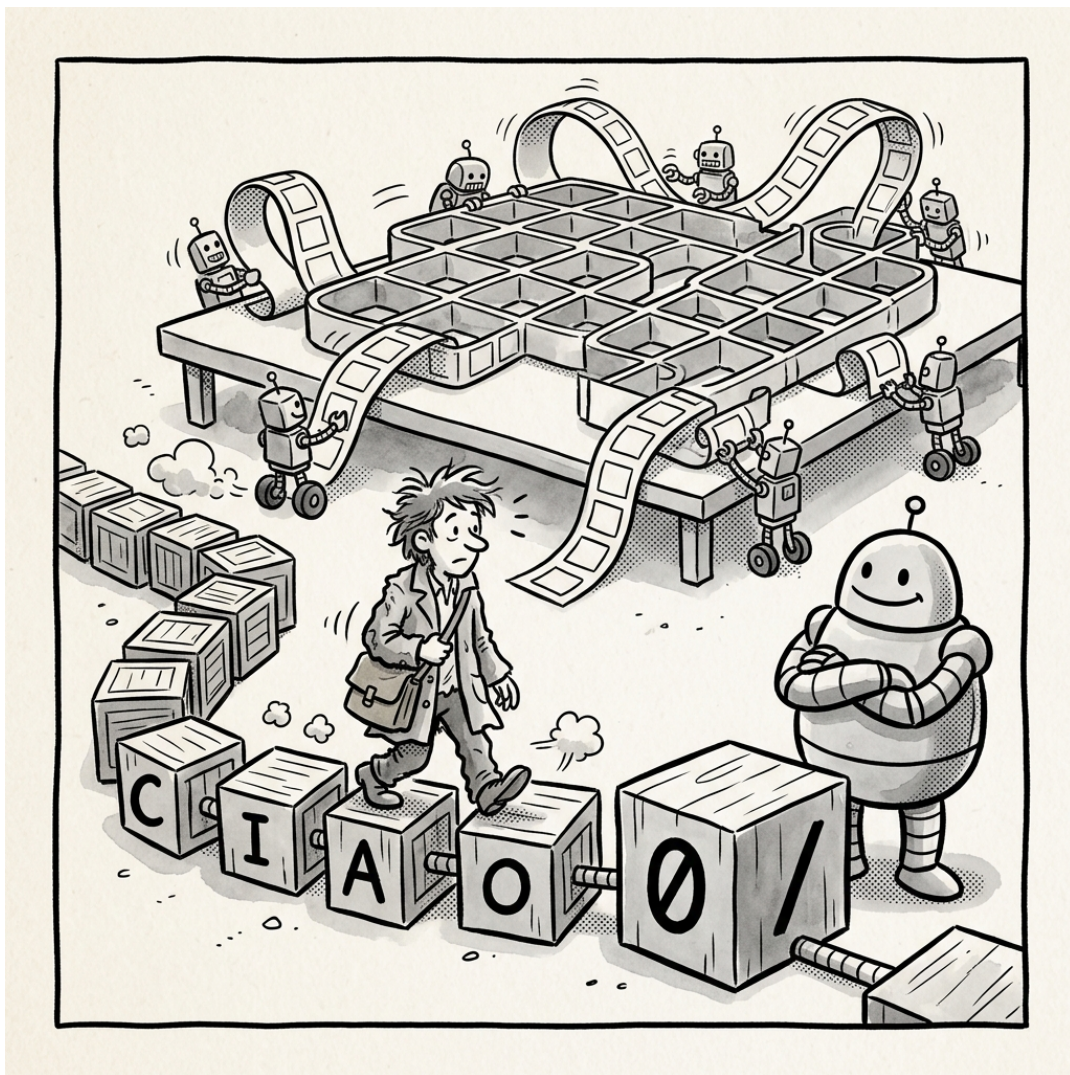


LEZIONE 9

Le Stringhe e lo Spazio Multidimensionale



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Le Stringhe e lo Spazio Multidimensionale

In questa lezione affronteremo due concetti apparentemente distanti ma accomunati dallo stesso principio architettonico: **come la memoria RAM (struttura lineare e contigua) viene organizzata dal compilatore per rappresentare astrazioni complesse come testi (stringhe) e griglie geometriche (matrici)**. Comprendremo perché il C non possiede un tipo stringa nativo, l'importanza della sentinella `\{\}\{\}\0` per delimitare i dati utili all'interno di buffer statici, e come calcolare gli indirizzi fisici degli elementi multidimensionali in Row-Major Order.

