

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



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

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

Nelle lezioni precedenti abbiamo visto come aggregare dati eterogenei usando le *struct* e come gestirli in blocchi fissi. Tuttavia, siamo sempre rimasti vincolati alla memoria *automatica* dello Stack, in cui le variabili nascono e muoiono rigidamente entro lo scope della funzione corrente.

In questa lezione compiremo un salto fondamentale: impareremo ad evadere dallo Stack allocando manualmente la memoria in un'area persistente controllata direttamente dal programmatore, lo **Heap**. Utilizzeremo le funzioni standard `malloc`, `calloc` e `free` per richiedere e rilasciare RAM, analizzeremo il funzionamento e i costi della riallocazione (`realloc`), e infine getteremo le basi per le strutture dati dinamiche definendo il **Nodo autoreferenziale**, l'unità fondamentale che ci consentirà di slegare i dati dalla contiguità fisica degli array.

1.1 Il Muro dello Stack e l'Ingresso nello Heap

1.1.1 Il problema irrisolvibile: Variabili Locali e Dangling Pointer

Si consideri il tentativo di scrivere una funzione che generi un array contenente i primi n numeri della successione di Fibonacci e lo restituisca al chiamante:

```
1  int* genera_fibonacci_sbagliato(int n) {  
2      // Array locale nello Stack (Variable Length Array - VLA)  
3      int seq[n];  
4  
5      seq[0] = 0;  
6      if (n > 1) seq[1] = 1;  
7  
8      for (int i = 2; i < n; i++) {  
9          seq[i] = seq[i-1] + seq[i-2];  
10     }  
11  
12     return seq; // Ritorno dell'indirizzo del primo elemento  
13 }
```

Compilando questo codice, il compilatore emette un avviso critico: `warning: function returns address of local variable [-Wreturn-local-addr]`

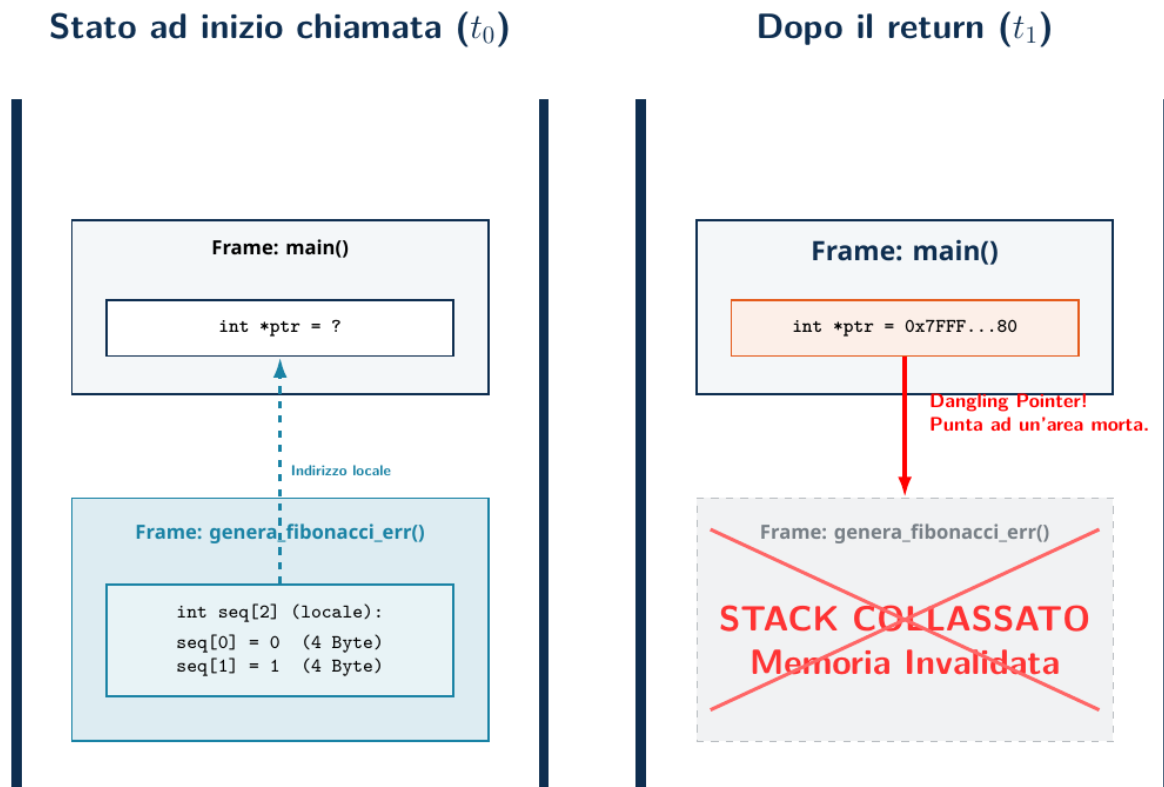


Figura 1.1: Collasso dello Stack

All'esecuzione, il programma subirà un crash immediato (`Segmentation fault`) o produrrà valori spazzatura.

