



Programmazione 1 (Canale O-Z)

Lezione 8: I Puntatori e la Gestione Esplicita della Memoria

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

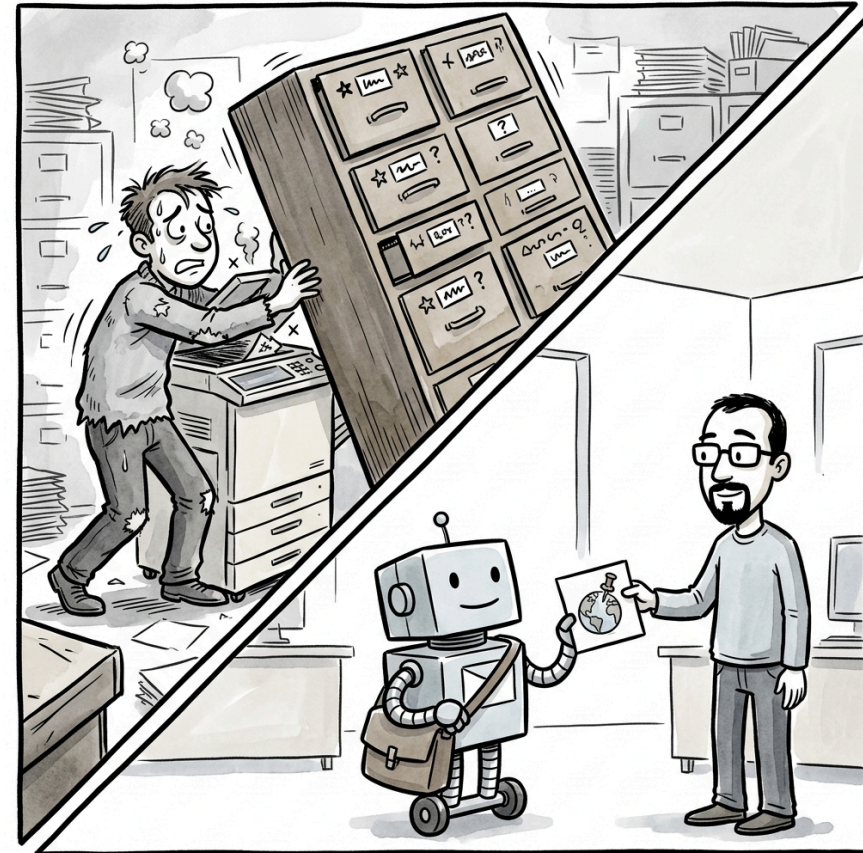
Sito Web: antoninofurnari.github.io/prog1-2627



Il Limite delle Fotocopie

- Le funzioni ricevono solo **copie** dei dati (passaggio per valore).
- Due limiti evidenti:
 - una funzione **non può modificare** una variabile del chiamante;
 - copiare grandi quantità di dati costa **tempo e memoria**.
- Oggi scendiamo nei dettagli fisici della RAM: i **puntatori**.

? Come fa una funzione a raddoppiare una variabile del `main` senza `return`? Eppure `scanf` lo fa da settimane con `&x ...`



La RAM e l'Illusione delle Variabili

- La CPU non sa cosa sia una variabile con nome (es. `x`).
- La RAM è una sequenza lineare di celle (byte), ciascuna con un **indirizzo fisico** numerico (es. `0x7fffd9b3`).
- In Assembly il nome svanisce: resta un **offset** rispetto a `rbp` (base dello stack frame).
- Se conosciamo l'indirizzo di una cella, possiamo leggerla/scriverla **scavalcando l'isolamento** dei record di attivazione.

C:

```
int x = 42;  
x = 100;
```

Assembly x86-64:

```
mov DWORD PTR [rbp-4], 42 ; scrivi 42 in RBP-4  
mov DWORD PTR [rbp-4], 100 ; sovrascrivi con 100
```

Cos'è un Puntatore?

- Un **puntatore** è una variabile che contiene un **indirizzo di memoria**.
- A 64 bit un indirizzo è lungo 64 bit ➡ ogni puntatore occupa **esattamente 8 byte**, qualunque sia il tipo puntato.

```
int *p;    // p è un puntatore ad un intero  
double *q; // q è un puntatore a un double
```

- Si legge **da destra a sinistra**: "*p* è un puntatore (***) a un intero (*int*)".

? Se tutti gli indirizzi occupano 8 byte, perché non esiste un unico tipo generico `pointer`
`p;` ?

Perché i Puntatori sono Tipizzati?

