Esami Svolti – Programmazione 2 (C)

Autore: **Scarlata Davide**
Anno: **Primo anno**
Corso: **Linguaggio C – Programmazione 2**

> ⚠️ **Nota bene:**
> Le soluzioni non sono garantite corrette: **sono esercizi svolti da me** (studente) e **potrebbero contenere errori**.
> Questo materiale è pensato come supporto allo studio e confronto personale, **non come riferimento ufficiale**.

🔗 Collegamenti utili

- 🌐 Moodle corso C
- 📝 Piattaforma esami

📚 Come usare questa raccolta

- Puoi usare questi esercizi per **ripassare la teoria** attraverso il codice.
- Prova a **risolvere prima da solo** ogni esercizio, poi confronta con la mia soluzione.
- Se trovi un errore o vuoi suggerire una miglioria, **scrivimi** e ti fornirò il codice Markdown per effettuare un *push* della modifica.

🙏 Ringraziamenti

Un ringraziamento speciale a **edo.js** e a chi ha collaborato alla creazione della **"Bibbia" di Programmazione 2**:
una risorsa fondamentale per comprendere e affrontare al meglio questo corso.

☕ Se vuoi offrirmi un caffè

- paypal

- revolut

SIMULAZIONE

CONTA FOGLIE

```
/**
* @brief Dato un albero binario restituisce:
* il numero di foglie (se l'albero è vuoto restituisce 0).
*/
int nfoglie(CharTree tree);
```

soluzione

```
int nfoglie(CharTree tree){
    if(!tree){  //caso con nessuna foglia
        return 0;
    }
    if(tree->left == NULL &&  tree->right == NULL){ //caso con nessun figlio
        return 1;
    }
    return nfoglie(tree->left) + nfoglie(tree->right);
}
```

INVERTIRE UNA LISTA ITERATIVO

```
/**@brief Riorganizza i nodi di *lsPtr invertendone l'ordine. Ad ad esempio,
data [1, 2,3] la trasforma in [3, 2,1]. Non alloca nuova memoria sullo heap.
*/
void reverse (IntList *plst);
```

SOLUZIONE ITERATIVA

```
void reverse (IntList *plst) {
    IntList prev = NULL;
    IntList current = *plst;
    IntList next = NULL;

    while (current) {
```

```
        next = current->next;   // Salva il prossimo nodo
        current->next = prev;   // Inverti il puntatore
        prev = current;         // Avanza prev
        current = next;         // Avanza current
    }
    *plst = prev;  // Aggiorna il puntatore alla testa della lista
}
```

SOLUZIONE RICORSIVA

```
void reverse (IntList *plst) {
    if (*plst == NULL || (*plst)->next == NULL){
         return;
    }
    IntList first = *plst;
    // Prendiamo il puntatore al resto della lista
    IntList rest = first->next;
    // Chiamata ricorsiva per invertire il resto della lista
    reverse(&rest);
    // Dopo la chiamata ricorsiva, rest è la testa della lista invertita
    // Quindi colleghiamo il primo nodo (che era la testa originale)
    // come l'ultimo della lista invertita
    first->next->next = first;
    // Spezzare il link originale per evitare ciclo
    first->next = NULL;
    // La nuova testa della lista è il risultato della chiamata ricorsiva
    *plst = rest;
```

PRIMO APPELLO

STRINGA PALINDROMA

```
/**
 * @brief Verifica se una sottostringa è palindroma.
 *
 * P-IN(s, first, last):
 *  - `s` è una stringa valida (cioè un array di caratteri terminato da '\0')
 * - `first` e `last` sono indici validi in `s` tali che 0 ≤ first ≤ last <
strlen(s)
 *
 * P-OUT(s, first, last, result):
 * - `result` è il valore di verità dell'affermazione:
 *"la sequenza di caratteri contenuti in `s[first...last]` è palindroma"
 *
```

```
 * @param s La stringa di input
 * @param first L'indice iniziale della sottostringa da verificare
 * @param last L'indice finale della sottostringa da verificare
 * @return true se la sottostringa è palindroma, false altrimenti
 */
_Bool isPalindrome(const char *s, int first, int last);
```

soluzione

```
_Bool isPalindrome(const char *s, int first, int last) {
    while (first < last) {
        if (s[first] != s[last]) {
            return 0;  // Trovati due caratteri diversi
        }
        first++;
        last--;
    }
    return 1;  // Tutti i caratteri combaciano
}
```

```
/**@brief Restituisce la lista alternata dei nodi di *lsPtr1 e *lpPrt2,
togliendoli da *lsPtr1 e *lsPtr2 , che alla fine conterranno entrambi NULL
(non alloca nuova memoria). Ad es. date [1, 5, 9] e [0, 2, 4, 6, 8]
restituisce [1, 0, 5,  2, 9,  4, 6, 8].
*/
IntList mixAlternate(IntList *lsPtr1, IntList *lsPtr2);
```