La ragione fisica: L'array `seq` risiede nello Stack Frame di `genera_fibonacci_err()`. Al termine dell'esecuzione della funzione (`return`), il suo intero frame viene rimosso dallo Stack e invalidato dal sistema operativo. L'indirizzo restituito punta ad un'area non più riservata (un **Dangling Pointer**), pronta ad essere sovrascritta da altre chiamate a funzione.

Attenzione

Attenzione Tutte le variabili create nello Stack nascono e muoiono con lo scope della funzione in cui sono dichiarate. Non è possibile restituire puntatori a variabili locali o array locali dallo stack.

1.1.2 La Fuga: La Memoria HEAP

Per superare la logica LIFO (Last-In, First-Out) dello Stack, i sistemi operativi offrono l'area di memoria **Heap** (Mucchio):

- **Ampiezza:** È limitata solo dalla RAM fisica complessiva del sistema.
- **Persistenza:** La memoria allocata nello Heap sopravvive alla chiusura delle funzioni e rimane attiva fino a quando il programmatore non ne richiede esplicitamente la distruzione, o fino al termine dell'intero processo.

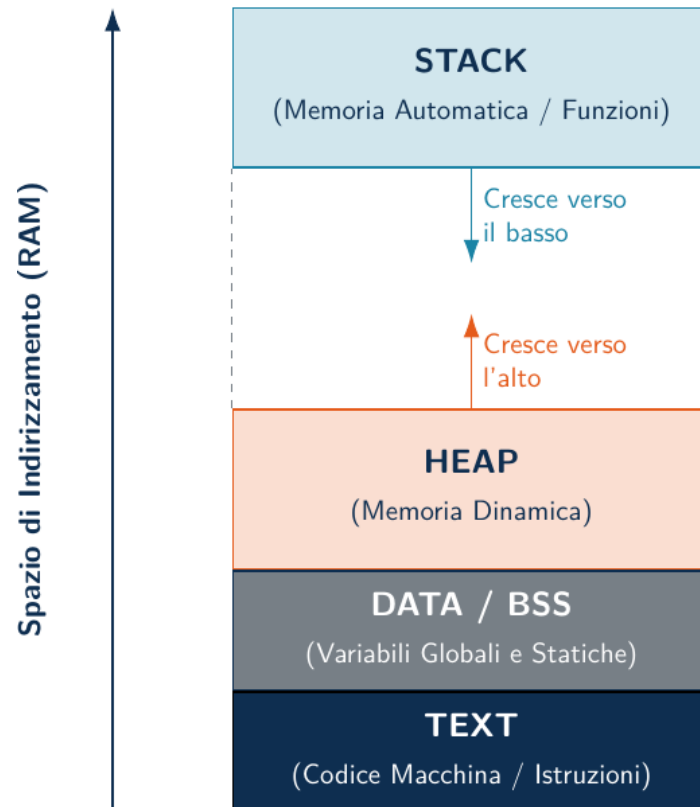


Figura 1.2: Stack e Heap

- **Manuale:** La gestione è interamente affidata allo sviluppatore.

1.1.3 Le Chiavi dello Heap: `malloc` e `free`

La libreria standard `<stdlib.h>` fornisce gli strumenti per dialogare con lo Heap.

Allocazione con `malloc`

La funzione `malloc` (Memory Allocation) richiede al sistema operativo un blocco continuo di byte:

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

- Riceve come parametro il numero esatto di byte da allocare.
- Restituisce un puntatore a tipo generico (`void*`) che punta all'inizio del blocco allocato nello Heap.
- Se l'allocazione fallisce (ad es. per mancanza di RAM), restituisce `NULL`.

