



Programmazione 1 (Canale O-Z)

Lezione 14: La Fuga dallo Stack (Memoria Dinamica e la nascita del Nodo)

Docente: Prof. Antonino Furnari

Email: antonino.furnari@unict.it

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

Prigionieri dello Stack

- Finora tutte le nostre variabili vivevano nella memoria **automatica** dello Stack: nascono e muoiono con lo scope della funzione.
- Oggi evadiamo nello **Heap**:
 - `malloc`, `calloc`, `free`;
 - ridimensionare con `realloc`;
 - il **Nodo autoreferenziale**: dati non più contigui.



! La Sfida: Restituire un Array

Una funzione che generi i primi n numeri di Fibonacci e li **restituisca** al chiamante:

```
int* genera_fibonacci_sbagliato(int n) {  
    // Array locale nello Stack (Variable Length Array - VLA)  
    int seq[n];  
  
    seq[0] = 0;  
    if (n > 1) seq[1] = 1;  
  
    for (int i = 2; i < n; i++) {  
        seq[i] = seq[i-1] + seq[i-2];  
    }  
  
    return seq; // Ritorno dell'indirizzo del primo elemento  
}
```

? Cosa succede quando proviamo a compilarla ed eseguirla dal `main`?

Il Compilatore si Ribella

```
gcc -Wall fib_err.c -o fib_err
```

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

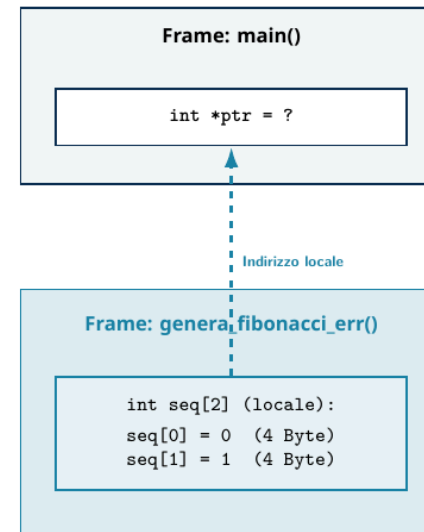
- All'esecuzione: crash immediato (`Segmentation fault`) o **valori spazzatura**.
- Ecco perché compiliamo **sempre con** `-Wall`: il warning ci avvisa prima del disastro.

 **Live:** compilare con `-Wall`, leggere il warning, poi eseguire `./fib_err` stampando l'array nel `main` e osservare il risultato.

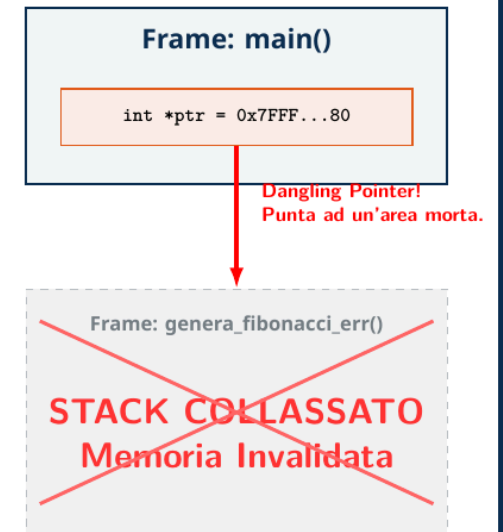
Il Collasso dello Stack: Dangling Pointer

- Durante la chiamata (t_0): `seq` vive nello Stack Frame di `genera_fibonacci_sbagliato`.
- Dopo il `return` (t_1): il frame viene rimosso e invalidato.
- L'indirizzo restituito punta ad un'area non più riservata: un **Dangling Pointer**, pronto ad essere sovrascritto da altre chiamate.
- Tutto ciò che nasce nello Stack **muore** nello Stack.