Il tipo non dice quanto è grande l'indirizzo (sempre 8 byte), ma **come interpretare la memoria** una volta arrivati a destinazione:

Puntatore	Cosa legge/scrive all'indirizzo
<code>char *p</code>	1 byte
<code>int *p</code>	4 byte consecutivi
<code>double *p</code>	8 byte consecutivi (standard IEEE 754)

- Il tipo è la "lente" con cui guardiamo le celle di memoria.

Layout in Memoria dei Puntatori

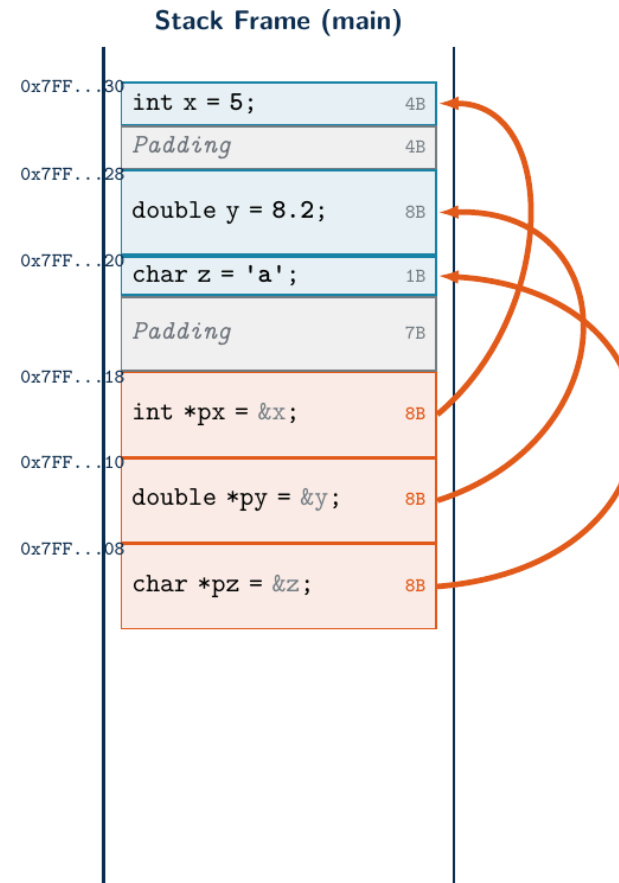
// Variabili Standard

```
int x = 5;  
double y = 8.2;  
char z = 'a';
```

// Variabili Puntatore

```
int *px = &x;  
double *py = &y;  
char *pz = &z;
```

- `x`, `y`, `z` occupano 4, 8, 1 byte.
- `px`, `py`, `pz` occupano **tutti 8 byte**.



! Trappola Sintattica: Dichiarazioni Multiple

```
int* p, q; // Crea p come puntatore a int, ma q come normale intero!
```

- L'asterisco si lega **solo** all'identificatore immediatamente successivo.
- **Best practice:** un solo puntatore per riga.

```
int *p = NULL;  
int q = 0;
```

? Che cosa c'è dentro `p` se lo dichiariamo senza inizializzarlo?

❌ L'Importanza di Inizializzare a **NULL**

- Un puntatore non inizializzato è un **dangling pointer** (puntatore selvaggio): contiene un indirizzo spazzatura casuale.
- Accedervi provoca crash (**Segmentation Fault**) o comportamenti imprevedibili.
- **Regola aurea**: inizializzare sempre i puntatori non associati a **NULL** (o **0**).
- L'indirizzo **0** è riservato dal Sistema Operativo: non è accessibile dal programma utente.

```
int *ptr = NULL; // ptr non punta a nessun indirizzo valido
```

- **NULL** dichiara: *"questo puntatore al momento è vuoto"* ➡ possiamo **controllarlo** prima di usarlo.

Gli Operatori `&` (Indirizzo) e `*` (Dereferenziazione)

- `&` (address-of): restituisce l'indirizzo di una variabile esistente.
- `*` (dereferenziazione): accede al valore all'indirizzo puntato, in lettura o scrittura.

```
#include <stdio.h>

int main(void) {
    int x = 42;
    int *p = &x; // p memorizza l'indirizzo di x

    printf("Valore di x: %d\n", x);           // Stampa 42
    printf("Indirizzo di x: %p\n", (void*)p); // Stampa es. 0x7ffd82

    *p = 100; // Dereferenziazione: vai all'indirizzo in p e scrivi 100

    printf("Nuovo valore di x: %d\n", x);     // Stampa 100!
    return 0;
}
```



& e *: Esecuzione

```
gcc -Wall puntatori.c -o puntatori  
./puntatori
```

```
Valore di x: 42  
Indirizzo di x: 0x7ffd...  
Nuovo valore di x: 100
```