Riscriviamo la generazione di Fibonacci in sicurezza:

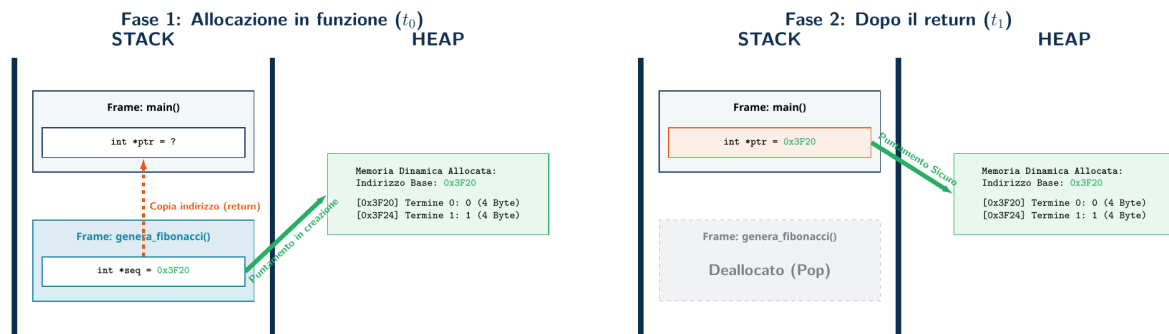


Figura 1.3: Ciclo di Vita Heap

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int* genera_fibonacci(int n) {
5      // Chiediamo spazio nell'Heap per n interi
6      int *seq = (int*) malloc(n * sizeof(int));
7
8      if (seq == NULL) {
9          fprintf(stderr, "Errore: Memoria insufficiente!\n");
10         return NULL;
11     }
12
13     seq[0] = 0;
14     if (n > 1) seq[1] = 1;
15
16     for (int i = 2; i < n; i++) {
17         seq[i] = seq[i-1] + seq[i-2];
18     }
19
20     return seq; // Perfettamente sicuro: l'Heap non collassa al return!
21 }

```

L'immagine che segue mostra come si può usare lo heap per costruire un nuovo array dentro una funzione e restituire un puntatore al chiamante:

Rilascio della memoria con free

Poiché la memoria Heap non viene pulita automaticamente, dimenticare di liberarla genera un **Memory Leak** (perdita di memoria). La memoria occupata rimane bloccata e indisponibile. Per liberare un blocco di memoria si usa `free`:

```

1 void free(void* ptr);

```

Dopo aver liberato un puntatore, è ottima norma impostarlo a `NULL` per evitare di riutilizzarlo accidentalmente (Dangling Pointer).

```

1 int main(void) {
2     int n = 8;
3     int *fib = genera_fibonacci(n);

```

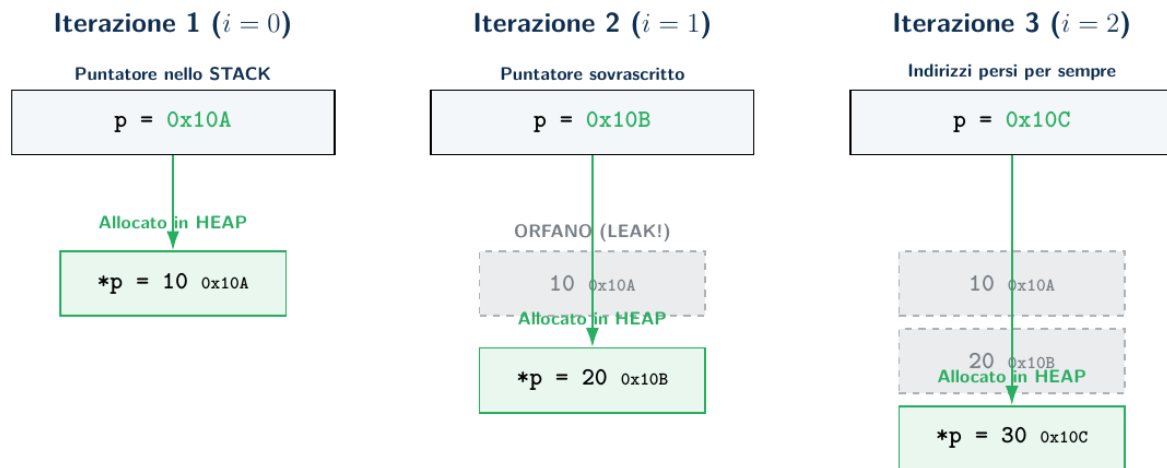


Figura 1.4: Memory Leak