soluzione

```
#include <stdlib.h>
IntList mixAlternate(IntList *IsPtr1, IntList *IsPtr2) {
    IntList head = NULL;     // testa della lista
    IntList *tail = &head;// puntatore al puntatore dell'ultimo next da
aggiornare

    // alterna nodi da *IsPtr1 e *IsPtr2 fino a esaurimento
    while (*IsPtr1 != NULL || *IsPtr2 != NULL) {
        if (*IsPtr1 != NULL) {
            *tail = *IsPtr1;            // collega nodo corrente da lista 1
            *IsPtr1 = (*IsPtr1)->next;  // avanza lista 1
            tail = &((*tail)->next);    // sposta tail in avanti
        }
        if (*IsPtr2 != NULL) {
```

```
            *tail = *IsPtr2;            // collega nodo corrente da lista 2
            *IsPtr2 = (*IsPtr2)->next;  // avanza lista 2
            tail = &((*tail)->next);    // sposta tail in avanti
        }
    }
    // entrambi i puntatori ora devono essere NULL
    *IsPtr1 = NULL;
    *IsPtr2 = NULL;

    return head;
}
```

specchia albero

```
/**
* @brief Trasforma un albero nella sua versione speculare.
*
* P-IN: tree è un albero binario valido (o NULL).
* P-OUT: l'albero originale viene modificato in-place scambiando
ricorsivamente i sottoalberi sinistro e destro di ogni nodo.
 */
void mirror(IntTree tree);
```

```
void mirror(IntTree tree) {
    if (tree == NULL) {
        return; // Caso base: albero vuoto
    }

    // Scambia i figli sinistro e destro
    IntTree temp = tree->left;
    tree->left = tree->right;
    tree->right = temp;

    // Chiamate ricorsive sui sottoalberi
    mirror(tree->left);
    mirror(tree->right);
}
```

SECONDO APPELLO

```
/**
 * @brief Verifica se i primi n2 caratteri della stringa s1 coincidono con la
stringa s2.
```

```
 *
 * Dati due puntatori a stringhe s1 e s2 e le loro lunghezze n1 e n2,
 * restituisce 1 se i primi n2 caratteri di s1 coincidono esattamente con s2,
 * altrimenti 0.
 *
 * @param s1 La stringa da cui si confrontano i primi n2 caratteri.
 * @param n1 La lunghezza di s1 (>= 0).
 * @param s2 La stringa da confrontare.
 * @param n2 La lunghezza di s2 (>= 0).
 * @return _Bool 1 se s2 coincide con i primi n2 caratteri di s1, 0
altrimenti.
 */

_Bool check(const char *s1, int n1, const char *s2, int n2);
```

soluzione

```
if (n1 < n2) return 0;
    for (int i = 0; i < n2; ++i) {
        if (s1[i] != s2[i]) return 0;
    }
    return 1;
}
```

```
/**
 * @brief Conta i nodi dell'albero binario la cui profondità è compresa tra
due valori.
 *
 * Dato un albero binario `tree` e due interi `m` e `n`, restituisce il numero
di nodi
 * che si trovano a profondità compresa tra `m` e `n` (inclusi). La radice è
considerata
 * a profondità 0. Se `tree` è NULL, restituisce 0.
 *
 * Esempio:
 * Dato un albero con la seguente struttura:
 *
 *        R
 *       / \
 *      F   Z
 *     / \
 *    D   H
 *       / \
 *      G   L
```

```
 *         / \
 *        I   M
 *
 * e m = -2, n = 3, la funzione restituisce 7.
 *
 * @param tree Puntatore alla radice dell'albero binario.
 * @param m Profondità minima (inclusa) da considerare.
 * @param n Profondità massima (inclusa) da considerare.
 * @return Numero di nodi compresi tra le profondità m e n.
 */
int count_helper(CharTree tree, int m, int n, int depth);
```