Stato ad inizio chiamata (t_0)



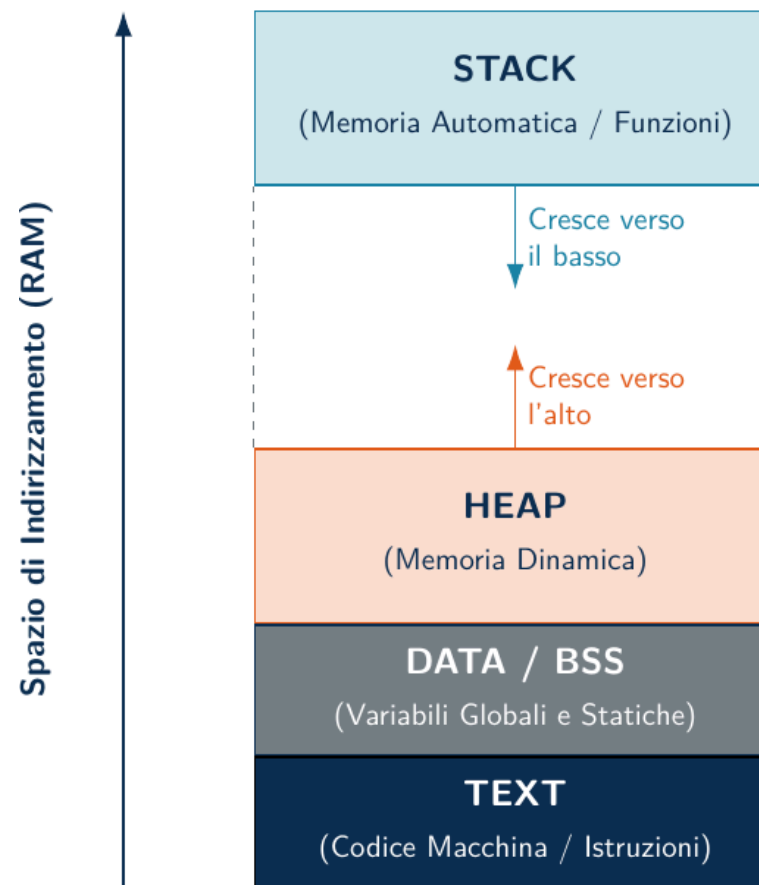
Dopo il return (t_1)





La Fuga: La Memoria HEAP

- Non segue la logica **LIFO** dello Stack: i blocchi si allocano dove c'è spazio.
- **Ampiezza**: limitata solo dalla RAM fisica complessiva del sistema.
- **Persistenza**: sopravvive alla chiusura delle funzioni, fino a distruzione esplicita o fine del processo.
- **Manuale**: la gestione è interamente affidata allo sviluppatore.



⚙ Le Chiavi dello Heap: `malloc`

```
#include <stdlib.h>

void* malloc(size_t size);
```

- Riceve il **numero esatto di byte** da allocare: si usa sempre `sizeof` (es. `n * sizeof(int)`).
- Restituisce un puntatore generico `void*` all'inizio del blocco nello Heap.
- Se l'allocazione fallisce (es. RAM esaurita) restituisce `NULL` ➡ va **sempre** controllato.

? Come riscriviamo `genera_fibonacci` in modo che l'array sopravviva al `return`?

Fibonacci nello Heap

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

int* genera_fibonacci(int n) {
    // Chiediamo spazio nell'Heap per n interi
    int *seq = (int*) malloc(n * sizeof(int));

    if (seq == NULL) {
        fprintf(stderr, "Errore: Memoria insufficiente!\n");
        return NULL;
    }

    seq[0] = 0;
    if (n > 1) seq[1] = 1;

