



Programmazione 1 (Canale O-Z)

# Lezione 11: Interazione con l'OS: CLI, Flussi di Testo e Tokenizzazione

---

Docente: Prof. Antonino Furnari

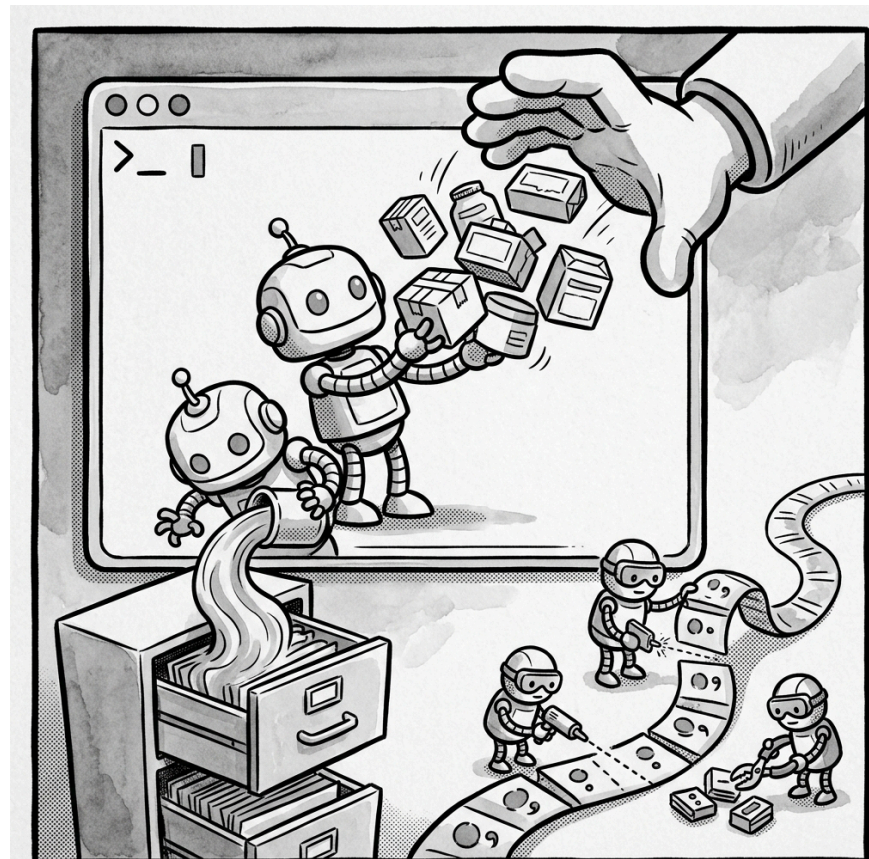
Email: `antonino.furnari@unict.it`

Sito Web: [antoninofurnari.github.io/prog1-2627](https://antoninofurnari.github.io/prog1-2627)



# Oltre l'Input Interattivo (CLI)

- Finora i programmi si fermavano ad attendere dati da tastiera ( `scanf` ).
- I comandi reali ( `cp foto.png backup/` , `ls` , `git` ) non fanno domande: ricevono i parametri **al momento del lancio**.
- Così i comandi si possono concatenare e **automatizzare** negli script.



? Una calcolatrice da terminale che chiede "Inserisci il primo numero" [Invio] ogni volta: ha senso? E se scrivessimo `./calcola 15 + 27`?



# L'Anatomia di `argc` e `argv`

Per ricevere i parametri di lancio estendiamo la firma del `main`:

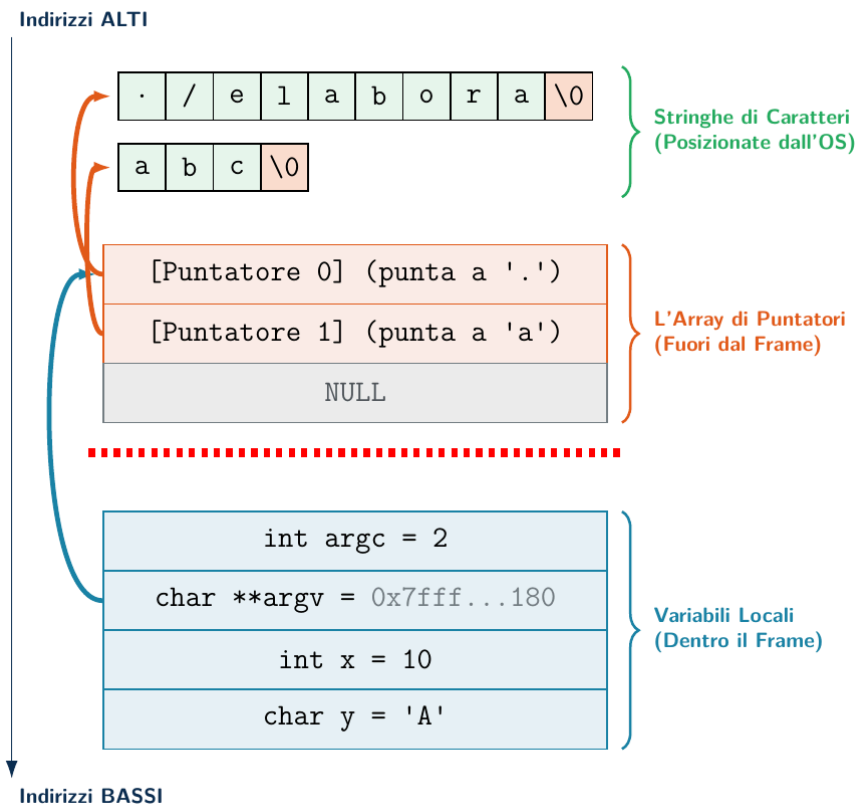
```
int main(int argc, char *argv[])
```

- `argc` (**Argument Count**): numero di argomenti digitati, **incluso** il nome del programma. Sempre  $\geq 1$ .
- `argv` (**Argument Vector**): array di puntatori a caratteri (stringhe):
  - `argv[0]` ➡ nome dell'eseguibile (es. `"./elabora"`);
  - `argv[1]`, ..., `argv[argc-1]` ➡ i parametri veri e propri;
  - `argv[argc]` ➡ `NULL`, sentinella di fine array.



# argc e argv nello Stack

- Errore comune: pensare che le stringhe di `argv` stiano in **RoData** (sola lettura).
- All'avvio il *loader* dell'OS deposita i testi degli argomenti **all'inizio assoluto dello Stack** (indirizzi alti), seguiti dall'array `argv[]`.
- Solo dopo si alloca lo Stack Frame del `main()`.
- ➡ Le stringhe sono **mutabili**: `argv[1][0] = 'X'` non causa segmentation fault.





# Codice C: Stampare e Modificare gli Argomenti

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

int main(int argc, char *argv[]) {
    printf("Hai lanciato il programma con %d argomenti.\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d]: \"%s\"\n", i, argv[i]);
    }

    // Intercettare un'opzione di aiuto
    if (argc > 1 && strcmp(argv[1], "--help") == 0) {
        printf("\nUso: ./programma [valore_1] [operatore] [valore_2]\n");
    }

    // Dimostrazione di mutabilità nello Stack
    if (argc > 1) {
        argv[1][0] = 'X';
        printf("Argomento 1 modificato fisicamente: %s\n", argv[1]);
    }
    return 0;
}
```



## Lanciamolo con dei Parametri

```
gcc -Wall args.c -o args  
./args ciao 42
```

Hai lanciato il programma con 3 argomenti.

argv[0]: "./args"

argv[1]: "ciao"

argv[2]: "42"

Argomento 1 modificato fisicamente: Xiao



**Live:** provare `./args` senza parametri (argc vale 1) e `./args "due parole"` (le virgolette fanno un solo argomento).



## E con `--help`?

```
./args --help
```

Il controllo di `--help` avviene **prima** della modifica di `argv[1]`, quindi compare l'aiuto:

```
Hai lanciato il programma con 2 argomenti.
```

```
argv[0]: "./args"
```

```
argv[1]: "--help"
```

```
Uso: ./programma [valore_1] [operatore] [valore_2]
```

```
Argomento 1 modificato fisicamente: X-help
```

## I Tre Flussi Standard

Ogni processo C ha, fin dall'avvio, tre canali identificati da un *File Descriptor*:

1. `stdin` (FD 0): ingresso dei dati (tastiera).
2. `stdout` (FD 1): uscita standard dei dati (schermo).
3. `stderr` (FD 2): uscita **indipendente** per errori e diagnostica.

? Se stampiamo "Impossibile aprire il file" con `printf`, dove finisce? E se un altro programma sta leggendo il nostro output per fare calcoli?

## Lo Standard Error e `fprintf`

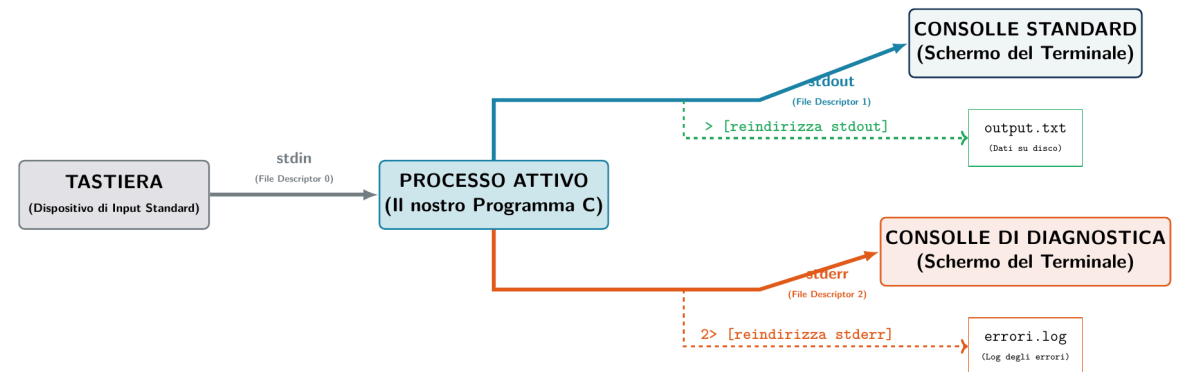
Con `fprintf` (la *f* sta per file/stream) scegliamo il canale di destinazione:

```
fprintf(stdout, "Questo e' un dato normale.\n"); // Equivalente a printf(...)
fprintf(stderr, "ERRORE CRITICO: Parametro non valido!\n"); // Va su stderr
```

- Con `printf` gli errori finiscono su `stdout` **mescolati** ai dati utili.
- Con `stderr` dati puliti e diagnostica restano **separati**.

# Reindirizzamento dei Flussi da Terminale

- Solo `stdout` su file: `make > output.txt` (gli errori restano a schermo).
- Solo `stderr` su file: `./programma 2> errori.log`
- Entrambi, separati: `./programma > output.txt 2> errori.log`
- Uniti in un file: `./programma > tutto.txt 2>&1` (canale 2 deviato nel canale 1).



 **Live:** programma con le due `fprintf` sopra (`gcc -Wall flussi.c -o flussi`); lanciare `./flussi > output.txt`: a schermo resta solo l'errore. Poi `cat output.txt`.

## ! Le Insidie di `scanf`: il Tipo Errato

```
int eta;  
printf("Inserisci la tua eta': ");  
scanf("%d", &eta);  
printf("Hai inserito: %d\n", eta);
```

- Se l'utente digita `A`, la conversione **fallisce** e `eta` non viene toccata.
- Se non inizializzata, `eta` contiene **spazzatura** dello stack.
- La `'A'` resta **incastrata nel buffer**: blocca o manda in loop le letture successive.

 **Live:** `gcc -Wall eta.c -o eta && ./eta`, digitare `A` e osservare il valore stampato.

## ! Le Insidie di `scanf` : gli Spazi

```
char nome[50];  
printf("Inserisci nome e cognome: ");  
scanf("%s", nome);  
printf("Benvenuto %s!\n", nome);
```

```
Inserisci nome e cognome: Antonino Furnari  
Benvenuto Antonino!
```

- `scanf("%s", ...)` si ferma al **primo carattere di spaziatura** (spazio, tab, invio).
- ➡ Non può acquisire frasi complete.

? Come rendiamo la lettura robusta?

## Regola 1: Controllare il Ritorno di `scanf`

`scanf` restituisce il **numero di elementi convertiti** con successo:

```
int eta = 0;
printf("Inserisci la tua eta': ");
if (scanf("%d", &eta) != 1) {
    fprintf(stderr, "Errore: Inserire un valore numerico valido!\n");
    // Gestione dell'errore e svuotamento del buffer...
}
```

- Inizializzare `eta = 0` evita comunque la spazzatura.
- Il messaggio d'errore va su `stderr`.

## Regola 2: `fgets` per le Stringhe

```
char nome[50];  
printf("Inserisci nome e cognome: ");  
fgets(nome, sizeof(nome), stdin); // Legge in sicurezza fino a 49 caratteri + '\0'
```

- Legge un'intera riga, spazi compresi.
- Rispetta la dimensione massima ➡ previene il **Buffer Overflow**.
- Cattura anche l'invio finale `\n`, che di solito si rimuove:

```
nome[strcspn(nome, "\n")] = '\0';
```

## `getchar()` e `EOF` : Perché Ritorna `int` ?