1.1 L'Anatomia delle Stringhe e il Terminatore Null

1.1.1 L'Astrazione del Testo (Il tipo che non esiste)

Fino a questo momento abbiamo manipolato tipi numerici primitivi (`int`, `float`, `double`) e singoli caratteri alfanumerici (`char`). Tuttavia, per gestire una sequenza di caratteri (parole, frasi, percorsi file), ci accorgiamo che **in C non esiste un tipo di dato nativo string**.

Fedele alla sua natura di linguaggio orientato al basso livello, il C rappresenta una stringa come una sequenza ordinata e contigua di caratteri in memoria. Ovvero: **un semplice array di caratteri (`char`)**.

```
1 char parola[10];
```

Poiché l'array ha una dimensione fissa allocata staticamente nello Stack (nell'esempio sopra, 10 byte), sorge un problema: i testi che scriviamo hanno lunghezze variabili (es. "Etna" ha 4 caratteri, "Catania" ne ha 7). Come fa il programma a sapere dove finisce il testo utile all'interno dell'array senza "sbrodolare" nelle celle adiacenti dello Stack?

1.1.2 Inizializzazione e l'Importanza del Terminatore `\0`

In C, la fine logica di una stringa è marcata da un carattere speciale chiamato **sentinella** o **terminatore null**, rappresentato dalla sequenza di escape `\{\}\{\}\0` (corrispondente al valore binario 0 della tabella ASCII).

Possiamo inizializzare una stringa in due modi:

```
1 // Metodo 1: Inizializzazione Implicita (Virgolette double quotes)
2 // Il compilatore alloca automaticamente 5 byte (4 caratteri + il '\0'
   ↪ inserito in coda)
3 char s1[] = "Ciao";
4
5 // Metodo 2: Inizializzazione Esplicita (Richiede l'inserimento manuale
   ↪ del terminatore)
6 char s2[] = {'C', 'i', 'a', 'o', '\0'};
```

Cosa succede se dimentichiamo il `\0`?

Consideriamo questo frammento di codice errato:

```
1 #include <stdio.h>
2
3 int main(void) {
4     char s_bug[] = {'B', 'u', 'g'}; // ERRORE: Manca la sentinella!
5     printf("%s\n", s_bug);
6     return 0;
7 }
```

Quando invochiamo `printf("\{%s", s\{}_bug)`, la CPU si posiziona sull'indirizzo iniziale dell'array `s\{}_bug` (la lettera 'B') e inizia a stampare consecutivamente i caratteri in RAM. Poiché non trova il valore nullo `\{\}\{\}\0` che gli impone l'arresto, la CPU continuerà a leggere e interpretare come caratteri le celle di memoria adiacenti sullo Stack (dangling memory). Questo comporta la stampa a video di simboli casuali e spazzatura finché, per pura coincidenza, non incontra un byte a zero nella RAM, oppure causa un crash immediato per violazione di accesso (Segmentation Fault).

1.1.3 Lunghezza Massima vs Lunghezza Logica

Grazie alla sentinella `\{\}\{\}\0`, possiamo riservare un buffer statico molto capiente (es. `char nome[100]`) e utilizzarlo per memorizzare stringhe di lunghezza variabile.

- **Dimensione Statica (Lunghezza Massima):** è la memoria fisicamente riservata dal compilatore nello Stack (il numero di elementi dell'array).
- **Lunghezza Logica (Dimensione Effettiva):** è il numero di caratteri utili presenti prima del terminatore `\{\}\{\}\0` (misurato dalla funzione `strlen`).

Il seguente diagramma mostra chiaramente la differenza strutturale tra due stringhe in memoria a parità di dimensione allocata (10 Byte):

I caratteri successivi alla sentinella `\{\}\{\}\0` sono **caratteri spuri (spazzatura)** e vengono completamente ignorati dalle funzioni standard di stampa e manipolazione del linguaggio C.