```

4
5     if (fib != NULL) {
6         for (int i = 0; i < n; i++) {
7             printf("%d ", fib[i]);
8         }
9         printf("\n");
10
11         free(fib); // Riconsegniamo i byte allo Heap
12         fib = NULL; // Annulliamo il puntatore per sicurezza
13     }
14     return 0;
15 }

```

L'immagine che segue mostra un esempio di memory leak:

1.2 Array Dinamici e Ridimensionamento

1.2.1 Sintassi di Accesso negli Array Dinamici

Una volta allocato un array dinamico e assegnato l'indirizzo a un puntatore, l'accesso avviene con la medesima sintassi degli array statici:

```

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

```

L'operatore parentesi quadre [] compie implicitamente l'aritmetica dei puntatori ed esegue la dereferenziazione.

1.2.2 calloc: Allocazione con Inizializzazione a Zero

A differenza di `malloc`, che lascia intatto il contenuto preesistente della memoria (spazzatura), la funzione `calloc` pulisce a zero tutti i bit allocati:

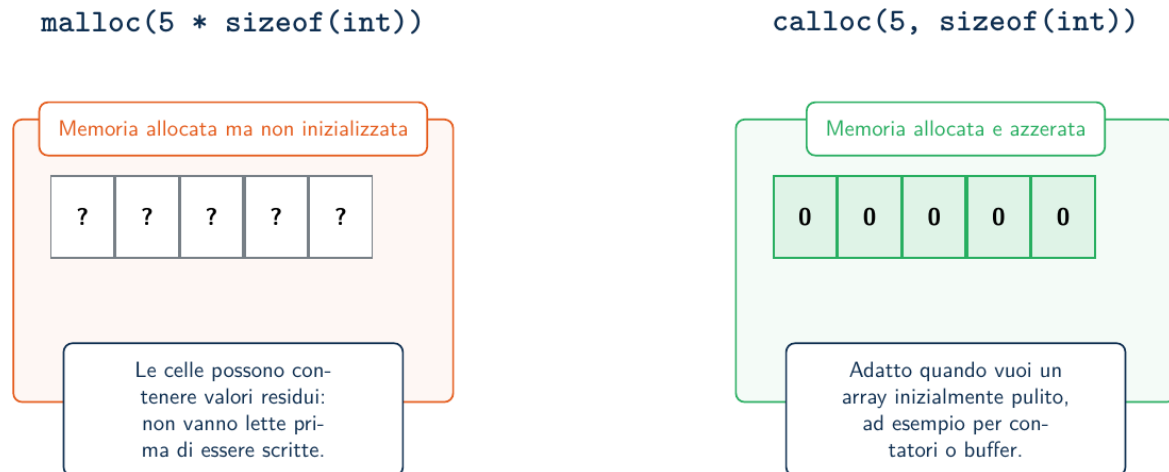


Figura 1.5: malloc vs calloc

```
1 void* calloc(size_t nmemb, size_t size);
```

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

Il diagramma seguente mostra le differenze fisiche nell'inizializzazione della memoria tra malloc e calloc:

1.2.3 Ridimensionare la memoria: realloc

Se la dimensione originaria di un array dinamico si rivela insufficiente, possiamo modificarne l'estensione tramite realloc:

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

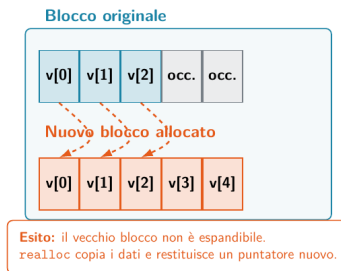
Meccanismo di funzionamento:

1. **Espansione in situ:** Se lo spazio contiguo immediatamente successivo nello Heap è libero, la realloc estende semplicemente il blocco originale senza spostarlo.
2. **Trasloco:** Se lo spazio adiacente è occupato, la realloc cerca una nuova area capiente altrove nello Heap, copia l'intero contenuto, dealloca il vecchio spazio in automatico e restituisce il nuovo indirizzo.
3. **Fallimento:** Se non c'è RAM sufficiente, restituisce NULL lasciando il puntatore originario valido e intatto.

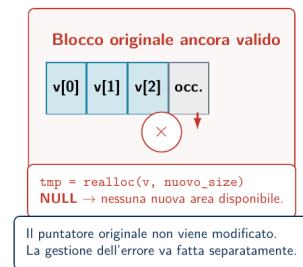
Scenario A: Espansione in avanti



Scenario B: Copia su nuovo blocco



Scenario C: Fallimento

Figura 1.6: Scenari `realloc`

Nota

Nota Per evitare perdite di memoria in caso di fallimento della `realloc`, non bisogna mai sovrascrivere direttamente il puntatore originale con il risultato della funzione senza prima controllare se è valido (NULL).