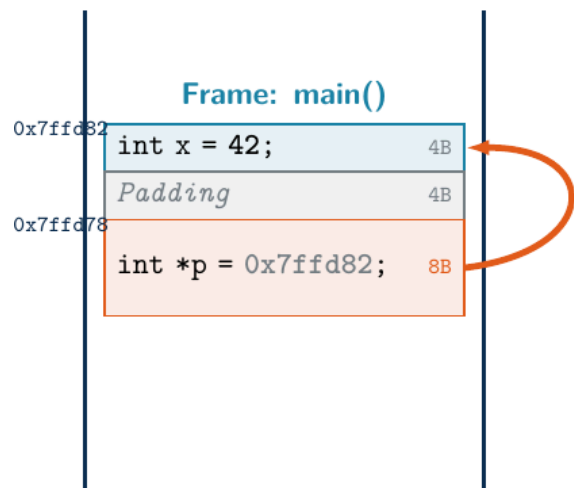
- Non abbiamo mai scritto `x = 100`: abbiamo scritto **all'indirizzo** di `x`.
- L'indirizzo stampato con `%p` cambia a ogni esecuzione.

 **Live:** compilare con `-Wall`, eseguire due o tre volte e far notare che l'indirizzo stampato cambia. Poi riprendere `scanf("%d", &x)`: ora sappiamo cosa riceve `scanf`.

Sotto il Cofano della Dereferenziazione

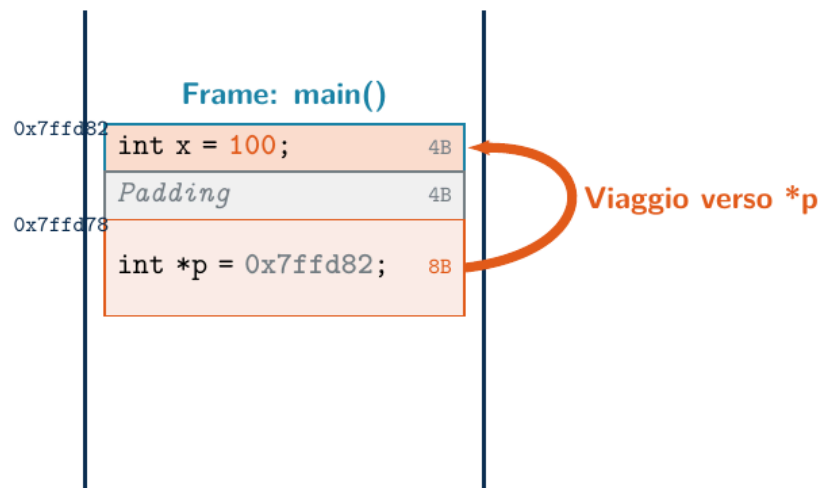
t_0 - Inizializzazione

```
int *p = &x;
```



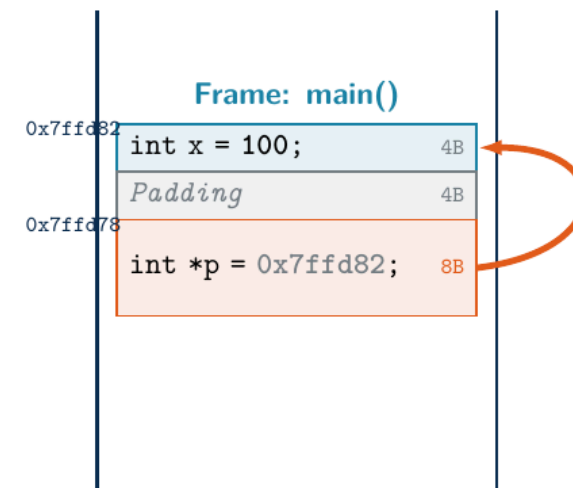
t_1 - Dereferenziazione

```
*p = 100;
```



t_2 - Stato Finale

```
printf("%d", x); // 100
```





Sicurezza nell'Accesso

Prima di dereferenziare, sfruttiamo l'inizializzazione a `NULL` per verificare che l'indirizzo sia valido:

```
if (ptr  $\neq$  NULL) {  
    *ptr = 10; // Sicuro  
}
```

? Cosa succede se in un programma più strutturato ci dimentichiamo questo controllo?