soluzione

```
int count_helper(CharTree tree, int m, int n, int depth) {
    if (tree == NULL){
        return 0;
    }

    int count = 0;

    if (depth >= m && depth <= n) {
        count = 1;
    }

    count += count_helper(tree->left, m, n, depth + 1);
    count += count_helper(tree->right, m, n, depth + 1);

    return count;
}
```

```
/**
 * @brief Sposta i nodi da una lista all'altra in base a un criterio di
appartenenza.
 *
 * Rimuove dalla lista puntata da `*IsPtr1` tutti i nodi il cui valore è
presente
 * nella lista `Is2`, mantenendo l'ordine originale dei nodi rimossi.
 * I nodi rimossi vengono restituiti in una nuova lista, senza allocare nuova
memoria.
 *
 * La lista `*IsPtr1` viene modificata in-place, escludendo i nodi estratti.
 * La lista restituita conterrà solo i nodi estratti, nell'ordine in cui
apparivano in `*IsPtr1`.
```

```
 *
 * Esempio:
 * - Input: *IsPtr1 = [1,2,5,3,4,5,9,8], Is2 = [7,5,2,4]
 * - Output:
 *     *IsPtr1 → [1,3,9,8]
 *     return → [2,5,4,5]
 *
 * @param IsPtr1 Puntatore alla lista da modificare (verrà aggiornata).
 * @param Is2 Lista contenente i valori da cercare e rimuovere da *IsPtr1.
 * @return Nuova lista contenente i nodi rimossi da *IsPtr1, senza
riordinarli.
 */
int transfer(IntList *IsPtr1, IntList Is2);
```

soluzione

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

typedef struct node {
    int data;
    struct node *next;
} IntNode, *IntList;


int transfer(IntList *IsPtr1, IntList Is2) {
    IntList curr = *IsPtr1;
    IntList prev = NULL;
    IntList removed = NULL;
    IntList removedTail = NULL;

    while (curr != NULL) {
        // Cerca curr->data dentro Is2 (manuale, inline)
        IntList search = Is2;
        int found = 0;
        while (search != NULL) {
            if (search->data == curr->data) {
                found = 1;
                break;
            }
            search = search->next;
        }

        if (found) {
            // Rimuovi il nodo da *IsPtr1
```

```
            IntList toRemove = curr;
            curr = curr->next;

            if (prev == NULL) {
                *IsPtr1 = curr;
            } else {
                prev->next = curr;
            }

            // Inserisci in coda alla lista removed
            toRemove->next = NULL;
            if (removed == NULL) {
                removed = removedTail = toRemove;
            } else {
                removedTail->next = toRemove;
                removedTail = toRemove;
            }

        } else {
            prev = curr;
            curr = curr->next;
        }
    }

    return removed;
}
```

TERZO APPELLO

```
/**
 * @brief Verifica se una sottostringa è palindroma (ricorsivamente).
 *
 * La funzione determina se la porzione di stringa compresa tra gli indici
`first` e `last`
 * (inclusi) è un palindromo. Una stringa è palindroma se è vuota oppure se
 * la metà sinistra è lo specchio della metà destra, con corrispondenza
**case-sensitive**.
 *
 * Esempi validi: "kayak", "anna" → palindromi
 * Esempi non validi: "Kayak" → non palindromo perché 'K' ≠ 'k'
 *
 * @param s Puntatore alla stringa da analizzare (non NULL).
 * @param first Indice del primo carattere della sottostringa da considerare.
 * @param last Indice dell'ultimo carattere della sottostringa da considerare.
 * @return true (_Bool 1) se la sottostringa `s[first...last]` è palindroma,
```

```
false (_Bool 0) altrimenti.
 */
 _Bool isPalindrome(const char *s, int first, int last);
```

soluzione

```
#include <stdbool.h>

_Bool isPalindrome(const char *s, int first, int last) {
    // Caso base: stringa vuota o di un solo carattere
    if (first >= last)
        return true;

    // Se i caratteri agli estremi non coincidono, non è palindroma
    if (s[first] != s[last])
        return false;

    // Passo ricorsivo: verifica la sottostringa interna
    return isPalindrome(s, first + 1, last - 1);
}
```

```
/**
 * @brief Unisce due liste ordinate in una nuova lista ordinata, riutilizzando
i nodi.
 *
 * La funzione riceve due puntatori a liste ordinate (`*LsPtr1` e `*LsPtr2`) e
restituisce
 * una nuova lista ordinata che contiene **tutti i nodi** di entrambe le
liste,
 * riordinati in modo crescente.
 *
 * ⚠ Non viene allocata nuova memoria: i nodi vengono **spostati** dalle due
liste iniziali
 * alla lista risultante. Alla fine dell'operazione, `*LsPtr1` e `*LsPtr2`
saranno entrambi `NULL`.
 *
 * Esempio:
 * - Input: `*LsPtr1 = [1, 5, 9]`, `*LsPtr2 = [0, 2, 4, 6, 8]`
 * - Output: `[0, 1, 2, 4, 5, 6, 8, 9]`
 *
 * @param LsPtr1 Puntatore al primo puntatore di lista ordinata (modificato,
diventa NULL).
 * @param LsPtr2 Puntatore al secondo puntatore di lista ordinata (modificato,
diventa NULL).
```