```

1 // Corretto schema d'uso di realloc
2 int *nuovo = (int*) realloc(fib, (n + 1) * sizeof(int));
3 if (nuovo != NULL) {
4     fib = nuovo;
5 } else {
6     // Gestione errore: fib contiene ancora i dati precedenti!
7 }

```

Il diagramma seguente illustra i tre diversi scenari che si possono verificare durante una chiamata a `realloc`:

1.3 Verso le Strutture Dinamiche: Il Nodo

1.3.1 Superare il vincolo della contiguità

Gli array in C devono essere allineati in celle di memoria contigue. Ciò rende costoso l'inserimento o la riallocazione in mezzo ai dati. L'alternativa consiste nel disporre i dati sparsi in zone distinte dello Heap, legandoli tra loro tramite puntatori, in una caccia al tesoro logica.

1.3.2 Il Nodo Autoreferenziale

L'elemento base di questa catena è la struct autoreferenziale, detta **Nodo**:

```

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

```

Attenzione

Attenzione Dentro la definizione della struct non è possibile usare l'alias `Nodo` in quanto non è ancora stato registrato dal compilatore. Bisogna usare obbligatoriamente la nomenclatura formale completa `struct Nodo *prossimo`.

1.3.3 Collegamento manuale di nodi

Il seguente codice mostra come allocare e collegare manualmente tre nodi nello Heap:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Nodo {
5      int dato;
6      struct Nodo *prossimo;
7  } Nodo;
8
9  int main(void) {
10     // 1. Allocazione dei nodi sparsi nello Heap
11     Nodo *testa = (Nodo*) malloc(sizeof(Nodo));
12     Nodo *secondo = (Nodo*) malloc(sizeof(Nodo));
13     Nodo *terzo = (Nodo*) malloc(sizeof(Nodo));
14
15     // 2. Assegnamento dei dati
16     testa->dato = 10;
17     secondo->dato = 20;
18     terzo->dato = 30;
19
20     // 3. Collegamento
21     testa->prossimo = secondo;
22     secondo->prossimo = terzo;
23     terzo->prossimo = NULL; // Fine della catena
24
25     // Navigazione
26     printf("Secondo dato: %d\n", testa->prossimo->dato); // 20
27
28     // Liberazione manuale
29     free(testa);
30     free(secondo);
31     free(terzo);
32
33     return 0;
34 }
```

Il diagramma seguente illustra la disposizione in memoria dei nodi nello Heap e il relativo collegamento tramite puntatori:

1.4 Autovalutazione ed Esercizi

Heap: nodi sparsi e collegati

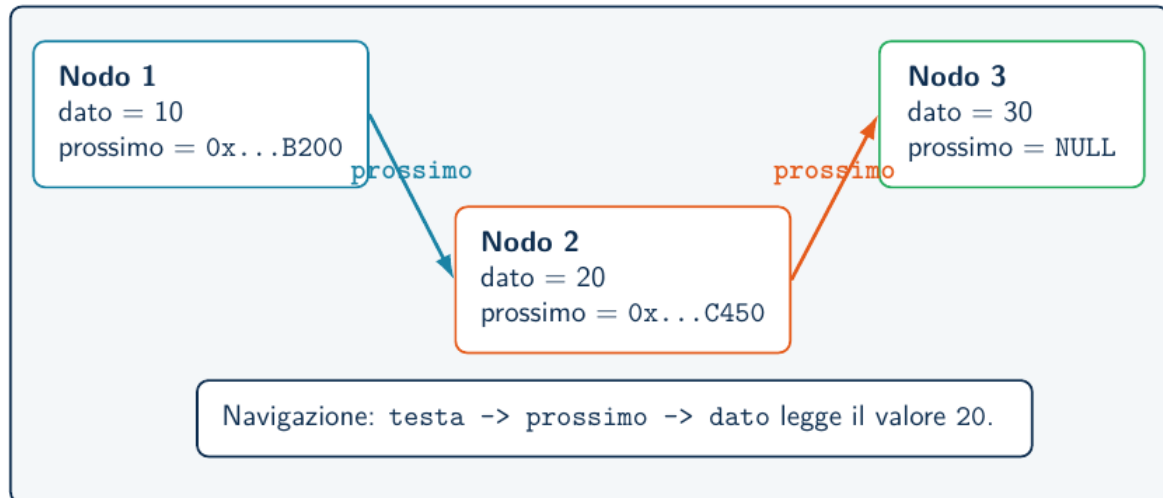


Figura 1.7: Collegamento dei Nodi

1.4.1 Test a Risposta Multipla (10 Quiz)

Quiz 1

Domanda

1. Cosa succede se una funzione restituisce l'indirizzo di una variabile locale allocata nello Stack?

- a) La variabile viene promossa automaticamente nello Heap.
- b) Si ottiene un dangling pointer che punta a una zona di memoria invalidata dal return.
- c) Il compilatore rifiuta di compilare bloccando l'esecuzione con errore bloccante.
- d) La variabile sopravvive fino al termine del main.