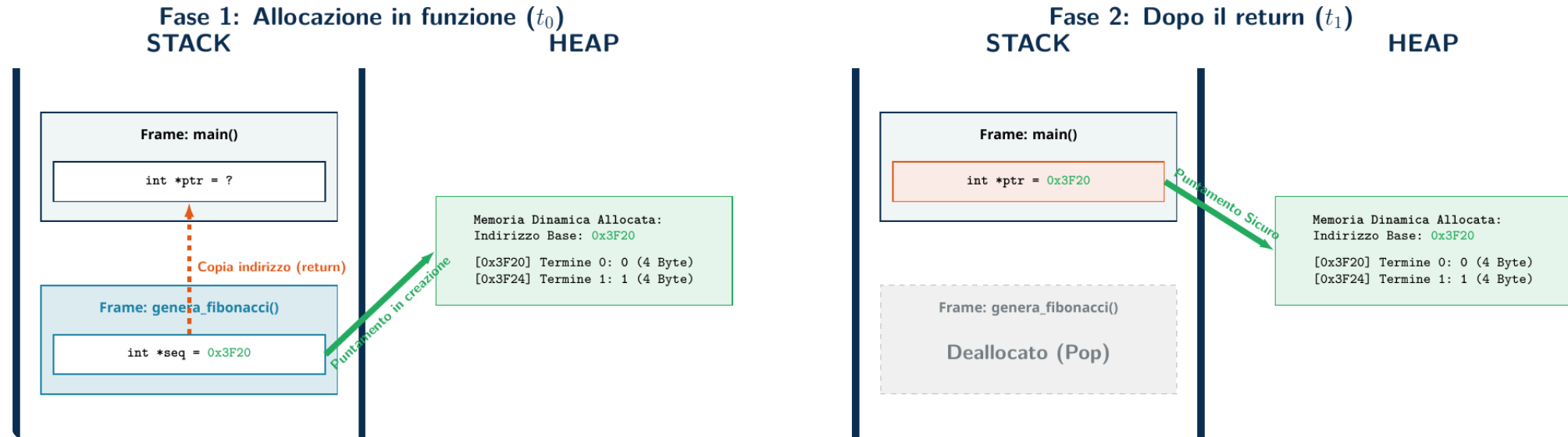
    for (int i = 2; i < n; i++) {
        seq[i] = seq[i-1] + seq[i-2];
    }

    return seq; // Perfettamente sicuro: l'Heap non collassa al return!
}
```



Ciclo di Vita nello Heap

- **Fase 1 (t_0):** il puntatore locale `seq` (nello Stack) contiene l'indirizzo del blocco creato nello Heap.
- **Fase 2 (t_1):** il frame della funzione si svuota, ma il blocco resta: il puntatore del `main` ne eredita l'indirizzo.





Restituire la Memoria: `void free(void* ptr)`

```
int main(void) {
    int n = 8;
    int *fib = genera_fibonacci(n);

    if (fib != NULL) {
        for (int i = 0; i < n; i++) {
            printf("%d ", fib[i]);
        }
        printf("\n");

