



Programmazione 1 (Canale O-Z)

Lezione 9: Le Stringhe e lo Spazio Multidimensionale

Docente: Prof. Antonino Furnari

Email: antonino.furnari@unict.it

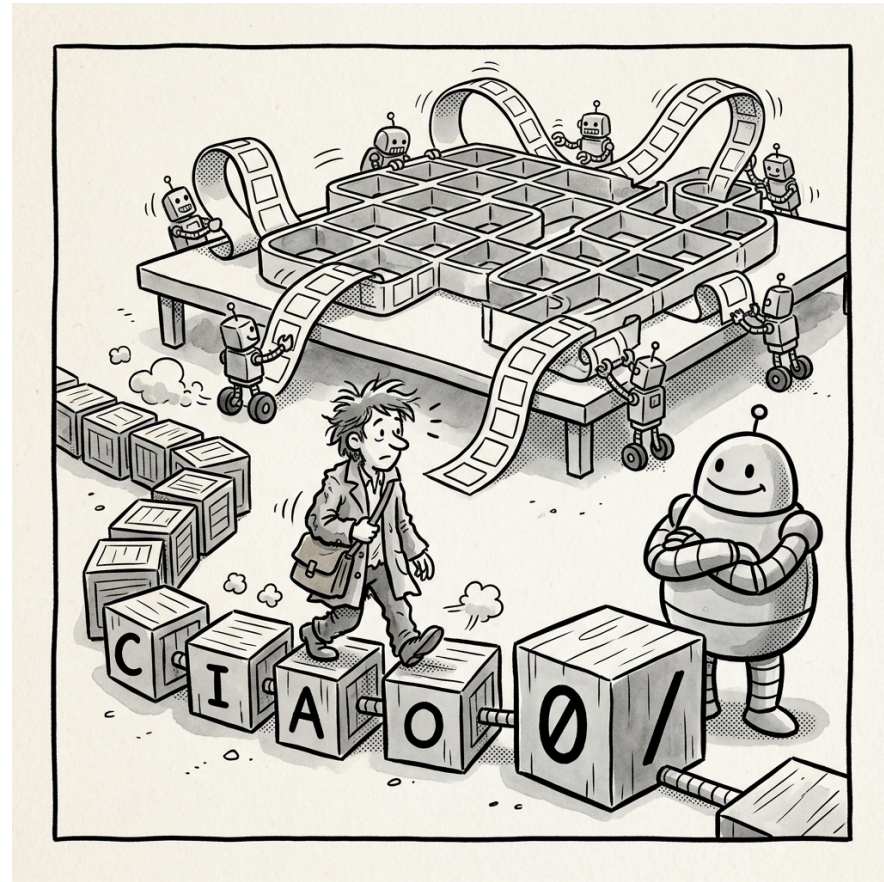
Sito Web: antoninofurnari.github.io/prog1-2627

L'Astrazione del Testo (Il Tipo che non Esiste)

- Finora: `int`, `float`, `double` e singoli `char`.
- Per parole, frasi, percorsi file... in C **non esiste** un tipo nativo `string`.
- Una stringa è una sequenza contigua di caratteri: un **array di `char`**.

```
char parola[10]; // 10 byte fissi nello Stack
```

? "Etna" ha 4 caratteri, "Catania" 7: dove finisce il testo utile in un array di dimensione fissa?



Il Terminatore `\0` (Sentinella)

- La fine logica di una stringa è marcata dal **terminatore null** `\0` (valore `0` della tabella ASCII).

```
// Metodo 1: Inizializzazione Implicita (virgolette)  
// Il compilatore alloca 5 byte: 4 caratteri + il '\0' inserito in coda  
char s1[] = "Ciao";  
  
// Metodo 2: Inizializzazione Esplicita (terminatore manuale)  
char s2[] = {'C', 'i', 'a', 'o', '\0'};
```

? Cosa succede se nel Metodo 2 dimentichiamo il `'\0'`?