Esempio di Crash: Manca il Controllo

```
#include <stdio.h>

void elabora_sensore(const int *dati, int *risultato) {
    // BUG: Mancano i controlli di sicurezza!
    // Se dati o risultato sono NULL, avremo un crash.
    *risultato = (*dati) * 2;
}

int main(void) {
    int sensor_val = 25;
    int output = 0;

    // Caso d'uso corretto:
    elabora_sensore(&sensor_val, &output);
    printf("Risultato 1: %d\n", output);

    // Caso d'uso errato (ad esempio, sensore scollegato o guasto):
    int *sensore_guasto = NULL;
    elabora_sensore(sensore_guasto, &output); // Qui il programma andrà in crash!

    printf("Risultato 2: %d\n", output);
    return 0;
}
```

- Esecuzione ➡ Segmentation fault, **senza** indicare quale riga ha causato il problema.



Debugging con LLDB: Compilare ed Eseguire

1. Compilare con i simboli di debug (`-g` , sempre insieme a `-Wall`):

```
gcc -Wall -g sensore.c -o sensore
```

2. Avviare LLDB sull'eseguibile:

```
lldb ./sensore
```

3. Eseguire il programma dentro LLDB:

```
(lldb) run
```

 **Live:** prima eseguire `./sensore` normalmente e mostrare il `Segmentation fault` muto; poi ripartire con `lldb`.

LLDB: Analizzare il Punto di Arresto

4. LLDB intercetta il crash e mostra la riga colpevole:

```
Process 12345 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x0)
  frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0x0000000000000000, risultato=...) at sensore.c:6:18
    3      void elabora_sensore(const int *dati, int *risultato) {
    4          // BUG: Mancano i controlli di sicurezza!
    5          // Se dati o risultato sono NULL, avremo un crash.
→   6          *risultato = (*dati) * 2;
    7      }
```

- Stop alla **riga 6** di `sensore.c`, dentro `elabora_sensore`.
- Motivo: `EXC_BAD_ACCESS` = accesso a una zona di memoria non consentita.
- Il parametro `dati` vale `0x0000000000000000`, cioè `NULL`.



LLDB: la Catena delle Chiamate (backtrace)

5. Esaminare lo stack dei frame con `backtrace` (o `bt`):

```
(lldb) backtrace
* frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0x0000000000000000, risultato=...) at sensore.c:6:18
  frame #1: 0x0000000100000fc4 sensore`main at sensore.c:19:5
```

- `elabora_sensore` è stata invocata dal `main` alla **riga 19**: ecco il `sensore_guasto`.
- Usciamo dal debugger con `quit` (o `q`).

? Ora che sappiamo **dove** e **perché** il programma va in crash, come lo correggiamo?

LLDB: Risolvere il Bug

6. Aggiungere la guardia `if` all'inizio della funzione:

```
void elabora_sensore(const int *dati, int *risultato) {  
    if (dati == NULL || risultato == NULL) {  
        printf("Errore: puntatori non validi.\n");  
        return;  
    }  
    *risultato = (*dati) * 2;  
}
```

 **Live:** ricompilare `gcc -Wall -g sensore.c -o sensore && ./sensore` ➡ Risultato 1: 50, poi il messaggio di errore e Risultato 2: 50 (l'output non è stato toccato), senza crash.

Il Limite del Passaggio per Valore: lo Swap

```
#include <stdio.h>

void swap_sbagliato(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main(void) {
    int x = 5;
    int y = 10;
    swap_sbagliato(x, y); // Passa i valori di x e y (fotocopie)
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

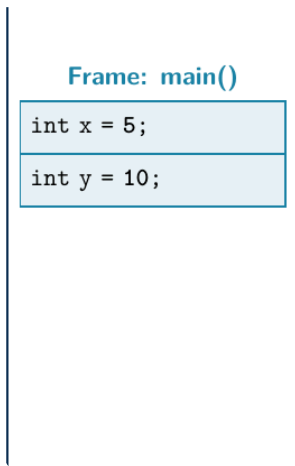
? Cosa stampa questo programma?



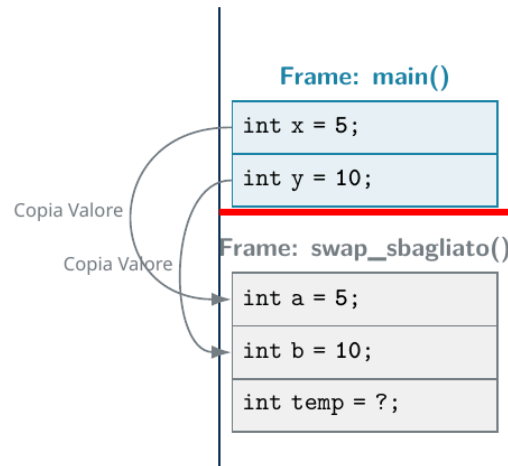
Swap per Valore nello Stack

Output: `x = 5, y = 10`. `swap_sbagliato` scambia solo le **copie** `a` e `b`; al `return` il frame viene distrutto.

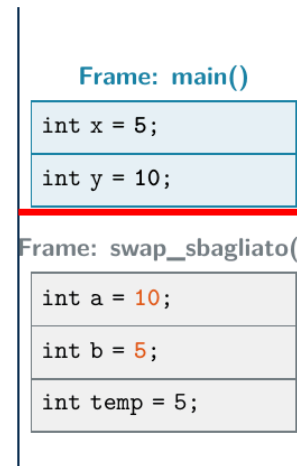
t_0 - Avvio



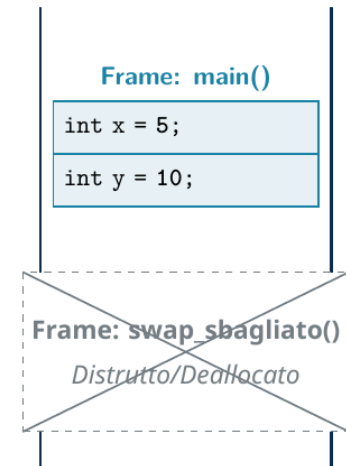
t_1 - Chiamata



t_2 - Scambio Locale



t_3 - Dopo il return



Il Passaggio per Riferimento Simulato

Passiamo alla funzione gli **indirizzi** delle variabili:

```
#include <stdio.h>

void swap_corretto(int *a, int *b) {
    int temp = *a;    // Copia il valore puntato da a
    *a = *b;           // Scrive il valore puntato da b all'indirizzo a
    *b = temp;         // Scrive temp all'indirizzo b
}

int main(void) {
    int x = 5, y = 10;
    swap_corretto(&x, &y); // Passa gli indirizzi di x e y
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

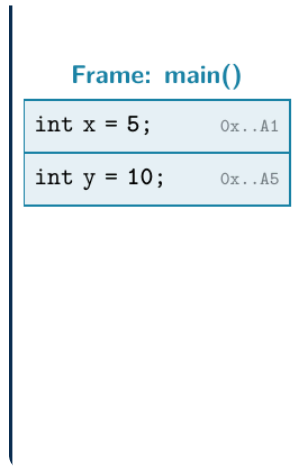
x = 10, y = 5



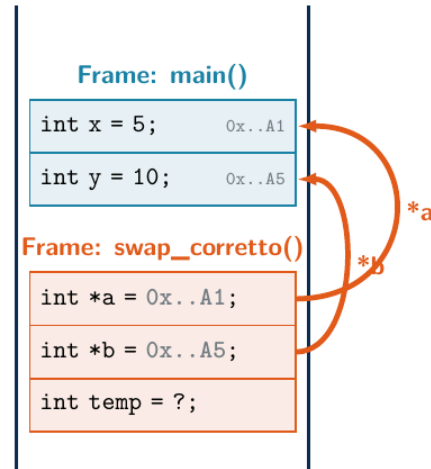
Swap per Riferimento nello Stack

`a` e `b` contengono gli **indirizzi** di `x` e `y`: come "bracci meccanici" oltrepassano il frame e modificano le variabili del `main`.

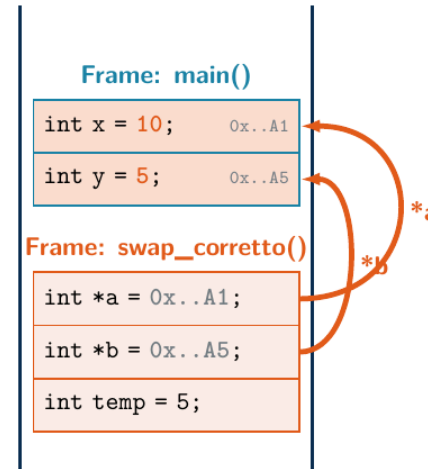
t_0 - Avvio



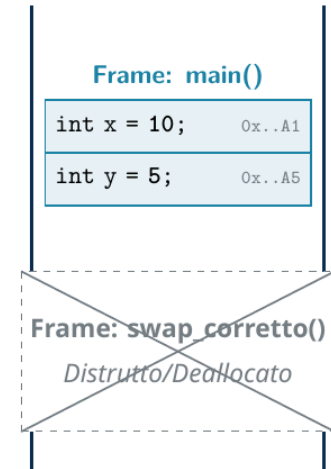
t_1 - Chiamata (&)



t_2 - Modifica Diretta (*)



t_3 - Dopo il return



Live: eseguire entrambe le versioni su Python Tutor e confrontare le frecce dei puntatori.