        free(fib);    // Riconsegniamo i byte allo Heap
        fib = NULL;   // Annulliamo il puntatore per sicurezza
    }
    return 0;
}
```

 **Live:** `gcc -Wall fib.c -o fib && ./fib` ➡ nessun warning, stampa 0 1 1 2 3 5 8 13.

! Il Problema del Memory Leak

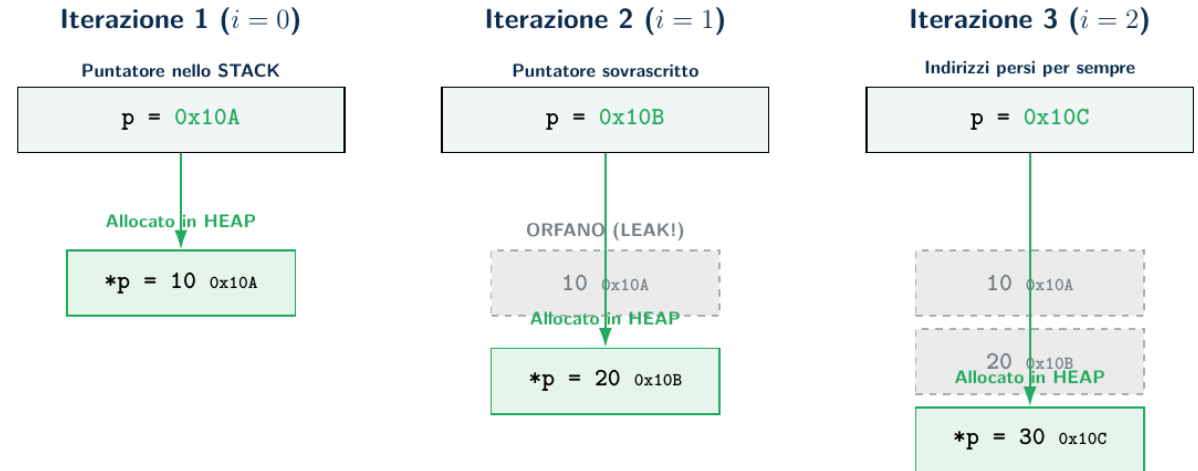
- Lo Heap **non si pulisce da solo** durante l'esecuzione.
- Perdere l'indirizzo di un blocco senza aver chiamato `free` crea un **Memory Leak**: la memoria resta bloccata e indisponibile.
- Una `malloc` dentro un ciclo da milioni di iterazioni senza `free` satura la RAM: il Sistema Operativo uccide il processo (**OOM Killer**).
- **Best practice**: dopo `free(p)`, impostare `p = NULL` per non riutilizzare per sbaglio una zona liberata (Dangling Pointer).

? Cosa succede al blocco precedente se riassegniamo `p = malloc(...)` a ogni iterazione senza `free`?



Visualizzazione: Effetto del Memory Leak

- Iterazione 1 ($i = 0$): `p` punta a `0x10A`.
- Iterazione 2 ($i = 1$): `p` viene sovrascritto con `0x10B` senza liberare `0x10A`, che diventa **orfano**.
- Iterazione 3 ($i = 2$): `p` si sposta su `0x10C`. Sia `0x10A` che `0x10B` sono bloccati e inaccessibili.



⚙️ Array Dinamici: Stessa Sintassi

```
int *arr = (int*) malloc(5 * sizeof(int));  
arr[0] = 10;  
*(arr + 1) = 20; // Equivalente a arr[1] = 20;
```

- Un array dinamico si usa **esattamente** come uno statico.
- L'operatore `[]` compie implicitamente l'aritmetica dei puntatori e la dereferenziazione.
- Cambia solo il **luogo di nascita** dei dati: Stack (`int arr[5]`) vs Heap (`malloc`).

? Cosa contengono `arr[2]`, `arr[3]` e `arr[4]` subito dopo la `malloc`?

`calloc` : Allocazione con Azzeramento

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