Cosa Succede se Dimentichiamo il `\0`?

```
#include <stdio.h>

int main(void) {
    char s_bug[] = {'B', 'u', 'g'}; // ERRORE: Manca la sentinella!
    printf("%s\n", s_bug);
    return 0;
}
```

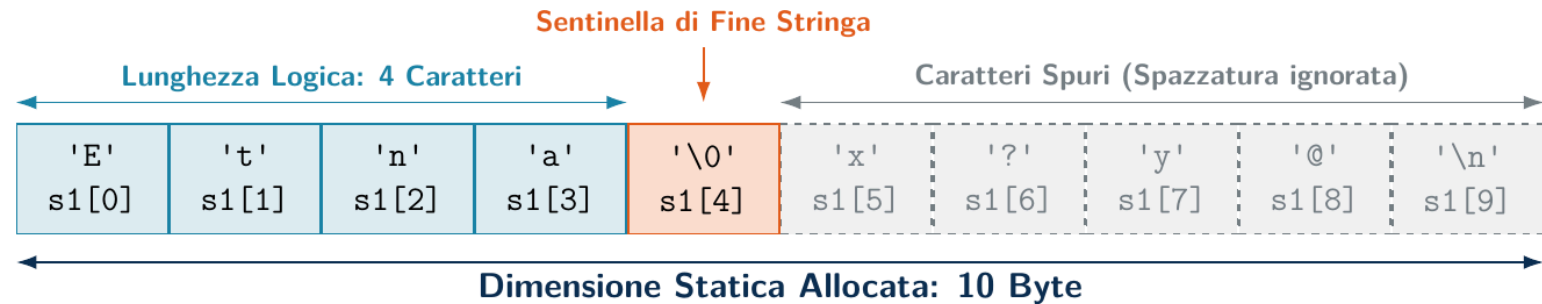
- `printf("%s")` parte da `'B'` e stampa byte dopo byte **finché non trova uno `0`**.
- Senza sentinella continua a leggere le celle adiacenti dello Stack: stampa **spazzatura** finché, per coincidenza, incontra uno zero, oppure va in crash (`Segmentation Fault`).

 **Live:** `gcc -Wall bug.c -o bug && ./bug`, eseguire più volte. Il risultato non è prevedibile: può anche sembrare corretto per puro caso. Poi aggiungere `'\0'` e riprovare.

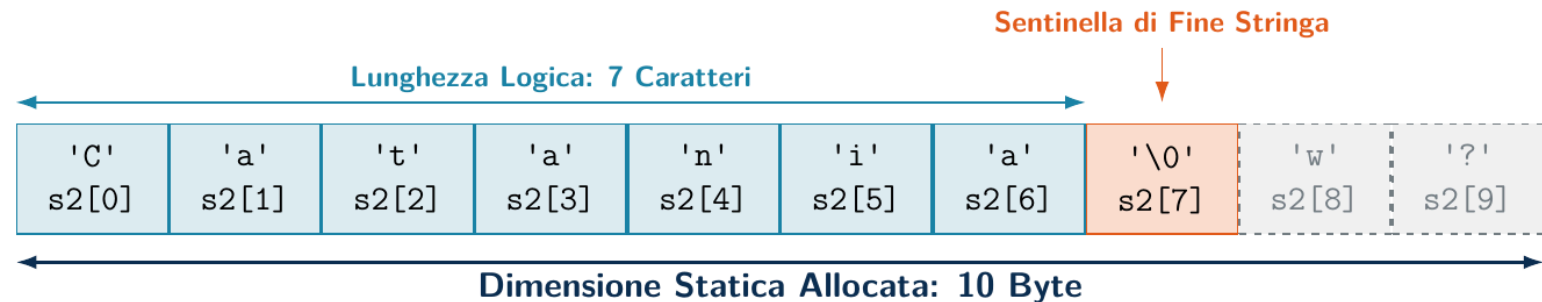
Lunghezza Massima vs Lunghezza Logica

- **Dimensione statica** (lunghezza massima): i byte riservati nello Stack (es. `char nome[100]`).
- **Lunghezza logica**: i caratteri utili **prima** del `\0` (misurata da `strlen`).
- I caratteri dopo il `\0` sono **spuri** (spazzatura): le funzioni standard li ignorano.

```
char s1[10] = "Etna";
```



```
char s2[10] = "Catania";
```



