzo di  $a$
- print  $p \Rightarrow 207$
- print  $*p \Rightarrow 207$
- print  $\&p \Rightarrow 64$
- print  $\&a \Rightarrow 5$
- $*p = 8$  → aggiorna il valore di  $a$
- print  $a \Rightarrow 8$

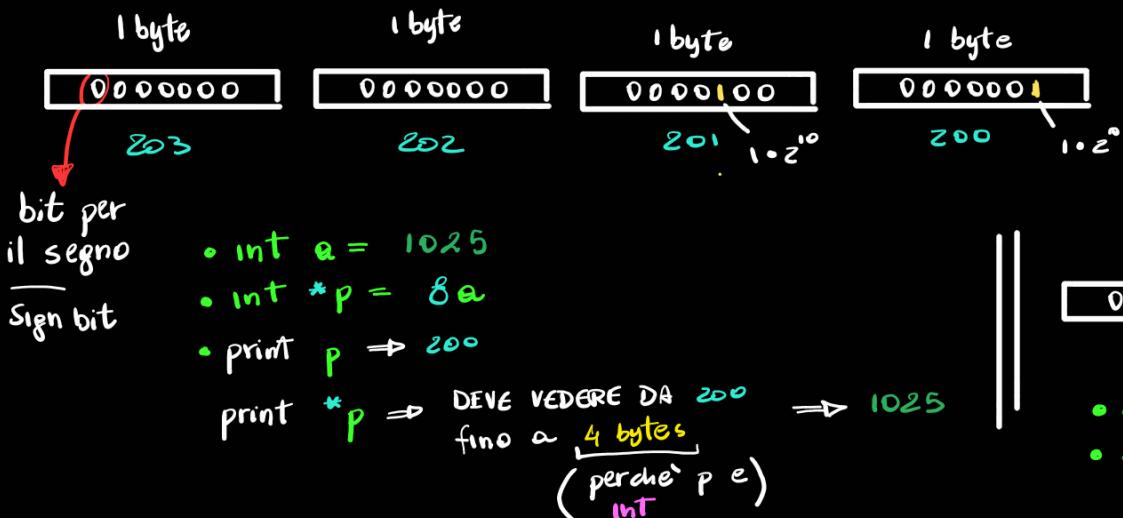
# Type Pointers

Why strange types?

Why not some generic types?

perché abbiamo  
bisogno di usare  
un puntatore dello  
stesso tipo?

- int - 4 bytes
- char - 1 bytes
- float - 4 bytes



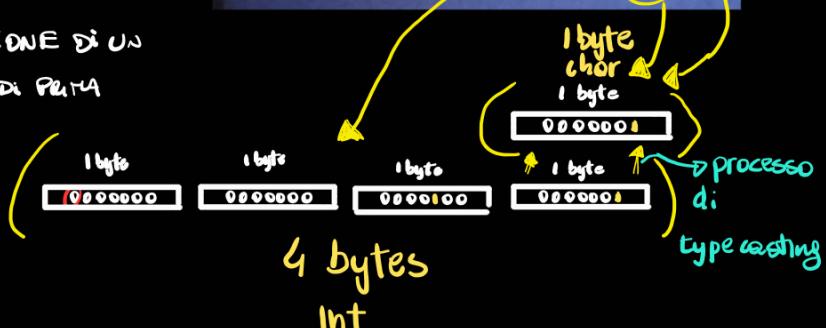
Se prendiamo un puntatore  
di tipo **char\*** e lo lo assegniamo  
a un puntatore **int\*** non si può fare  
bisogna "costarci" (forzare a cambiare tipo)  
• la cosa interessante è che se proviamo  
a vedere il valore del puntatore (**char\***)  
OK sarà lo stesso di prima (**int\***)  
ma il valore SARÀ SOLO LA PORZIONE DI UN  
1 BYTE DEL INT DI PRIMA

```
int a = 1025;
int *p = &a;
printf("%p --> %d\n", p, a);
char *pc = (char*)p; // typecasting
printf("%p --> in decimale : %d\n", pc, *pc);
printf("%p --> in char : %c\n", pc, *pc);
```

esempio

↓ output

```
:!./out
0x7ffc8b1ae504 --> 1025
0x7ffc8b1ae504 --> in decimale : 1
0x7ffc8b1ae504 --> in char : ^A
```