Soluzione
Risposta: b. Le variabili locali nello Stack cessano di esistere al return; il puntatore diventa dangling.

Quiz 2

Domanda

2. Quale delle seguenti affermazioni descrive correttamente la memoria Heap?

- a) È ordinata secondo la logica LIFO.
- b) Viene gestita automaticamente dal compilatore senza intervento umano.

- c) Consente allocazioni persistenti a vita manuale, fino alla chiamata di `free`.
- d) Ha dimensioni inferiori a quelle dello Stack di sistema.

Soluzione
Risposta: c. Lo Heap è governato interamente in modo manuale dal programmatore.

Quiz 3

Domanda

3. Cosa restituisce `malloc` in caso di fallimento dell'allocazione?

- a) Un puntatore a `0x0001`.
- b) Un valore casuale.
- c) `NULL`.
- d) Genera un'eccezione di runtime interrompendo il programma.

Soluzione
Risposta: c. Restituisce `NULL`, motivo per cui va sempre controllata.

Quiz 4

Domanda

4. Qual è la principale differenza tra `malloc` e `calloc`?

- a) `calloc` alloca solo memoria nello Stack.
- b) `calloc` alloca la memoria e azzerà il valore di tutti i bit.
- c) `malloc` accetta due parametri anziché uno.
- d) `calloc` non può fallire.

Soluzione
Risposta: b. `calloc` inizializza a zero, `malloc` lascia spazzatura.

Quiz 5

Domanda

5. Cosa si intende per "Memory Leak"?

- a) La perdita fisica del computer contenente i file sorgente.
- b) Una fuga di dati sensibili a causa di un puntatore non protetto.

- c) Memoria Heap allocata che non viene liberata con `free`, rimanendo inutilizzabile.
- d) Il collasso dello Stack Frame per ricorsione infinita.

Soluzione
Risposta: c. È l'accumulo di memoria allocata e mai rilasciata.

Quiz 6

Domanda

6. Come si comporta `realloc` se non trova spazio adiacente per allargare il blocco?

- a) Restituisce `NULL` e distrugge i vecchi dati.
- b) Sposta l'intero blocco in una nuova posizione nello Heap e copia i dati.
- c) Converte l'array in una lista concatenata in modo trasparente.
- d) Sospende l'esecuzione del programma fino a quando non si libera spazio.

Soluzione
Risposta: b. Eseguendo un "trasloco" dei dati in una nuova area Heap.

Quiz 7

Domanda

7. Perché nello schema `realloc` non si deve riassegnare direttamente il puntatore originale?

- a) Perché l'assegnamento diretto causerebbe un errore sintattico.
- b) Perché se la `realloc` fallisce e restituisce `NULL`, perderemmo l'indirizzo originale creando un memory leak.
- c) Perché il puntatore originale verrebbe invalidato in automatico dal compilatore.
- d) Perché la `realloc` richiede un cast esplicito a `void**`.

Soluzione
Risposta: b. Sovrascrivere con `NULL` rende inaccessibile il vecchio blocco.

Quiz 8

Domanda

8. Qual è la sintassi corretta per un puntatore autoreferenziale in una struct chiamata `Nodo`?

- a) `Nodo *prossimo;` all'interno della struct senza alias definito.
- b) `struct Nodo *prossimo;`
- c) `void *prossimo;` castato sempre a `struct Nodo`.
- d) `struct prossimo *Nodo;`

Soluzione
Risposta: b. Poiché l'alias `Nodo` non è ancora registrato, è necessario usare `struct`
`Nodo *`.

Quiz 9

Domanda

9. Che valore si assegna tipicamente al campo `prossimo` dell'ultimo nodo di una lista?

- a) `0x0001`.
- b) `EOF`.
- c) `NULL`.
- d) L'indirizzo del nodo di testa (sempre).

Soluzione
Risposta: c. `NULL` funge da terminatore.