4 Le Stringhe e lo Spazio Multidimensionale

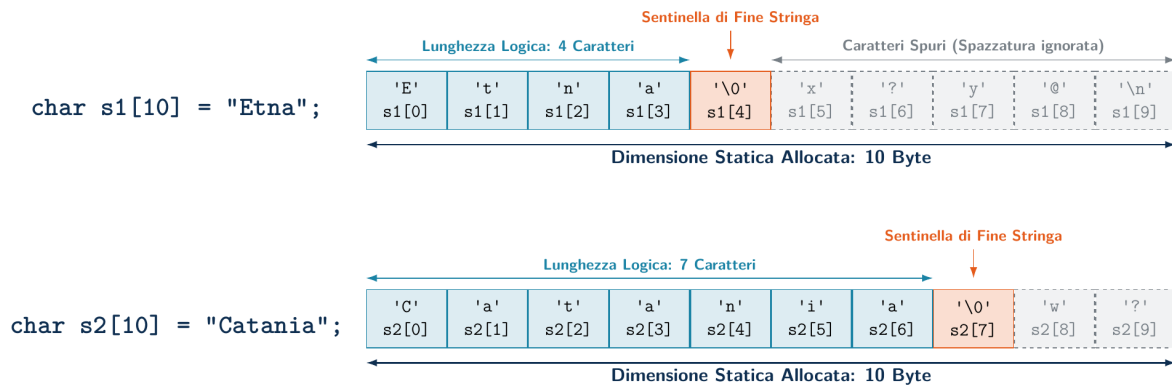


Figura 1.1: Diagramma: Anatomia delle Stringhe

1.1.4 Scansione di Stringhe ed Aritmetica dei Puntatori

Vediamo due tecniche per scansionare una stringa carattere per carattere fino al raggiungimento del terminatore.

```
1 #include <stdio.h>
2
3 int main(void) {
4     char str[] = "Etna";
5
6     // Versione A: Indicizzazione classica dell'array
7     printf("Versione A: ");
8     for (int i = 0; str[i] != '\0'; i++) {
9         putchar(str[i]);
10    }
11    putchar('\n');
12
13    // Versione B: Scansione con puntatori (Aritmetica dei puntatori)
14    printf("Versione B: ");
15    char *p = str; // p punta all'indirizzo iniziale str[0]
16    while (*p != '\0') {
17        putchar(*p);
18        p++; // Sposta l'indirizzo avanti di 1 byte (dimensione di un
19        ↪ char)
20    }
21    putchar('\n');
22
23    return 0;
24 }
```

Tagliare le stringhe con l'aritmetica

Poiché il nome dell'array `str` decade a puntatore al primo elemento, possiamo stampare sotto-stringhe senza copiare dati in nuovi array semplicemente variando l'indirizzo di partenza:

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

1.2 Manipolazione, Librerie Standard e Sicurezza

1.2.1 Analisi dei Caratteri con <ctype.h>

Per evitare l'uso di "numeri magici" legati alla tabella ASCII (es. `if (c >= 65 && c <= 90)` per individuare lettere maiuscole), il C mette a disposizione la libreria standard <ctype.h>. Questa contiene funzioni ottimizzate ed indipendenti dal set di caratteri utilizzato:

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 int main(void) {
5     char c = 'a';
6
7     if (isalpha(c)) {
8         printf("%c e' una lettera\n", c);
9     }
10    if (islower(c)) {
11        printf("Ed e' minuscola! In maiuscolo: '%c'\n", toupper(c));
12    }
13    return 0;
14 }
```

Principali funzioni di <ctype.h>:

- `isalpha(c)`: restituisce vero se `c` è una lettera.
- `isdigit(c)`: restituisce vero se `c` è una cifra decimale (0..9).
- `isalnum(c)`: restituisce vero se `c` è alfanumerico.
- `isspace(c)`: restituisce vero se `c` è uno spazio, tabulazione o ritorno a capo.
- `toupper(c)` / `tolower(c)`: converte il carattere.

1.2.2 Conversione Sicura di Stringhe (strtod vs atoi)

Quando acquisiamo dati testuali, abbiamo spesso la necessità di convertirli in valori numerici per effettuare operazioni matematiche.

Approccio Unsafe: atoi (e atof)

La funzione `atoi` (ASCII to Integer) della libreria <stdlib.h> è molto diffusa ma presenta un grave difetto di sicurezza:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     int val = atoi("123"); // Ritorna 123 (Corretto)
6     int err = atoi("ciao"); // Ritorna 0! (Fallimento)
7 }
```

Se `atoi` ritorna 0, è impossibile determinare se l'utente ha digitato effettivamente il valore numerico "0" o se la conversione è fallita per un input errato.