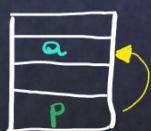
se abbiamo un puntatore di tipo **void**  
possiamo avere solo l'indirizzo e nessun valore



## Pointers Arithmetic

```
C puntatori.c    x
#include <stdio.h>
#include <stdlib.h>

int main(){
    int a = 10;
    int *p = &a;
    printf("%p --> %d\n", p, a);
    printf("%p --> %d\n", p+1, *(p+1));
}
```

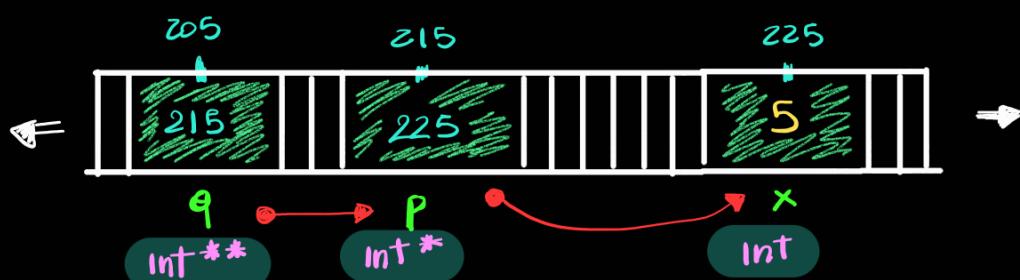


```
:!./out
0x7ffd75a9ba7c --> 10
0x7ffd75a9ba80 --> 1974057596
```

1 byte

## Pointers To Pointers

Memoria ↴



- Int x = 5

- Int\* p = &x

- Int\*\* q = &p

print \*p => 5

print \*q => 215

print \*(\*q) => 5

oppure  
\*\*q

```
int a = 5;
int *p = &a; // oppure int* p = &a;
printf("%p --> %d\n", p, a);
int** q = &p;
printf("%p --> %d\n", p, *q, **q);
```

↓ output

```
:!./out
0x7ffee4d6f2b4 --> 5
0x7ffee4d6f2b8 --> 0x7ffee4d6f2b4 --> 5
```

{ possiamo anche  
creare un puntatore  
che punta ad  
un puntatore }

## Pointers as function arguments

call by reference

(utilizzo dei puntatori nelle funzioni)



ALBERT

=>  
scrive  
la funzione

```
void INCREMENT(int a){
    a = a + 1;
}

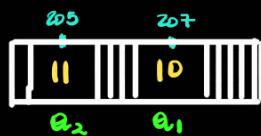
int main(){
    int a = 10;
    INCREMENT(a);
    printf("a = %d\n", a);
}
```

=> a = 11  
quello che si aspetta di vedere Albert

↓ output vero

```
void INCREMENT(int a){
    a = a + 1; UN'ALTRA a (LOCALE)
}

int main(){
    int a = 10; LOCALE a
    INCREMENT(a);
    printf("a = %d\n", a);
}
```

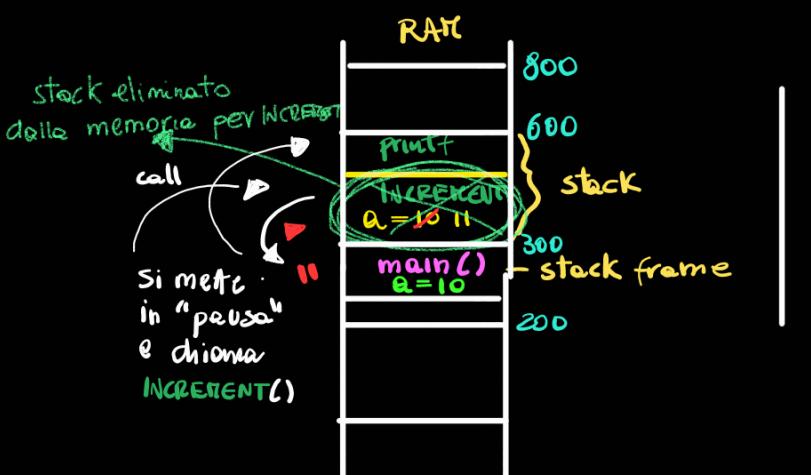
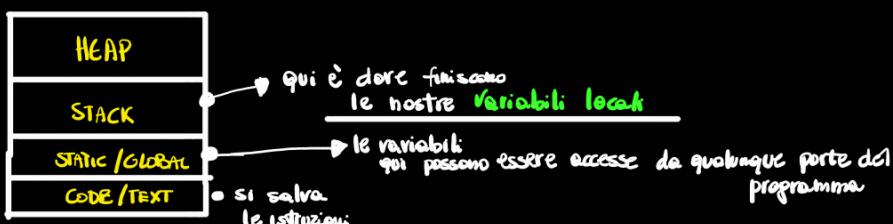