Quiz 10

Domanda

10. A quale delle seguenti operazioni equivale sintatticamente `*(ptr + i)`?

- 1 - a) `'ptr->i'`
- 2
- 3 - b) `'ptr[i]'`
- 4
- 5 - c) `'&ptr[i]'`
- 6
- 7 - d) `'ptr.i'`

Soluzione
Risposta: b. L'accesso con parentesi quadre è pura aritmetica e dereferenziazione di puntatori.

1.4.2 Esercizi di Programmazione Progressivi

Esercizio 1: Allocazione di un Intero Singolo

Scrivere un programma che allochi dinamicamente memoria per un singolo intero nello Heap, vi assegni il valore 42, lo stampi, liberi la memoria e azzeri il puntatore.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int *p = malloc(sizeof(int));
7      if (p == NULL)
8          return 1;
9      *p = 42;
10     printf("Valore nello Heap: %d\n", *p);
11     free(p);
12     p = NULL;
13     return 0;
14 }

```

Soluzione

!

Esercizio 2: Vettore Dinamico Semplice

Scrivere una funzione `double* crea_vettore(int n)` che allochi un array di `n` double tramite `calloc`, inserisca in ciascuna cella il proprio indice e restituisca il puntatore. Testare la funzione nel main e liberare la memoria.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double* crea_vettore(int n)
5  {
6      double *arr = calloc(n, sizeof(double));
7      if (arr == NULL)
8          return NULL;
9      for (int i = 0; i < n; i++)
10         arr[i] = i;
11     return arr;
12 }

```

Soluzione

!

Esercizio 3: Stringa Dinamica da Input

Scrivere un programma che legga da tastiera un intero `lunghezza` e allochi dinamicamente una stringa (`char*`) della dimensione esatta (incluso il terminatore `\0`). Leggere la stringa con `scanf` e stamparla al contrario.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6     int len;
7     printf("Inserisci la lunghezza massima: ");
8     if (scanf("%d", &len) != 1)
9         return 1;
10    char *s = (char*) malloc((len + 1) * sizeof(char));
11    if (s == NULL)
12        return 1;
13    printf("Inserisci la stringa: ");
14    scanf("%s", s);
15    int reale = strlen(s);
16    for (int i = reale - 1; i >= 0; i--)
17        putchar(s[i]);
18    putchar("\n");
19    free(s);
20    return 0;
21 }

```

Soluzione

!

Esercizio 4: Prevenzione di Perdite con Malloc

Creare una funzione che allochi un array di 1000 interi. Simulare un fallimento inserendo intenzionalmente un ciclo che non libera la memoria. Aggiungere le istruzioni di rilascio necessarie per mantenere la stabilità del sistema ed evitare il leak.

```
int main(void) for (int i = 0; i < 10000; i++) alloca_err(lascaia()); // Senza free provocherebbe leak continuo printf("E
// Rilascio corretto free(temp);
void alloca_err(lascaia(void) int * temp = (int*) malloc(1000 * sizeof(int)); if(temp != NULL) // Operazioni fini
bi include <stdio.h> include <stdlib.h>
```

Soluzione

!

Esercizio 5: Espansione di un Buffer con Realloc

Creare un array dinamico di 3 interi, popolarlo con i valori 10, 20, 30. Ridimensionarlo a 5 elementi usando realloc e inserire nelle nuove posizioni 40, 50. Stampare l'array finale.

```
free(arr); return 0;
for (int i = 0; i < 5; i++) printf("%i", arr[i]);
nuovo; arr[3] = 40; arr[4] = 50;
int * nuovo = (int*) realloc(arr, 5 * sizeof(int)); if (nuovo == NULL) free(arr); return 1; arr =
arr[0] = 10; arr[1] = 20; arr[2] = 30;
int main(void) int * arr = (int*) malloc(3 * sizeof(int)); if (arr == NULL) return 1;
bi include <stdio.h> include <stdlib.h>
```

Soluzione

!

Esercizio 6: Stringa Dinamica Concatenata

Scrivere una funzione char* concatena(const char *s1, const char *s2) che allochi dinamicamente lo spazio strettamente necessario a ospitare la concatenazione di due stringhe, la esegua e restituisca il puntatore al main.