```
int getchar(void);
```

- Estrae **un singolo byte** dallo stream di ingresso.
- Deve restituire tutti i **256** valori di un carattere (0–255) **più** `EOF` (End Of File, solitamente `-1`).
- Lo standard non dice se `char` sia `signed` o `unsigned`. Se ritornasse `char`:
  - **signed**: il byte `0xFF` (255) diventerebbe `-1` ➡ scambiato per `EOF`;
  - **unsigned**: `EOF` diventerebbe `255` ➡ scambiato per un carattere valido, ciclo infinito.
- Con `int`, `EOF` resta **fuori** dall'intervallo dei caratteri validi.



# getchar : Costruire una Stringa (1/2)

```
#include <stdio.h>

int main(void) {
    char stringa[50];
    printf("Inserisci la stringa (max 49 car. - premi invio per confermare): ");
    int len = 0;
    int c;    // int, non char!

    // Lettura manuale a basso livello con getchar()
    do {
        c = getchar();
        // Escludiamo il newline e l'EOF, rispettando il limite fisico del buffer
        if (c != '\n' && c != EOF && len < 49) {
            stringa[len] = (char)c;
            len++;
        }
    } while (c != EOF && c != '\n' && len < 49);
```



## getchar: ...e Invertirla (2/2)

```
// Inserimento manuale del terminatore di stringa
stringa[len] = '\0';

printf("\nStringa invertita: ");
for (int i = len - 1; i ≥ 0; i--) {
    printf("%c", stringa[i]);
}
printf("\n");

return 0;
}
```

- Senza `'\0'` finale, `stringa` non sarebbe una stringa C valida.
- La stampa parte dall'**ultimo** carattere valido (`len - 1`).



## getchar : Proviamolo

Inserisci la stringa (max 49 car. - premi invio per confermare): Ciao mondo

Stringa invertita: odnom oaiC

- Gli **spazi** vengono letti come qualsiasi altro carattere.
- Il terminatore `'\0'` va aggiunto **a mano**.

 **Live:** gcc -Wall inverti.c -o inverti && ./inverti; poi premere Ctrl + D senza digitare nulla per inviare EOF.

## Persistenza su File: il Tipo `FILE` \*

`FILE` è una struttura di `<stdio.h>` con lo stato del file (cursore, buffer, flag di errore).  
Noi usiamo solo il puntatore:

- `fopen("nome", "modo")` apre il file:
  - `"r"` lettura (restituisce `NULL` se il file è assente);
  - `"w"` scrittura (crea un file vuoto, **cancellando** il contenuto preesistente);
  - `"a"` append (scrive in coda senza cancellare).
- **Lettura/scrittura:** `fprintf(fp, ...)`, `fscanf(fp, ...)`, `fgets(..., fp)`.
- `fclose(fp)` è **obbligatoria**: svuota i buffer e garantisce che i dati arrivino sul disco.

# ★ La Trappola di `fscanf` nei File Reali

Un file di esami con una riga malformata:

```
Programmazione_1 9 27  
Analisi_1 9 ASSENTE  
Fisica 6 18
```

- `fscanf(fp, "%s %d %d", ...)` prova a convertire `ASSENTE` in intero e **fallisce**.
- I caratteri incongruenti **restano nel buffer** dello stream.
- ➡ Loop infinito o tutte le righe successive saltate.

? Come leggiamo un file senza che una sola riga corrotta blocchi tutto?

## Robust Line-by-Line Parsing

Separare la **lettura** dalla **decodifica**:

1. Acquisire l'intera riga "grezza" come stringa con `fgets()`.
2. Decodificare (tokenizzare) **solo la stringa in RAM**.
  - Se una riga è corrotta, la decodifica fallisce **localmente**, senza bloccare lo stream.
  - Segnaliamo l'anomalia su `stderr` e passiamo alla riga successiva.

## Strumento A: `sscanf`

Come `scanf` / `fscanf`, ma legge da una **stringa in RAM** invece che da un canale:

```
char riga[] = "Programmazione_1 9 27";
char materia[50];
int cfu, voto;