Scansione di una Stringa: Indice vs Puntatore

```
#include <stdio.h>

int main(void) {
    char str[] = "Etna";

    // Versione A: indicizzazione classica
    printf("Versione A: ");
    for (int i = 0; str[i] != '\0'; i++) {
        putchar(str[i]);
    }
    putchar('\n');
```

```
// Versione B: aritmetica dei puntatori
printf("Versione B: ");
char *p = str; // p punta a str[0]
while (*p != '\0') {
    putchar(*p);
    p++; // avanti di 1 byte (un char)
}
putchar('\n');

return 0;
}
```

Versione A: Etna
Versione B: Etna

? `str` decade a puntatore al primo carattere: cosa stampa `printf("%s\n", str + 2);`?

Tagliare le Stringhe con l'Aritmetica

```
printf("%s\n", str + 2); // Salta i primi due caratteri
```

na

- Nessuna copia in un nuovo array: basta **spostare l'indirizzo di partenza**.
- `printf` stampa da lì fino al `\0`, che è rimasto al suo posto.

 **Live:** aggiungere la riga al programma precedente, poi provare `str + 1` e `str + 4` (stringa vuota: si parte direttamente dal `\0`).

Analisi dei Caratteri con `<ctype.h>`

- No ai numeri magici ASCII (`if (c ≥ 65 && c ≤ 90)`): usiamo le funzioni di `<ctype.h>`.

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

int main(void) {
    char c = 'a';

    if (isalpha(c)) {
        printf("'%c' e' una lettera\n", c);
    }
    if (islower(c)) {
        printf("Ed e' minuscola! In maiuscolo: '%c'\n", toupper(c));
    }
    return 0;
}
```

```
'a' e' una lettera
Ed e' minuscola! In maiuscolo: 'A'
```

- Altre: `isdigit(c)` (cifra `0..9`), `isalnum(c)` (alfanumerico), `isspace(c)` (spazio, tab, a capo), `tolower(c)`.

! Da Stringa a Numero: l'Approccio Unsafe (`atoi`)

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

int main(void) {
    int val = atoi("123"); // Ritorna 123 (Corretto)
    int err = atoi("ciao"); // Ritorna 0! (Fallimento)
}
```

- Se `atoi` ritorna `0`, è **impossibile** capire se l'utente ha digitato davvero `"0"` o se la conversione è fallita.

? Come potrebbe una funzione dirci anche **dove** si è fermata la conversione?



L'Approccio Safe: `strtol` e `strtod`

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

int main(void) {
    char input[] = "137.56 euro";
    char *endptr;

    double prezzo = strtod(input, &endptr);