```
// Alloca 10 interi e li inizializza tutti a 0  
int *vettore = (int*) calloc(10, sizeof(int));
```

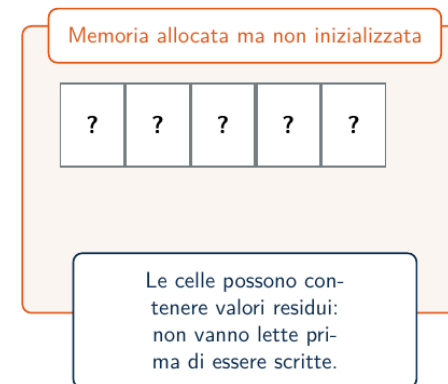
- `malloc` lascia intatto il contenuto preesistente della memoria (**spazzatura**).
- `calloc` riceve **numero di elementi** e **dimensione** di ciascuno, e azzerava tutti i bit.
- Un piccolo costo in più per la pulizia, ma più sicura per contatori, statistiche e buffer.



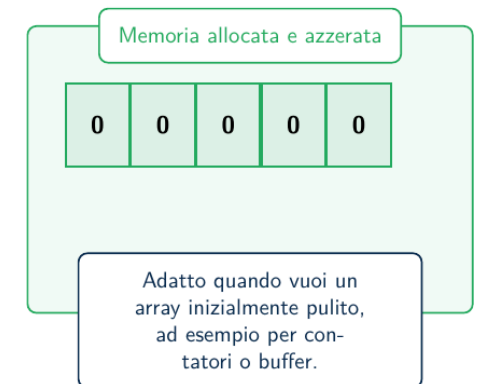
Confronto Fisico: malloc vs calloc

- **malloc**:
 - Riserva la dimensione richiesta lasciando inalterato il vecchio contenuto della RAM (?).
 - Richiede attenzione: bisogna scrivere prima di leggere.
- **calloc**:
 - Alloca e pulisce a zero ogni byte.

`malloc(5 * sizeof(int))`



`calloc(5, sizeof(int))`



Ridimensionare: `realloc`

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

- **Problema:** abbiamo n numeri di Fibonacci e vogliamo aggiungere l' $(n + 1)$ -esimo. Il blocco ha esattamente n posti.
- Tre possibili esiti:
 - i. **Espansione in situ:** lo spazio adiacente è libero ➡ il blocco si allunga senza spostarsi.
 - ii. **Trasloco:** spazio adiacente occupato ➡ nuova area altrove, copia di tutto il contenuto ($O(N)$), liberazione automatica del vecchio blocco, nuovo indirizzo.
 - iii. **Fallimento:** RAM insufficiente ➡ `NULL`, il blocco originale resta **valido e intatto**.



realloc: lo Schema Corretto

```
// ERRORE: fib = realloc(fib, ...); se fallisce, fib = NULL e i dati sono persi (leak)!

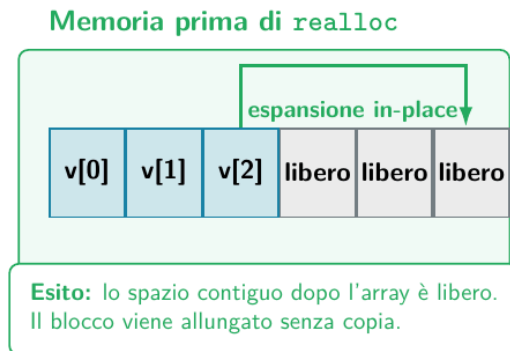
// Corretto schema d'uso di realloc
int *nuovo = (int*) realloc(fib, (n + 1) * sizeof(int));
if (nuovo != NULL) {
    fib = nuovo;
} else {
    // Gestione errore: fib contiene ancora i dati precedenti!
}
```

- Mai sovrascrivere direttamente il puntatore originale senza controllare il risultato.
- Riallocare un record alla volta su milioni di elementi: la CPU passa il tempo a **traslocare byte**.

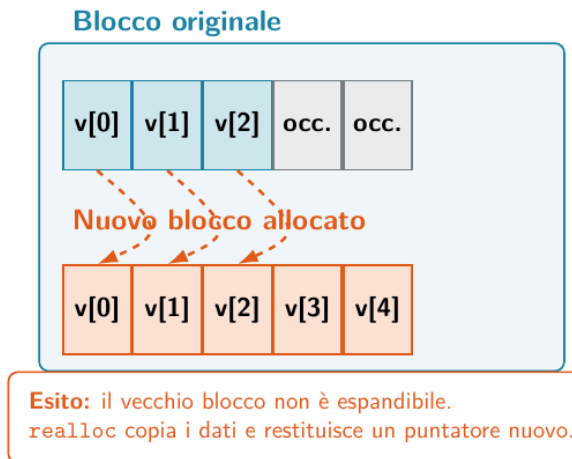


I Tre Scenari della `realloc`

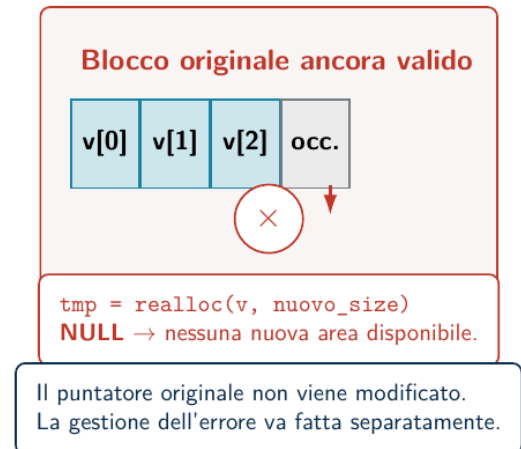
Scenario A: Espansione in avanti



Scenario B: Copia su nuovo blocco



Scenario C: Fallimento



? Perché la `realloc` è costretta a copiare tutto? Cosa impone un array alle sue celle?

Sganciare la Memoria

- **Il vincolo degli array:** le celle devono essere **contigue** ➡ inserimenti e riallocazioni costosi.
- **L'alternativa:** rinunciare alla contiguità e spargere i dati in zone distinte dello Heap.
- **La caccia al tesoro:** ogni "scatola" contiene, oltre al dato utile, un **biglietto con le coordinate** (puntatore) della scatola successiva.
- Questa scatola è l'unità fondamentale delle strutture dinamiche: il **Nodo**.

La Struct Autoreferenziale (il Nodo)