?

: ./out  
a = 10

?

Quando un programma viene eseguito il OS assegna un "pezzo" di memoria al programma



```
void INCREMENT(int a){
    a = a + 1;
}

int main(){
    int a = 10;
    INCREMENT(a);
    printf("a = %d\n", a);
}
```

a → a copia

formal argument

CHIAMATA PER VALORE

actual argument

All BY VALUE

## CHIAMATA PER RIFERIMENTO

(CALL BY REFERENCE)

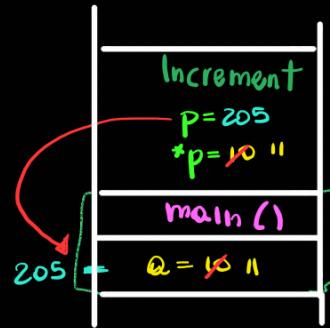
```
void INCREMENT(int* pa){
```

```
    *pa = *pa + 1; } } lavoriamo  
direttamente  
con il valore  
di *pa
```

```
int main(){
```

```
    int a = 10;  
    INCREMENT(&a); → passiamo  
    printf("a = %d\n",a); l'indirizzo di a
```

```
}
```



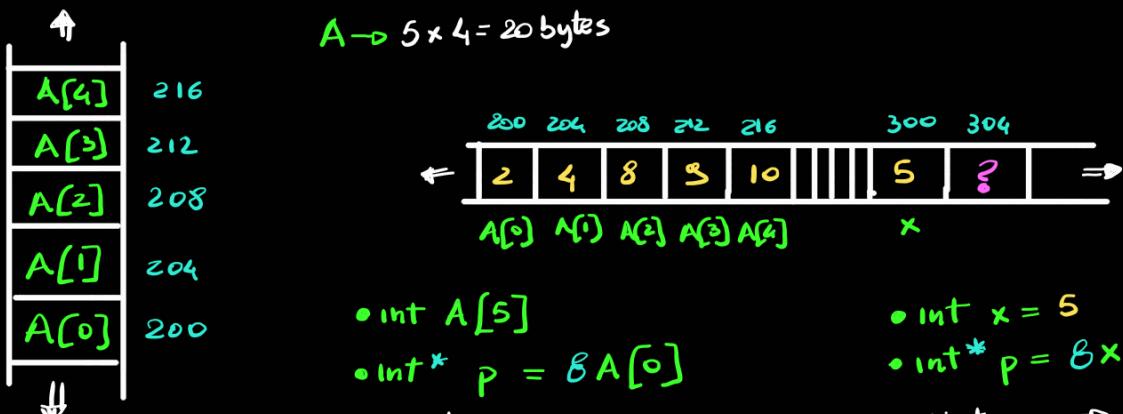
## Pointers and Arrays

- Supponiamo di avere un array di 5 elementi:

- $\text{int } A[5]$

- int - 4 bytes

$A \rightarrow 5 \times 4 = 20 \text{ bytes}$



- $\text{int } A[5]$

- $\text{int } * p = &A[0]$

- $\text{print } p \Rightarrow 200$

- $\text{print } *p \Rightarrow 2$

- $\text{print } (p+1) \Rightarrow 204$

- $\text{print } *(p+1) \Rightarrow 4$

- $\text{int } x = 5$

- $\text{int } * p = &x$

- $\text{print } p \Rightarrow 300$

- $\text{print } *p \Rightarrow 5$

- $p = p + 1$

- $\text{print } p \Rightarrow 304$

- $\text{print } *p \Rightarrow ?$

Una cosa interessante  
è che :