```
 * @return Lista ordinata risultante contenente tutti i nodi di `*LsPtr1` e
`*LsPtr2`.
 */
```

soluzione

```c
#include <stdlib.h>

typedef struct IntList IntList;
typedef IntList* inList;

struct IntList {
    int data;
    inList next;
};

inList merge(inList *IsPtr1, inList Is2) {
    inList curr = *IsPtr1;
    inList prev = NULL;

    inList extractedHead = NULL;
    inList extractedTail = NULL;

    while (curr != NULL) {
        // Cerco se curr->data è presente in Is2
        inList search = Is2;
        int found = 0;
        while (search != NULL) {
            if (search->data == curr->data) {
                found = 1;
                break;
            }
            search = search->next;
        }

        if (found) {
            // Nodo da estrarre
            inList toExtract = curr;
            curr = curr->next;

            if (prev == NULL) {
                *IsPtr1 = curr;
            } else {
                prev->next = curr;
            }
```

```
            toExtract->next = NULL;
            if (extractedTail == NULL) {
                extractedHead = extractedTail = toExtract;
            } else {
                extractedTail->next = toExtract;
                extractedTail = toExtract;
            }
        } else {
            prev = curr;
            curr = curr->next;
        }
    }
    return extractedHead;
}
```

```
/**
 * @brief Trasforma un albero, agendo su ogni nodo con entrambi i rami,
 *        scambiandoli se le loro radici non sono nell'ordine corretto
 *        (ovvero lo scambio avviene quando sinistra > destra).
 */
void sort(IntTree tree);
```

soluzione

```
#include <stdio.h>

typedef struct treeNode {
    struct treeNode *left;
    int data;
    struct treeNode *right;
} treeNode, *IntTree;

void sort(IntTree tree) {
    if (tree == NULL) return;

    // Chiamate ricorsive sui sottoalberi sinistro e destro
    sort(tree->left);
    sort(tree->right);

    // Condizione di scambio: entrambi i sottoalberi devono esistere
    if (tree->left && tree->right && tree->left->data > tree->right->data) {
        // Scambia i puntatori ai sottoalberi
        IntTree temp = tree->left;
```

```
        tree->left = tree->right;
        tree->right = temp;
    }
}
```

Primo appello turno 2 2025

```
/**
 * @brief Rimuove il terzultimo nodo dalla lista collegata puntata da lsPtr.
 *
 * La funzione modifica *lsPtr rimuovendo il terzultimo nodo (se esiste)
 * e liberando la memoria ad esso associata.
 *
 * @param lsPtr Puntatore al primo elemento della lista (modificabile).
 * @return int
 * - -1 se lsPtr è NULL
 * -  0 se la lista è vuota, ha meno di tre elementi, o se si verifica un
errore
 * -  1 se il terzultimo nodo è stato rimosso con successo
 */
int eliminaTerzultimo(IntList *lsPtr);
```

soluzione

```
#include<stdlib.h>
typedef struct node IntNode, *IntList;
struct node {
int data;
IntList next;
};
int eliminaTerzultimo(IntList *lsPtr) {
    if (lsPtr == NULL) return -1;
    if (*lsPtr == NULL || (*lsPtr)->next == NULL || (*lsPtr)->next->next ==
NULL)
        return 0;

    IntList current = *lsPtr;
    IntList prev = NULL;

    // Trova la lunghezza della lista
    int count = 0;
    while (current != NULL) {
        count++;
        current = current->next;
```

```
    }

    if (count < 3) return 0;

    int target = count - 3; // posizione del terzultimo (0-based)

    current = *lsPtr;
    for (int i = 0; i < target; i++) {
        prev = current;
        current = current->next;
    }

    if (prev == NULL) {
        // Il nodo da eliminare è la testa
        *lsPtr = current->next;
    } else {
        prev->next = current->next;
    }

    free(current);
    return 1;
}
```

```
/**
 * @brief Calcola la somma di tutte le foglie che sono figlie uniche del
proprio padre.
 *
 * Una foglia è un nodo senza figli (entrambi i puntatori `left` e `right`
sono NULL).
 * Una foglia è considerata "figlia unica" se è l'unico figlio non NULL del
proprio padre.
 *
 * ESEMPI:
 * - Albero vuoto → restituisce 0
 * - Albero con un solo nodo → restituisce il valore di quel nodo
 * - Per l'albero:
 *   @code
 *         1
 *        / \
 *       2   3
 *      / \    \
 *     4   5    7
 *    /
 *   10
 *   @endcode
```