Approccio Safe: strtol e strtod

Le funzioni `strtol` (String to Long) e `strtod` (String to Double) consentono un controllo degli errori robusto accettando un puntatore a puntatore (`char **endptr`) che memorizza la cella di memoria in cui la conversione si è interrotta:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     char input[] = "137.56 euro";
6     char *endptr;
7
8     double prezzo = strtod(input, &endptr);
9
10    if (input == endptr) {
11        printf("Errore: nessuna conversione numerica trovata!\n");
12    } else {
13        printf("Prezzo convertito: %.2f\n", prezzo);           //
14        printf("Stringa non convertita rimanente: '%s'\n", endptr); //
15        printf("Stampa \" euro\"");
16    }
17    return 0;
18 }

```

1.2.3 Uso della Libreria `<string.h>`

La libreria standard `<string.h>` fornisce funzioni ottimizzate per manipolare le stringhe:

- `strlen(s)`: restituisce la lunghezza logica della stringa (escluso il terminatore `\{\}\{\}\0`).
- `strcpy(dest, src)`: copia il contenuto di `src` in `dest` (attenzione: `dest` deve avere abbastanza spazio allocato per contenere la copia e il terminatore!).
- `strcat(dest, src)`: concatena `src` in coda a `dest`.
- `strcmp(s1, s2)`: effettua un confronto lessicografico sottraendo i caratteri. Ritorna 0 se le stringhe sono identiche, un valore negativo se `s1 < s2`, positivo se `s1 > s2`.

Esempio completo di utilizzo di <string.h>

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void) {
5     char str1[50] = "Etna";
6     char str2[] = "Coder";
```

```
7  char buffer[50];
8
9  // Lunghezza logica
10 printf("Lunghezza di str1: %zu\n", strlen(str1)); // Stampa 4
11
12 // Copia
13 strcpy(buffer, str1);
14 printf("Buffer copiato: %s\n", buffer); // Stampa "Etna"
15
16 // Concatenazione
17 strcat(buffer, str2);
18 printf("Buffer concatenato: %s\n", buffer); // Stampa "Etna Coder"
19
20 // Confronto lessicografico
21 if (strcmp(str1, "Etna") == 0) {
22     printf("Le due stringhe sono identiche!\n");
23 }
24
25 int confronto = strcmp("albero", "barca");
26 printf("Confronto 'albero' vs 'barca': %d\n", confronto); // Ritorna
    ↪ valore < 0
27
28 return 0;
29 }
```

1.3 L'Input/Output di Testo Sicuro e le Matrici

1.3.1 La piaga del Buffer Overflow e la soluzione fgets

L'acquisizione classica di stringhe tramite `scanf("\{%s", buffer)` costituisce una delle principali vulnerabilità software storiche. Se `buffer` ha una dimensione limitata (es. 10 byte) e l'utente inserisce una parola di 30 caratteri, `scanf` continuerà a scrivere in RAM sovrascrivendo i record adiacenti e gli indirizzi di ritorno dello Stack Frame (Buffer Overflow).

Esempio Unsafe (Che manderemo in crash):

```
1  #include <stdio.h>
2
3  int main(void) {
4      char password[8] = "segreto"; // 7 caratteri + '\0'
5      char input[8];                // Buffer di input limitato a 8 byte
6
7      printf("Inserisci la password di sblocco: ");
8      // PERICOLO: scanf non sa quanto sia grande 'input'!
9      scanf("%s", input);
10
11     // Se l'utente digita una password molto lunga (es. "1234567890abcdef
    ↪ "),
12     // i caratteri in eccesso sborderanno in memoria sovrascrivendo
13     // la variabile 'password' o i registri dello Stack Frame.
14     printf("Hai inserito: %s\n", input);
```

```

15     printf("Password di controllo attuale: %s\n", password);
16
17     return 0;
18 }

```

Se provate ad eseguire questo codice compilato ed inserite una stringa molto lunga, vedrete che la variabile `password` verrà alterata dal vostro stesso input, o il sistema andrà in crash restituendo un errore del tipo `Illegal instruction` o `Segmentation Fault`.