Equivalenza Fondamentale: Array e Puntatori

```
int arr[5] = {10, 20, 30, 40, 50};
```

- Il nome `arr` si comporta come un **puntatore costante al primo elemento**: `arr` $\equiv$ `&arr[0]`.
- La notazione con le parentesi quadre è aritmetica dei puntatori + dereferenziazione:

```
arr[i]  $\iff$  *(arr + i)
```

- Possiamo scorrere un array con l'indice `[i]` **oppure** manipolando direttamente l'indirizzo.

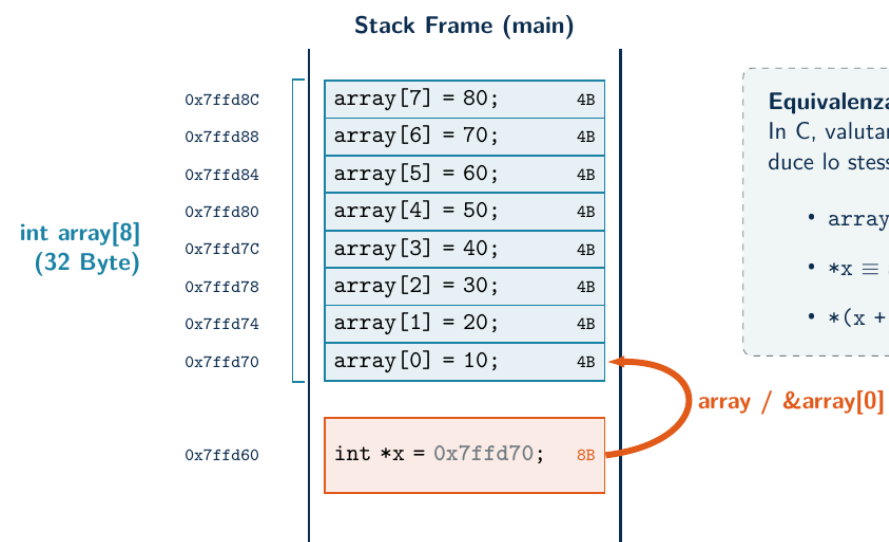
? Nel laboratorio abbiamo passato array alle funzioni **senza** `&`, eppure la funzione poteva modificarli. Perché?

L'Array nel suo Stack Frame

```
#include <stdio.h>

int main(void) {
    // 8 interi contigui (32 Byte)
    int array[8] = {10, 20, 30, 40,
                    50, 60, 70, 80};
    int *x = array; // indirizzo di partenza

    // array = &array[0]
    printf("Indirizzo array: %p\n",
           (void*)array);
    printf("Indirizzo primo elemento: %p\n",
           (void*)&array[0]);
    printf("Valore puntato da x: %d\n", *x);
    return 0;
}
```



Equivalenza Architetturale:

In C, valutare l'espressione `array` produce lo stesso risultato di `&array[0]`.

- `array` $\equiv$ `0x7ffd70`
- `*x` $\equiv$ `array[0]` (10)
- `*(x + 1)` $\equiv$ `array[1]` (20)

- I due indirizzi stampati coincidono; `*x` stampa 10.

Il Decadimento degli Array (Array Decay)

- Passato a una funzione, un array **non viene copiato**: decade a un puntatore al primo elemento.
- `int arr[]` o `int arr[100]` come parametro diventano `int *arr` ➡ la funzione **perde la dimensione**.

```
#include <stdio.h>

void stampa_dimensione(int arr[], int dim) {
    // arr è diventato un semplice int *: sizeof(arr) = dimensione di un puntatore
    printf("In funzione: sizeof(arr) = %lu byte\n", sizeof(arr));
}

int main(void) {
    int voti[5] = {18, 22, 25, 28, 30};
    // voti è un array di 5 interi: 5 * sizeof(int) = 20 byte
    printf("Nel main: sizeof(voti) = %lu byte\n", sizeof(voti));
    stampa_dimensione(voti, 5);
    return 0;
}
```

Array Decay: la Regola d'Oro

```
Nel main: sizeof(voti) = 20 byte  
In funzione: sizeof(arr) = 8 byte
```

- In C è **sempre necessario** passare la dimensione (`int dim`) a ogni funzione che elabora un array.
- Il `100` in `void elabora(int arr[100])` viene **ignorato** dal compilatore.

 **Live:** compilare con `gcc -Wall decay.c -o decay` e leggere il warning di gcc: `'sizeof' on array function parameter 'arr' will return size of 'int *'`. Poi mostrare su Python Tutor che nel frame di `stampa_dimensione` c'è solo una freccia.