- $\text{int } * p = A$

$\text{print } p \Rightarrow 200$

$\text{print } A \Rightarrow 200$

$\text{print } *A \Rightarrow 2$

per accedere ad un elemento  $i$   
in  $A$

Indirizzo :  $&A[i]$  oppure  $(A+i)$

Valore :  $A[i]$  oppure  $*(A+i)$



```
#include<stdio.h>
int main()
{
    int A[] = {2,3,4,5,6,7,8,9};
    printf("%p\n",A);
    printf("%p\n",&A[0]);
    printf("%d\n",A[0]);
    printf("%d\n",*A);
}
```

Messages  
:!.out  
0x7fd62ed690  
0x7fd62ed690  
2  
2

```
21 #include<stdio.h>
20
19     (int *A)
18     int somma_elementi(int A[]){
17         - int sum = 0;
16         int size = sizeof(A)/sizeof(A[0]);
14         -
13         for (int i = 0;i < size; i++) {
12             sum += A[i];
11         }
10         return sum;
9     }
8 }

5
int main()
4 {
3     int A[] = {2,3,4,5,6,7,8,9};
2     int totale = somma_elementi(A);
1     printf("%d\n",totale);
22
1
2 }
```

Messages  
:!.out  
? 5 ?  
? 5 ?

il problema sta nel fatto che quando la funzione somma\_elementi prende come parametro int A[] non sta prendendo l'array interamente, ma sta prendendo l'indirizzo dell'array

- **Sizeof(A)  $\Rightarrow$  4 (bytes)**  
perche' A in verita' e' un puntatore all'array di tipo int  
**(int \*A)**

- Una soluzione  
Sarebbe di passare size alla funzione somma\_elementi.

# Pointers and dynamic memory

## • Che cosa è la memoria dinamica?

la memoria dinamica è una "porzione" di memoria allocata durante l'esecuzione del programma e anche durante la compilazione.

→ (questa fase si chiama **RUNTIME**)

↓ quindi mentre il programma è in esecuzione

c'è servizio delle istruzioni

che ci permettono di "rubare" la memoria che ci serve,

(dico "rubare"  
perché nel momento che  
compilano il programma  
il sistema operativo  
ci dà un numero fissato  
di memoria)

→ questo porzione  
di memoria si chiama

### STACK



Invece da dare  
"robiamo" la memoria  
si chiama **HEAP**  
che è una porzione  
di memoria più grande

## • Funzioni di cui abbiamo bisogno

① dobbiamo includere il file stdlib.h

② Void \* malloc (size\_t size);

questa funzione ci permette di allocare memoria di cui abbiamo bisogno

● size : è la quantità di memoria da allocare (in byte)

● restituisce un puntatore al blocco di memoria allocato (oppure **NULL** in caso di errore)