Per risolvere questo problema si deve utilizzare **`fgets`**, che limita esplicitamente i byte letti:

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      char nome[20];
6      printf("Inserisci il tuo nome: ");
7
8      // Legge al massimo 19 caratteri lasciando il 20-esimo per il '\0'
9      if (fgets(nome, sizeof(nome), stdin) != NULL) {
10         // fgets acquisisce anche l'invio '\n', rimuoviamolo se presente
11         nome[strcspn(nome, "\n")] = '\0';
12         printf("Ciao %s!\n", nome);
13     }
14     return 0;
15 }

```

Svuotare il buffer residuo

Se leggiamo un valore numerico con `scanf` e subito dopo proviamo ad acquisire una stringa con `fgets`, quest'ultima leggerà l'invio (`\{\}\{\}\n`) rimasto intrappolato nel buffer di input di sistema. Per pulire il buffer residuo prima di una `fgets`:

```

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

```

1.3.2 Le Matrici (Linearizzazione in Row-Major Order)

Dal punto di vista logico, immaginiamo una matrice come una griglia bidimensionale `int M[Righe][Colonne]`. In realtà, **la memoria RAM è rigorosamente lineare (unidimensionale)**.

Il C risolve questa astrazione implementando il **Row-Major Order**: le righe della matrice vengono accodate consecutivamente in RAM.

Per calcolare l'indirizzo fisico in cui risiede l'elemento generico `M[i][j]`, la CPU applica la seguente formula di offset:

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

Notiamo che la CPU **non ha bisogno di conoscere il numero di righe complessivo**, ma deve necessariamente conoscere il **numero di colonne** per poter calcolare quanti elementi scavalcare per posizionarsi alla riga desiderata.

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)

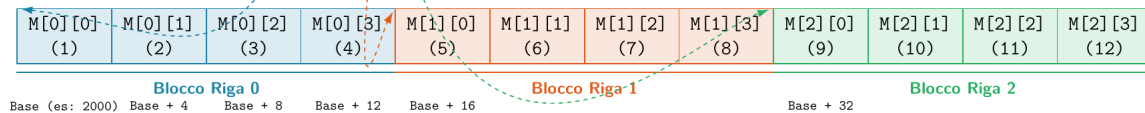


Figura 1.2: Diagramma: Linearizzazione delle Matrici

Ecco perché nella firma delle funzioni che accettano matrici è obbligatorio specificare la dimensione delle colonne:

```

1  #include <stdio.h>
2
3  // La colonna (4) serve alla CPU per il calcolo corretto dell'offset in
   ↪ RAM!
4  void stampa_matrice(int M[][4], int righe) {
5      for (int i = 0; i < righe; i++) {
6          for (int j = 0; j < 4; j++) {
7              printf("%d\t", M[i][j]);
8          }
9          printf("\n");
10     }
11 }
```

1.3.3 Matrici di Caratteri come Array di Stringhe

Una matrice bidimensionale di caratteri (`char arr[Righe][Colonne]`) può essere utilizzata in C come un **array di stringhe**. In questa astrazione, ciascuna riga rappresenta una stringa indipendente (il cui termine logico è stabilito dalla sentinella `\{\}\{\}\0`), mentre la dimensione delle colonne definisce la capacità fisica massima riservata per ogni elemento.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      // Definizione di un array di stringhe: 3 elementi, ciascuno con
   ↪ spazio per massimo 10 byte
6      char arr[3][10] = {
7          "Etna",
8          "Corso",
9          "Catania"
10     };
11 }
```