    if (input == endptr) {
        printf("Errore: nessuna conversione numerica trovata!\n");
    } else {
        printf("Prezzo convertito: %.2f\n", prezzo);
        printf("Stringa non convertita rimanente: '%s'\n", endptr);
    }
    return 0;
}
```



strtod: Come Funziona endptr

Prezzo convertito: 137.56

Stringa non convertita rimanente: ' euro'

- `&endptr` è un **puntatore a puntatore** (`char **`): `strtod` vi scrive l'indirizzo del primo carattere non convertito.
- `input == endptr` ➡ nessun carattere convertito: **errore**.
- Altrimenti `endptr` punta al resto della stringa (" euro").



Live: cambiare `input` in "ciao" e poi in "0": con `strtod` i due casi sono distinguibili, con `atoi` no.



La Libreria Standard `<string.h>`

- `strlen(s)`: lunghezza logica (escluso il `\0`).
- `strcpy(dest, src)`: copia `src` in `dest` (`dest` **deve avere spazio** per la copia e il `\0`!).
- `strcat(dest, src)`: concatena `src` in coda a `dest`.
- `strcmp(s1, s2)`: confronto lessicografico sottraendo i caratteri:
 - `0` se identiche;
 - negativo se `s1 < s2`, positivo se `s1 > s2`.



Live: consultare il manuale dal terminale: `man strlen`, `man strcpy`.



<string.h> in Azione: Lunghezza, Copia, Concatenazione

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

int main(void) {
    char str1[50] = "Etna";
    char str2[] = "Coder";
    char buffer[50];

    printf("Lunghezza di str1: %zu\n", strlen(str1)); // Lunghezza logica
    strcpy(buffer, str1);                             // Copia
    printf("Buffer copiato: %s\n", buffer);
    strcat(buffer, str2);                               // Concatenazione
    printf("Buffer concatenato: %s\n", buffer);
}
```

```
Lunghezza di str1: 4
Buffer copiato: Etna
Buffer concatenato: Etna Coder
```



<string.h> in Azione: Confronto

```
// Confronto lessicografico
if (strcmp(str1, "Etna") == 0) {
    printf("Le due stringhe sono identiche!\n");
}

int confronto = strcmp("albero", "barca");
printf("Confronto 'albero' vs 'barca': %d\n", confronto);

return 0;
}
```

```
Le due stringhe sono identiche!
Confronto 'albero' vs 'barca': -1
```

- Il valore esatto dipende dall'implementazione: conta solo il **segno** (`< 0`: "albero" precede "barca").

 **Live:** compilare con `gcc -Wall stringhe.c -o stringhe`; provare `strcmp("barca", "albero")` e `strcmp("Etna", "etna")`.



La Piaga del Buffer Overflow

```
#include <stdio.h>

int main(void) {
    char password[8] = "segreto"; // 7 caratteri + '\0'
    char input[8];                // Buffer di input limitato a 8 byte

    printf("Inserisci la password di sblocco: ");
    // PERICOLO: scanf non sa quanto sia grande 'input'!
    scanf("%s", input);

    printf("Hai inserito: %s\n", input);
    printf("Password di controllo attuale: %s\n", password);
    return 0;
}
```

- `scanf("%s")` **non controlla i limiti** del buffer: i caratteri in eccesso sovrascrivono le variabili adiacenti o gli indirizzi di ritorno dello Stack Frame.

Buffer Overflow: Proviamo a Romperlo

 **Live:** `gcc -Wall overflow.c -o overflow && ./overflow`

- prima con un input corto (`ciao`): tutto regolare;
 - poi con `1234567890abcdef`: osservare se `password` viene alterata dal nostro input o se il programma va in crash (`Segmentation Fault`, `Illegal instruction`...). L'esito dipende da compilatore e sistema.
- **Regola assoluta:** non usare mai `scanf("%s")` per acquisire testo libero.
- ? Ci serve una funzione che conosca la dimensione del buffer: quale?



La Soluzione Sicura: fgets

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

int main(void) {
    char nome[20];
    printf("Inserisci il tuo nome: ");

    // Legge al massimo 19 caratteri lasciando il 20-esimo per il '\0'
    if (fgets(nome, sizeof(nome), stdin) != NULL) {
        // fgets acquisisce anche l'invio '\n', rimuoviamolo se presente
        nome[strcspn(nome, "\n")] = '\0';
        printf("Ciao %s!\n", nome);
    }
    return 0;
}
```

 **Live:** eseguire con un nome lungo più di 19 caratteri: nessun crash, la stringa viene troncata. Poi commentare la riga con `strcspn` e far vedere l'a capo in più.



Svuotare il Buffer Residuo

- Una `scanf` numerica lascia l'invio `\n` nel buffer di input.
- Una `fgets` subito dopo leggerebbe **solo quell'invio**: stringa vuota!
- Prima della `fgets`, consumiamo i caratteri spuri:

```
int c;  
while ((c = getchar()) != '\n' && c != EOF); // Consuma i caratteri spuri
```

- `getchar()` legge **un singolo carattere** dal buffer di input (il duale di `putchar`).

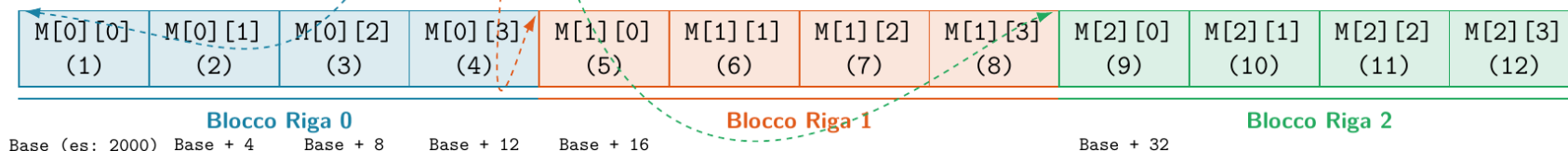
Le Matrici e la Bugia della Griglia

- Logicamente: una griglia `int M[3][4]` (3 righe, 4 colonne). Ma la RAM è **rigorosamente lineare**.
- **Row-Major Order**: le righe vengono accodate una dopo l'altra in celle consecutive.

Rappresentazione Logica (Griglia 3 × 4)

Riga 0	M[0][0] (1)	M[0][1] (2)	M[0][2] (3)	M[0][3] (4)
Riga 1	M[1][0] (5)	M[1][1] (6)	M[1][2] (7)	M[1][3] (8)
Riga 2	M[2][0] (9)	M[2][1] (10)	M[2][2] (11)	M[2][3] (12)

Rappresentazione Fisica in RAM (Disposizione Lineare)



? Come trova la CPU `M[i][j]` in questa striscia di 12 celle?

La Formula di Offset

Indirizzo Fisico = Indirizzo Base + $(i \times \text{Numero Colonne} + j) \times \text{sizeof(tipo)}$

- La CPU **non** ha bisogno di conoscere il numero di righe.
- La CPU **deve** conoscere il numero di **colonne**: per sapere quanti elementi scavalcare per arrivare alla riga `i`.
- Esempio in `int M[3][4]: M[2][1]` ➡ Base + $(2 \times 4 + 1) \times 4 = \text{Base} + 36$ byte.

? Quindi, cosa deve comparire obbligatoriamente nella firma di una funzione che riceve una matrice?

Passare una Matrice a una Funzione

```
#include <stdio.h>

// La colonna (4) serve alla CPU per
// il calcolo corretto dell'offset in RAM!
void stampa_matrice(int M[][4], int righe) {
    for (int i = 0; i < righe; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d\t", M[i][j]);
        }
        printf("\n");
    }
}
```

```
int main(void) {
    int matrice[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    stampa_matrice(matrice, 3);
    return 0;
}
```

1	2	3	4
5	6	7	8
9	10	11	12

- Le righe si passano come parametro (`int righe`), le colonne sono **fissate** nella firma.

abc Matrici di Caratteri come Array di Stringhe

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

int main(void) {
    // 3 stringhe, max 10 byte ciascuna
    char arr[3][10] = {
        "Etna",
        "Corso",
        "Catania"
    };

    // arr[i]: indirizzo della riga i-esima
    for (int i = 0; i < 3; i++) {
        printf("Indice %d: %s\n", i, arr[i]);
    }
    return 0;
}
```

- Ogni **riga** è una stringa indipendente, terminata da `\0`.
- Le **colonne** fissano la capacità massima di ogni stringa.

```
Indice 0: Etna
Indice 1: Corso
Indice 2: Catania
```

Rappresentazione in Memoria e Spreco di Spazio

- `char arr[3][10]` alloca rigidamente $3 \times 10 = 30$ byte.
- "Etna" usa solo 5 byte (incluso `\0`): gli altri 5 della riga restano **allocati ma inutilizzati**.
- Più le stringhe sono di lunghezza diversa, più spazio sprechiamo.