```
typedef struct Nodo {  
    int dato;           // Il valore utile (Payload)  
    struct Nodo *prossimo; // Il puntatore al prossimo nodo  
} Nodo;
```

- La struct contiene un **puntatore a una struct del suo stesso tipo**.
-  Dentro le graffe l'alias `Nodo` **non esiste ancora**: il compilatore non ha finito di leggere il `typedef`.
- Bisogna usare obbligatoriamente il nome completo `struct Nodo *prossimo`.

Passo 1: Allocazione dei Nodi

```
Nodo *testa = (Nodo*) malloc(sizeof(Nodo));  
Nodo *secondo = (Nodo*) malloc(sizeof(Nodo));  
Nodo *terzo = (Nodo*) malloc(sizeof(Nodo));
```

```
testa    —> [ dato: ? | prossimo: ? ]  
secondo —> [ dato: ? | prossimo: ? ]      (tre blocchi sparsi nello Heap)  
terzo    —> [ dato: ? | prossimo: ? ]
```

- Tre blocchi **indipendenti**: nessun legame tra loro, contenuto non inizializzato.

Passo 2: Scrittura dei Dati

```
testa→dato = 10;  
secondo→dato = 20;  
terzo→dato = 30;
```

```
testa    —> [ dato: 10 | prossimo: ? ]  
secondo —> [ dato: 20 | prossimo: ? ]  
terzo    —> [ dato: 30 | prossimo: ? ]
```

- Accediamo ai campi con `→`: `testa`, `secondo`, `terzo` sono **puntatori** a struct.

Passo 3: Collegamento dei Nodi

```
testa→prossimo = secondo;  
secondo→prossimo = terzo;  
terzo→prossimo = NULL; // Fine della catena
```

```
testa —> [ 10 | ● ] —> [ 20 | ● ] —> [ 30 | NULL ]
```

- `NULL` è la **sentinella** che segna la fine della catena.
- Navigazione: `testa→prossimo→dato` vale `20`.

? Una volta finito, quante `free` servono? Una per ogni `malloc`!

Il Programma Completo

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

typedef struct Nodo {
    int dato;
    struct Nodo *prossimo;
} Nodo;

int main(void) {
    Nodo *testa = (Nodo*) malloc(sizeof(Nodo));
    Nodo *secondo = (Nodo*) malloc(sizeof(Nodo));
    Nodo *terzo = (Nodo*) malloc(sizeof(Nodo));

    testa->dato = 10; secondo->dato = 20; terzo->dato = 30;

    testa->prossimo = secondo;
    secondo->prossimo = terzo;
    terzo->prossimo = NULL; // Fine della catena

    printf("Secondo dato: %d\n", testa->prossimo->dato); // 20

    free(testa); free(secondo); free(terzo); // Liberazione manuale
    return 0;
}
```

Eseguiamo

```
gcc -Wall nodi.c -o nodi && ./nodi
```

```
Secondo dato: 20
```

- Tre `malloc` ➡ tre `free`: ogni blocco va restituito esplicitamente.