```
 *   le foglie "figlie uniche" sono 10 (figlia unica di 4) e 7 (figlia unica
di 3), quindi la funzione restituisce 17.
 *
 * @param tree Puntatore alla radice dell'albero.
 * @return int Somma dei valori delle foglie che sono figlie uniche del
proprio padre.
 */
 int sumX(IntTree tree);
```

```
int sumX(IntTree tree) {
    if (tree == NULL)
        return 0;

    // Caso in cui il nodo è una foglia
    if (tree->left == NULL && tree->right == NULL) {
        if (tree->parent != NULL) {
            // Verifica se è figlia unica del padre
            if ((tree->parent->left == tree && tree->parent->right == NULL) ||
                (tree->parent->right == tree && tree->parent->left == NULL)) {
                return tree->data;
            }
        }
        return 0;
    }

    // Chiamata ricorsiva su figli
    return sumX(tree->left) + sumX(tree->right);
}
```

Primo appello turno 3 2025

```
/**
 * @brief Rimuove dalla lista tutti i nodi in posizione pari (1-based).
 *
 * La funzione modifica direttamente la lista puntata da `lsPtr`, eliminando
 * tutti i nodi che si trovano in posizione pari (numerazione da 1 in su).
 *
 * @param[in,out] lsPtr Puntatore al primo elemento della lista (`IntList *`).
 *
 * @retval -1 Se `lsPtr` è NULL (parametro non valido).
 * @retval 0 Se la lista è vuota o contiene un solo elemento (nessun nodo
eliminato).
 * @retval N Numero di nodi eliminati (N > 0) se l'operazione ha successo.
 *
```

```
 * @note La funzione libera la memoria dei nodi eliminati usando `free()`.
 *
 * @warning Se lsPtr è NULL, la lista non viene modificata.
 */
int eliminaPari(IntList *lsPtr);
```

soluzione

```
#include <stdlib.h>

typedef struct node {
    int data;
    struct node* next;
} IntNode, *IntList;

int eliminaPari(IntList *lsPtr) {
    if (lsPtr == NULL) {
        return -1;  // lsPtr è NULL → errore
    }

    IntList current = *lsPtr;

    if (current == NULL) {
        return 0;  // lista vuota → nessun nodo da eliminare
    }

    int count = 0;           // numero di nodi eliminati
    int pos = 1;             // posizione corrente (parte da 1)
    IntList prev = NULL;     // nodo precedente
    while (current != NULL) {
        if (pos % 2 == 0) {
            // Nodo in posizione pari → da eliminare
            IntList toDelete = current;
            if (prev != NULL) {
                prev->next = current->next;
            } else {
                // Non dovrebbe accadere: prev è NULL solo alla posizione 1
                *lsPtr = current->next;
            }
            current = current->next;
            free(toDelete);
            count++;
        } else {
            // Nodo in posizione dispari → mantenerlo
            prev = current;
```

```
            current = current->next;
        }
        pos++;
    }

    return count;
}
```

```
/**
 * @brief Calcola la somma dei valori di tutti i nodi che sono figli sinistri
nel loro albero.
 *
 * La funzione scorre ricorsivamente l'albero binario e somma i valori dei
nodi che sono
 * figli sinistri rispetto al proprio nodo genitore.
 *
 * @param[in] tree Puntatore alla radice dell'albero binario.
 *
 * @retval 0 Se l'albero è vuoto o non ci sono figli sinistri.
 * @retval N Somma dei valori dei nodi che sono figli sinistri (N >= 0).
 *
 * @note La funzione non modifica l'albero.
 *
 * @par Esempi:
 * - Albero vuoto: restituisce 0.
 * - Albero con un solo nodo: restituisce 0.
 * - Albero con struttura:
 *       1
 *        \
 *         2
 *       /  \
 *      3    4
 *   Risultato: 3
 */
int sumLeft(IntTree tree);
```

soluzione

```
int sumLeft (IntTree tree) {
   if(tree == NULL){
       return 0;
   }
   int sum = 0;
```

```
    if(tree->left != NULL){
        sum += tree->left->data;
    }
   return sum + sumLeft(tree->left) + sumLeft(tree->right);
}
```