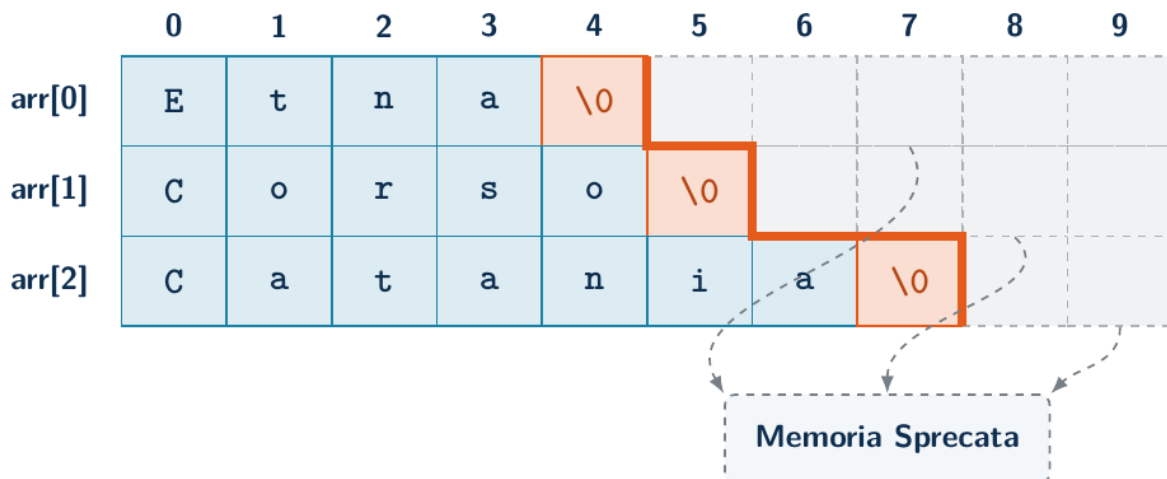


Figura 1.3: Diagramma: Rappresentazione in memoria di un array di stringhe e spreco di spazio

```

12 // arr[i] rappresenta l'indirizzo di inizio della riga i-esima (
    ↪ quindi della stringa corrispondente)
13 for (int i = 0; i < 3; i++) {
14     printf("Indice %d: %s\n", i, arr[i]);
15 }
16 return 0;
17 }

```

Rappresentazione in Memoria e lo Spreco di Spazio

Essendo la griglia statica della matrice fissa in memoria Stack, dichiarare `char arr[3][10]` impone al compilatore di allocare rigidamente $3 \times 10 = 30$ byte. Se una stringa memorizzata è più corta del limite massimo (ad esempio "Etna" che occupa solo 5 byte totali, incluso `\{\}\{\}\0`), i rimanenti byte di quella riga rimarranno allocati ma inutilizzati, causando uno spreco di spazio.

Operazioni Invalide sugli Array di Stringhe

Dato che `arr[0]` rappresenta l'indirizzo base di un array fisso (quindi un puntatore costante non riassegnabile), è **impossibile cambiare il valore di una stringa con una riassegnazione diretta**:

```

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

```

Per modificare i testi memorizzati all'interno dell'array di stringhe dobbiamo ricorrere alla copia profonda dei caratteri tramite `strcpy`:

```

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

```

Nelle lezioni successive vedremo come ovviare a questo spreco di memoria e abilitare la riassegnazione diretta dei puntatori impiegando gli **array di puntatori a stringhe** (`char *arr[]`).

1.3.4 Array N-Dimensionali

Poiché le dimensioni superiori sono solo astrazioni mentali, possiamo definire array a qualunque profondità desiderata. Per la CPU si tratterà sempre di una sequenza piatta in RAM.

Consideriamo un monitoraggio termodinamico di un plesso scolastico:

```
1 // 1D: Fila di 10 sensori lungo un tubo -> double tubo[10];
2 // 2D: Griglia di 10x10 sensori su un pavimento -> double pavimento
  ⇨ [10][10];
3 // 3D: Reticolo 10x10x5 sensori in una stanza volumetrica -> double
  ⇨ stanza[10][10][5];
4 // 4D: Misura 3D ad ogni ora di un giorno (24 ore) -> double log
  ⇨ [24][10][10][5];
5
6 // 5D: Monitoriamo 8 stanze diverse dello stesso plesso
7 double plesso[8][24][10][10][5];
```

Per accedere alla temperatura registrata nella Stanza 2, alle ore 14:00, alle coordinate spaziali $X = 4, Y = 7, Z = 1$:

```
1 double temp = plesso[2][14][4][7][1];
```

La CPU risolverà questo indirizzo espandendo ricorsivamente la formula di offset basandosi sulle dimensioni di ciascun blocco allocate a compile-time.