```
return 0;
int main(void) char * c = concatena("Hello, ", "World!"); if (c != NULL) printf("free(c); c = NULL;
strcat(res, s2); return res;
*res = (char*) malloc((l1 + l2 + 1) * sizeof(char)); if (res == NULL) return NULL; strcpy(res, s1);
char* concatena(const char *s1, const char *s2) int l1 = strlen(s1); int l2 = strlen(s2); char
bi include <stdio.h> include <stdlib.h> include <string.h>
```

Soluzione

!

Esercizio 7: Copia Dinamica di una Struct

Data una struct typedef struct \{\} char modello[20]; float prezzo; \{\} Auto;, scrivere una funzione Auto* duplica\{\}_auto(const Auto *originale) che al-

lochi nello Heap una copia identica della struct passata per riferimento e ne restituisca l'indirizzo.

```

        NULL); printf("Copia : free(d); return 0;
        int main(void) { Fiat Panda", 14500.0f, Auto * d = duplica_auto(mia); if(d)
        Auto * duplica_auto(const Auto * originale) { Auto * copia = (Auto *) malloc(sizeof(Auto)); if(copia == NULL) r
        typedef struct char modello[20]; float prezzo; Auto;
        b) include <stdio.h> include <stdlib.h> include <string.h>

```

Soluzione

!

Esercizio 8: Creazione di una Lista Dinamica di 3 Nodi

Scrivere un programma che definisca il tipo Nodo e popoli dinamicamente una catena di 3 nodi con valori inseriti dall'utente a runtime. Stampare la lista scorrendola tramite un puntatore temporaneo.

```

        free(testa); free(secondo); free(terzo); return 0;
        printf("curr = curr->prossimo; printf("NULL");
        // Scorrimento pulito con puntatore temporaneo Nodo * curr = testa; while (curr != NULL)
        testa->prossimo = secondo; secondo->prossimo = terzo; terzo->prossimo = NULL;
        printf("Inserisci tre interi:"); scanf(
        if (testa == NULL || secondo == NULL || terzo == NULL) return 1;
        malloc(sizeof(Nodo)); Nodo * terzo = (Nodo *) malloc(sizeof(Nodo));
        int main(void) { Nodo * testa = (Nodo *) malloc(sizeof(Nodo)); Nodo * secondo = (Nodo *)
        typedef struct Nodo int valore; struct Nodo * prossimo; Nodo;
        b) include <stdio.h> include <stdlib.h>

```

Soluzione

!

Esercizio 9: Matrice Lineare Dinamica (Row-Major Order)

Scrivere un programma che allochi un array monodimensionale nello Heap per simulare una griglia di righe e colonne (matrice linearizzata). Leggere righe R e colonne C, popolare la matrice con il prodotto degli indici $i * j$ e stamparla in forma tabellare.

```

        free(mat); return 0;
        for (int i = 0; i < R; i++) for (int j = 0; j < C; j++) printf("%d",
        for (int i = 0; i < R; i++) for (int j = 0; j < C; j++) mat[i * C + j] = i * j; // Formula Row-Major Order
        = (int *) malloc(R * C * sizeof(int)); if (mat == NULL) return 1;
        int main(void) { int R = 3, C = 4; // Linearizzazione: allocazione di R * C celle contigue int * mat
        b) include <stdio.h> include <stdlib.h>

```

Soluzione

!

Esercizio 10: Inserimento in Testa (Manuale)

Dato il puntatore iniziale `Nodo *testa` che punta a una catena esistente di 2 nodi (`10 -> 20 -> NULL`), implementare manualmente l'inserimento di un nuovo nodo di valore 5 in testa, aggiornando correttamente i puntatori. Stampare la nuova lista partendo da `testa`.

```

bi nclude<stdio.h> include<stdlib.h>
typedef struct Nodo {int valore; struct Nodo *prossimo; Nodo;
int main(void) // Lista di partenza: 10 -> 20 -> NULL
Nodo *n1 = (Nodo*) malloc(sizeof(Nodo));
Nodo *n2 = (Nodo*) malloc(sizeof(Nodo)); if (n1 == NULL || n2 == NULL) return 1; n1->valore = 10; n2->valore = 20; n1->prossimo = n2; n2->prossimo = NULL;
Nodo *testa = n1;
// NUOVO NODO DA INSERIRE IN TESTA
Nodo *nuovo = (Nodo*) malloc(sizeof(Nodo)); if
(nuovo == NULL) return 1; nuovo->valore = 5;
// INSERIMENTO: il nuovo nodo punta alla vecchia testa nuovo->prossimo = testa; // La testa
si sposta sul nuovo nodo testa = nuovo;
// Stampa finale per validazione
Nodo *temp = testa; while (temp != NULL) printf("temp =
temp->prossimo; printf("NULL");
// Liberazione di tutta la memoria free(n2); free(n1); free(nuovo);
return 0;

```

Soluzione