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

int main(){
    int *p = malloc(10 * sizeof(int))
    /*
        ISTRUZIONI CON IL VETTORE "p"
    */
}
```

qui possiamo mettere anche (\*p) solo se già prima dichiarato

questa istruzione pesa O(1)

BEST PRACTICE ✓  
Ricordiamoci  
di controllare  
sempre se è andato  
tutta a buon  
fine con NULL

③ `Void* realloc (void* ptr, size_t new_size);`

- modifica la dimensione di un blocco di memoria già allocato

- Ci sono due casi che possono accadere:

① Se c'è abbastanza spazio al blocco adiacente  
il blocco viene esteso senza spostare niente.

caso migliore

$O(1)$

② Se non c'è spazio sufficiente, viene allocato un nuovo blocco di memoria di dimensione new\_size e i dati esistenti vengono copiati nel nuovo blocco, il vecchio viene deallocato.

caso peggiore

$O(1) + O(n) = O(n)$

allocazione  
del nuovo blocco

spostamento  
(copia dei  
dati)

{  
 $n$  è la  
dimensione  
del vecchio  
blocco}

③ `Void free (void* ptr);`

- libera un blocco di memoria già utilizzato (allocato)  
e bisogna passargli il puntatore al blocco da liberare.
- Serve per evitare i memory leaks

$\Rightarrow$

$O(1)$

## Dynamic Array

- Vogliamo implementare un array dinamico  
diciamo che vogliamo implementare la lista che conosciamo in Python  
il problema è che siamo su C, in linguaggio di programmazione  
dove gli array hanno dimensione fissa.



Un primo metodo è il seguente:

### ① Primo Metodo (Bad $O(n^2)$ )

possiamo anzitutto allocare  
un array in memoria con n spazio (10 nell'esempio)  
attraverso malloc e poi ogni volta  
che aggiungono un elemento "espanderà" l'array con realloc.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(){
5
6     int m = 100;
7     int n = 1;
8     float* a = malloc(1*sizeof(float)); ●
9     a[0] = 10;
10
11    for(int i = 0; i < m; i++){
12        /* Nel caso peggiore tempo  $O(n^2)$ .
13         * Osservazione: il costo di n append() in Python è, nel caso peggiore,  $O(n)$ 
14         * */
15        n++;
16        a = realloc(a, n*sizeof(float)); ●
17        a[n-1] = n;
18    }
19
20    for(int i = 0; i < n; i++){
21        printf("%f ", a[i]);
22    }
23    printf("\n");
24 }
```

⚠ il problema è che realloc come abbiamo visto prima nel caso peggiore  
"pesa"  $O(n)$ , ma un  $O(n)$  all'interno di un ciclo per m volte, diventa  
in totale una complessità QUADRATICA  $O(n^2)$ , e quindi dobbiamo trovare  
un'altra strada

- almeno procediamo  
con una tattica e  
alcuni strumenti:  
oggiuntivo.



Prima di mostrare il secondo metodo  
dobbiamo introdurre un nuovo "tipo di dato"  
che ci verrà in aiuto.



## • STRUCT

Le struct sono un tipo di dato  
composito, che permette di raggruppare  
variabili di tipo diverso sotto un unico "box".  
È molto utile per rappresentare entità complesse  
con più attributi associati

## • Esempio

```
struct Punto {  
    float x; // Coefficiente x  
    float y; // Coefficiente y  
};
```

• per dichiarare

L struct Punto   p1;

• per accedere agli attributi

 p1.x = 5.0;

possiamo utilizzare  
**typedef** per rinominare (alias)  
un tipo di dato esistente.  
• **typedef tipo-esistente nuovo-name**

```
typedef struct {  
    float x;  
    float y;  
} Punto;  
  
int main() {  
    Punto p1; /  
    p1.x = 1.0;  
    p1.y = 2.0;  
    return 0;  
}
```

## ④ Secondo metodo (Good O(n))

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 typedef struct {
5     int*data;
6     int size;
7     int capacity;
8 } dynamic_array;
9
10 dynamic_array init(int initial_capacity);
11 void append(dynamic_array *array, int element);
12
13 int main(){
14
15     dynamic_array array = init(4);
16
17     for(int i = 1; i < 6; i++){
18         append(&array,i);
19     }
20
21     printf("elementi nell'array dinamico : \n");
22     printf("%d",array.size);
23     for(int i = 0; i < array.size; i++){
24         printf("elemento in %d -> %d\n",i,array.data[i]);
25     }
26
27     return 0;
28
29 }
30
31 dynamic_array init(int initial_capacity){
32     dynamic_array init_array = {NULL,0,0};
33     init_array.data = malloc(initial_capacity*sizeof(int));
34     init_array.size = 0;
35     init_array.capacity = initial_capacity;
36     return init_array;
37 }
38
39 void append(dynamic_array *array, int element){
40
41     if(array->size == array->capacity){
42         array->capacity = array->capacity * 2; // raddoppio la capacity
43         array->data = realloc(array->data, array->capacity * sizeof(int));
44         //controllo se tutto in check
45         if(array->data == NULL){
46             printf("errore nella riallocazione della memoria in append function!\n");
47             exit(1); // con questa finisco tutto il programma con codice 1 (errore)
48         }
49     }
50     array->data[array->size] = element;
51     array->size++;
52 }
```