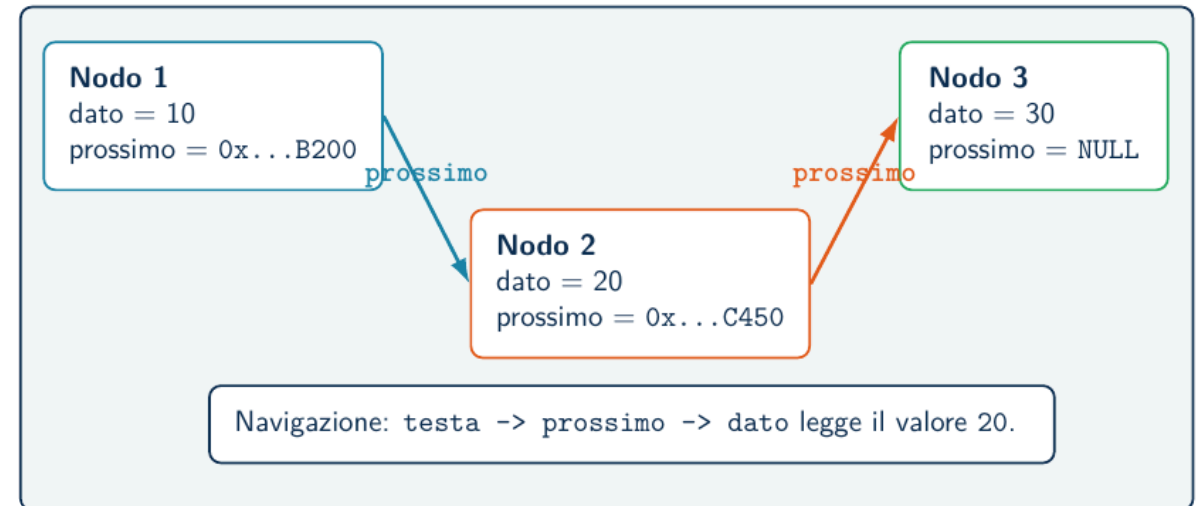
 **Live:** stampare gli indirizzi dei nodi (`printf("%p\n", (void*)secondo);`) per vedere che non sono per forza contigui. Poi togliere una `free`: il programma funziona lo stesso, ma quel blocco è un **leak**.



Heap: Nodi Sparsi e Collegati

- I tre nodi risiedono in indirizzi **non consecutivi** dello Heap.
- La connessione logica avviene interamente tramite i puntatori nel campo `prossimo`.
- Per inserire un nuovo elemento nel mezzo non si sposta alcun byte: basta aggiornare le frecce!
- **Prossima lezione:** cicli per scorrere, inserire e rimuovere nodi ➡ le **Liste Concatenate**.

Heap: nodi sparsi e collegati



✓ Cosa Abbiamo Imparato Oggi

- Non si può restituire l'indirizzo di una variabile locale: lo Stack Frame collassa (**Dangling Pointer**).
- Lo **Heap** è persistente e manuale: `malloc` (controllare `NULL`!) e `free` (poi `p = NULL`).
- Dimenticare la `free` causa **Memory Leak**.
- `calloc` alloca e azzerà; `realloc` ridimensiona (usare un puntatore temporaneo).
- Il **Nodo autoreferenziale** (`struct Nodo *prossimo`) svincola i dati dalla contiguità.

 **Per casa:** esercizi della Lezione 14 sul sito del corso.

Grazie per l'attenzione!

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