	0	1	2	3	4	5	6	7	8	9
arr[0]	E	t	n	a	\0					
arr[1]	C	o	r	s	o	\0				
arr[2]	C	a	t	a	n	i	a	\0		

Diagram illustrating memory allocation and waste. The array `arr` is a 3x10 grid. The first row (`arr[0]`) contains "Etna" and a null terminator at index 4. The second row (`arr[1]`) contains "Cors" and a null terminator at index 5. The third row (`arr[2]`) contains "Catana" and a null terminator at index 8. The remaining cells (indices 6-9 of row 0, 6-9 of row 1, and 9 of row 2) are shaded gray and labeled "Memoria Sprecata" (Wasted Memory) with dashed arrows.

Operazioni Invalidi sugli Array di Stringhe

- `arr[0]` è l'indirizzo base di un array fisso: un puntatore **costante**, non riassegnabile.

```
char arr[3][10] = {"Etna", "Corso", "Catania"};  
// ERROR: "assignment to expression with array type"  
arr[0] = "GFG";
```

- Per cambiare il testo copiamo i caratteri con `strcpy`:

```
// Operazione Corretta: copia "GFG" (compreso terminatore) nel buffer arr[0]  
strcpy(arr[0], "GFG");
```

- Nelle prossime lezioni: **array di puntatori a stringhe** (`char *arr[]`) contro lo spreco e per la riassegnazione diretta.

 **Live:** compilare con `-Wall` la versione con `arr[0] = "GFG";` e leggere l'errore di `gcc`, poi sostituirla con `strcpy`.

Oltre le Tre Dimensioni (Array N-Dimensionali)

- Le dimensioni superiori sono solo un'**astrazione mentale**: per la CPU è sempre una sequenza piatta in RAM.

```
// 1D: Fila di 10 sensori lungo un tubo → double tubo[10];  
// 2D: Griglia di 10×10 sensori su un pavimento → double pavimento[10][10];  
// 3D: Reticolo 10×10×5 sensori in una stanza → double stanza[10][10][5];  
// 4D: Misura 3D ad ogni ora di un giorno (24 ore) → double log[24][10][10][5];  
  
// 5D: Monitoriamo 8 stanze diverse dello stesso plesso  
double plesso[8][24][10][10][5];  
  
// Stanza 2, ore 14:00, coordinate X=4, Y=7, Z=1  
double temp = plesso[2][14][4][7][1];
```

- La CPU espande **ricorsivamente** la formula di offset usando le dimensioni di ciascun blocco, note a compile-time.

✓ Cosa Abbiamo Imparato Oggi

- In C una stringa è un **array di char** terminato dalla sentinella `\0`.
- Lunghezza massima (byte allocati) $\neq$ lunghezza logica (`strlen`).
- Scansione con indice o puntatore; `str + k` "taglia" la stringa a costo zero.
- `<ctype.h>` al posto dei numeri magici; `strtod` / `strtol` al posto di `atoi`.
- `<string.h>`: `strlen`, `strcpy`, `strcat`, `strcmp` (attenzione allo spazio in `dest`!).
- Mai `scanf("%s")` per testo libero: `fgets` + rimozione del `\n`.
- Matrici in **Row-Major Order**: le colonne vanno sempre indicate nella firma.

 **Prossimo passo:** rivedere la dispensa della Lezione 9 sul sito; le stringhe torneranno in pratica nel Laboratorio 5.

Grazie per l'attenzione!

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