if (sscanf(riga, "%49s %d %d", materia, &cfu, &voto) == 3) {
    printf("Letto: %s (%d CFU) - Voto: %d\n", materia, cfu, voto);
}
```

Letto: Programmazione\_1 (9 CFU) - Voto: 27

- `%49s` limita la lettura alla dimensione del buffer.
- Controlliamo che i campi convertiti siano esattamente **3**.



## Strumento B: strtok (Tokenizzatore In-Place)

- In `<string.h>`: divide una stringa in **token** separati da delimitatori (es. `,`).
- **Prima chiamata:** si passa la stringa, `strtok(riga, ",")`.
- **Chiamate successive:** `strtok(NULL, ",")` riprende da dove era arrivata.
- **⚠ Distruttiva:** sovrascrive ogni delimitatore con `'\0'`.
- Con una costante (`char *s = "Etna,Corso,Catania"`) ➡ *Segmentation Fault*.

### 0. Stato Iniziale in RAM

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|
| E | t | n | a | , | C | o | r | s | o | , | C | a | t | a | n | i | a | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|

Puntatore Statico  
(Prossima partenza)

### 1. Chiamata: `strtok(riga, ",")`

|   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|----|
| E | t | n | a | \0 | C | o | r | s | o | , | C | a | t | a | n | i | a | \0 |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|----|

↑ tok  
(ritorna "Etna")

Puntatore Statico

### 2. Chiamata: `strtok(NULL, ",")`

|   |   |   |   |    |   |   |   |   |   |    |   |   |   |   |   |   |   |    |
|---|---|---|---|----|---|---|---|---|---|----|---|---|---|---|---|---|---|----|
| E | t | n | a | \0 | C | o | r | s | o | \0 | C | a | t | a | n | i | a | \0 |
|---|---|---|---|----|---|---|---|---|---|----|---|---|---|---|---|---|---|----|

↑ tok  
(ritorna "Corso")

Puntatore Statico

### 3. Chiamata: `strtok(NULL, ",")`

|   |   |   |   |    |   |   |   |   |   |    |   |   |   |   |   |   |   |    |
|---|---|---|---|----|---|---|---|---|---|----|---|---|---|---|---|---|---|----|
| E | t | n | a | \0 | C | o | r | s | o | \0 | C | a | t | a | n | i | a | \0 |
|---|---|---|---|----|---|---|---|---|---|----|---|---|---|---|---|---|---|----|

↑ tok  
(ritorna "Catania")

NULL



# La Trappola del Terminatore

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

int main() {
    char str[] = "Ciao,sono,io";
    char *a = strtok(str, ",");
    while (a != NULL) {
        printf("%s\n", a);
        a = strtok(NULL, ",");
    }
    printf("Stringa finale: %s\n", str);
    return 0;
}
```

? Cosa stampa l'ultima `printf`? Seguiamo la memoria di `str` chiamata per chiamata.

## strtok Passo 1: strtok(str, ",")

| 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|----|---|---|---|---|---|----|----|----|
| C | i | a | o | \0 | s | o | n | o | , | i  | o  | \0 |

- La prima `,` (indice 4) diventa `'\0'`.
- `a` punta a `str[0]` ➡ stampa `Ciao`.

## strtok Passo 2: strtok(NULL, ",")

| 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12 |
|---|---|---|---|----|---|---|---|---|----|----|----|----|
| C | i | a | o | \0 | s | o | n | o | \0 | i  | o  | \0 |

- Riparte dall'indice 5; la , all'indice 9 diventa '\0'.
- a punta a str[5] ➡ stampa sono.



## strtok Passo 3 e 4: fine dei Token

| 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12 |
|---|---|---|---|----|---|---|---|---|----|----|----|----|
| C | i | a | o | \0 | s | o | n | o | \0 | i  | o  | \0 |

- Passo 3: riparte dall'indice 10, nessun delimitatore ➡ a punta a `str[10]`, stampa `io`.
- Passo 4: stringa esaurita ➡ `strtok` restituisce `NULL`, il ciclo termina.

# La Trappola del Terminatore: Risposta

```
Ciao  
sono  
io  
Stringa finale: Ciao
```

- `str` punta ancora a `str[0]`, ma all'indice 4 c'è un `'\0'` inserito da `strtok`.
- `printf("%s")` si ferma al **primo** `'\0'`: "sono" e "io" sono ancora in RAM, ma per il C la stringa è tagliata.

 **Live:** `gcc -Wall trappola.c -o trappola && ./trappola`; poi sostituire `char str[]` con `char *str` e osservare il crash.

## Cosa Abbiamo Imparato Oggi

- `int main(int argc, char *argv[])`: parametri di lancio, `argv[0]` è il programma, `argv[argc]` è `NULL`; le stringhe sono nello **Stack** e mutabili.
- Tre flussi: `stdin` (0), `stdout` (1), `stderr` (2); errori con `fprintf(stderr, ...)`; redirectione `>`, `2>`, `2>&1`.
- `scanf`: controllare **sempre** il valore di ritorno; per le stringhe `fgets`.
- `getchar` restituisce `int` per distinguere `EOF` dai 256 caratteri.
- File: `fopen` / `fclose`; parsing robusto riga per riga con `fgets` + `sscanf` / `strtok`.

 **Per casa:** esercizi della Lezione 11 sul sito del corso.

**Grazie per l'attenzione!**

**Sito Web del Corso:** [antoninofurnari.github.io/prog1-2627](https://antoninofurnari.github.io/prog1-2627)