Aritmetica dei Puntatori: il Passo

L'aritmetica è **tipizzata**: `p + 1` avanza di `sizeof(tipo)` byte, non di 1 byte.

Interi (`sizeof(int) = 4`):

```
int i[4] = {10, 20, 30, 40};
int *p = i; // es. indirizzo 1000

printf("p      : %p\n", (void*)p);
// 1000
printf("p+1    : %p\n", (void*)(p + 1));
// 1004
```

Reali (`sizeof(double) = 8`):

```
double d[4] = {1.5, 2.5, 3.5, 4.5};
double *q = d; // es. indirizzo 1000

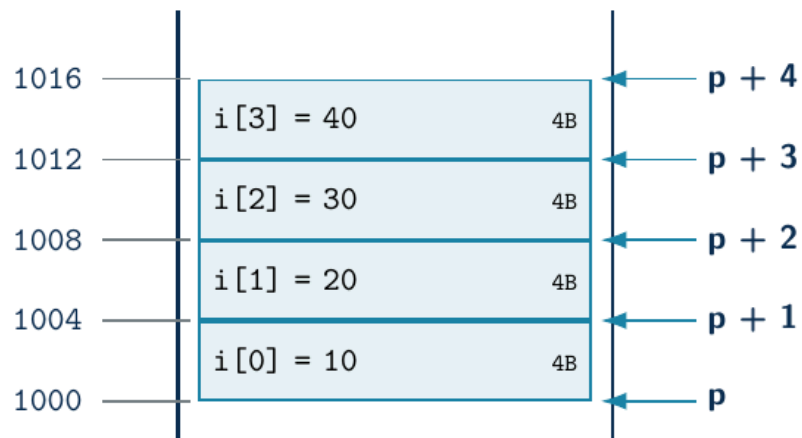
printf("q      : %p\n", (void*)q);
// 1000
printf("q+1    : %p\n", (void*)(q + 1));
// 1008
```

? Ecco perché un `int*` è diverso da un `double*`: quanti byte "salta" `p + 2`? E `q + 2`?

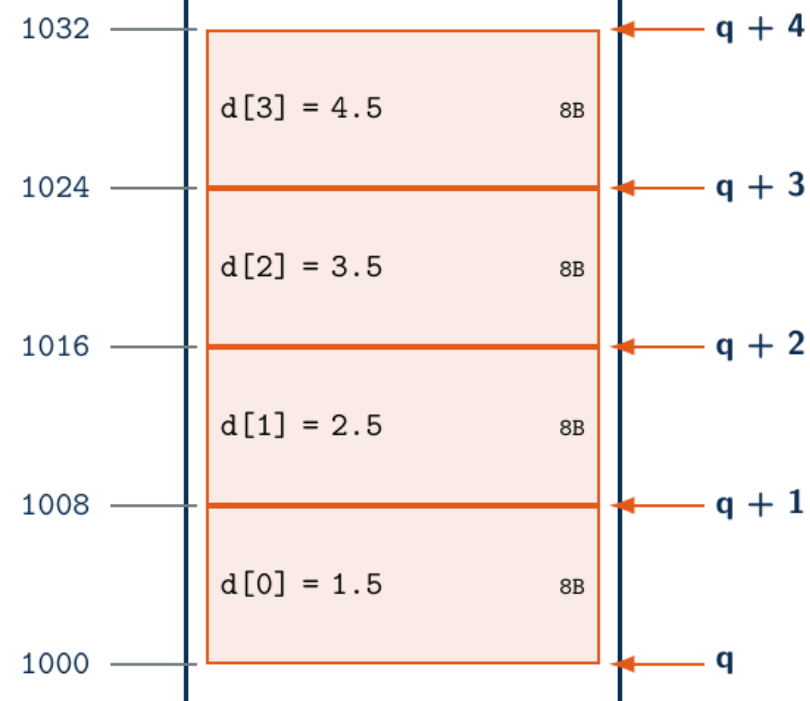
Rappresentazione del Passo dell'Aritmetica

- `int *: p + 2 ➡ +8 byte`; `double *: q + 2 ➡ +16 byte`. La CPU calcola `indirizzo + indice × sizeof(tipo)`.
- Accesso velocissimo: un **offset** seguito da una dereferenziazione.

Aritmetica su int (Passo = 4 Byte)



Aritmetica su double (Passo = 8 Byte)



Un Array è un Puntatore... Costante

- Il nome dell'array è un puntatore al primo elemento, ma **costante**: l'indirizzo base è fissato all'allocazione.

```
int x = 42;  
int arr[5];  
arr = &x; // ERRORE DI COMPILAZIONE! arr equivale a un int * const
```

? Passare un array a una funzione le dà il potere di modificarlo. E se vogliamo che la funzione possa solo **leggerlo**?

Proteggere i Dati in Lettura con `const`

```
void stampa_array(const int *arr, int dim) {  
    // arr[0] = 99; // ERRORE DI COMPILAZIONE! Il compilatore impedisce la scrittura  
}
```

Dichiarazione	Puntatore	Dato puntato
<code>const int *p</code>	può cambiare	costante (sola lettura)
<code>int * const p</code>	costante (come un array)	modificabile
<code>const int * const p</code>	costante	costante

 **Live:** scommentare `arr[0] = 99;`, compilare con `-Wall` e leggere l'errore (assignment of read-only location).

Funzioni `inline`

- Ogni chiamata a funzione ha un **overhead**: creazione e distruzione del record di attivazione nello stack.
- Per funzioni piccolissime chiamate spesso (es. dentro cicli intensivi) il costo del "viaggio" può superare quello del calcolo.
- `inline` **suggerisce** al compilatore di sostituire la chiamata con il corpo della funzione:

```
static inline int quadrato(int x) {  
    return x * x;  
}
```

- Si elimina l'overhead senza perdere modularità e leggibilità.

⚡ Caso 1: static inline senza Ottimizzazioni

```
#include <stdio.h>

static inline int add(int x, int y) {
    return x + y;
}

int main(void) {
    int x = 3 + add(5, 5);
    return x;
}
```

- Senza flag di ottimizzazione, il compilatore può **comunque** generare una chiamata standard.
- `static` serve: in C99 `inline` da solo non genera la funzione e senza `-O2` il linker dà `undefined reference to 'add'`.

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     dword ptr [rbp - 4], 0
    mov     esi, 5
    mov     edi, esi
    call    add      ; la call resta
    add     eax, 3    ; somma 3 al risultato
    mov     dword ptr [rbp - 8], eax
    mov     eax, dword ptr [rbp - 8]
    add     rsp, 16
    pop     rbp
    ret
```

Caso 2: Funzione Classica (senza inline)

```
int add(int x, int y) {  
    return x + y;  
}  
  
int main(void) {  
    int x = 3 + add(5, 5);  
    return x;  
}
```

```
add:  
    push    rbp  
    mov     rbp, rsp  
    mov     dword ptr [rbp - 4], edi ; 1° param  
    mov     dword ptr [rbp - 8], esi ; 2° param  
    mov     eax, dword ptr [rbp - 4]  
    add     eax, dword ptr [rbp - 8] ; somma  
    pop     rbp  
    ret
```

```
main:  
    push    rbp  
    mov     rbp, rsp  
    sub     rsp, 16  
    mov     dword ptr [rbp - 4], 0  
    mov     esi, 5  
    mov     edi, esi  
    call    add ; chiama direttamente add  
    add     eax, 3 ; somma 3  
    mov     dword ptr [rbp - 8], eax  
    mov     eax, dword ptr [rbp - 8]  
    add     rsp, 16  
    pop     rbp  
    ret
```

- Codice di `add` generato a parte e invocato con `call`.

inline con le Ottimizzazioni (-O2)

- Con le ottimizzazioni attive (es. -O2), il codice di `add` viene **iniettato** in `main`.
- L'intera espressione diventa un unico calcolo immediato (`mov eax, 13 + ret`): spariscono sia la `call` sia lo stack frame di `add`.

 **Live:** su godbolt.org incollare il Caso 1 e confrontare l'Assembly con -O0 e con -O2 (cercare la `call`).

Nota: provare anche a togliere `static:` con `gcc -Wall inline.c` (senza -O2) il linker dà `undefined reference to 'add'`, perché in C99 `inline` da solo non genera la funzione.

Cosa Abbiamo Imparato Oggi

- Un puntatore contiene un **indirizzo**: occupa sempre 8 byte a 64 bit; il tipo dice come **interpretare** la memoria.
- Inizializzare **sempre** i puntatori non associati a `NULL` e controllarli prima dell'uso.
- `&` estrae l'indirizzo, `*` legge/scrive il contenuto; `-g` + LLDB per trovare i crash.
- Passaggio per riferimento simulato: `swap_corretto(&x, &y)`.
- `arr` $\equiv$ `&arr[0]`, `arr[i]` $\equiv$ `*(arr + i)`; passato a funzione l'array **decade** (serve `dim`).
- Aritmetica a passi di `sizeof(tipo)`; `const int *` per i parametri in sola lettura; `inline` contro l'overhead.

 **Per casa:** esercizi della Lezione 8 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627