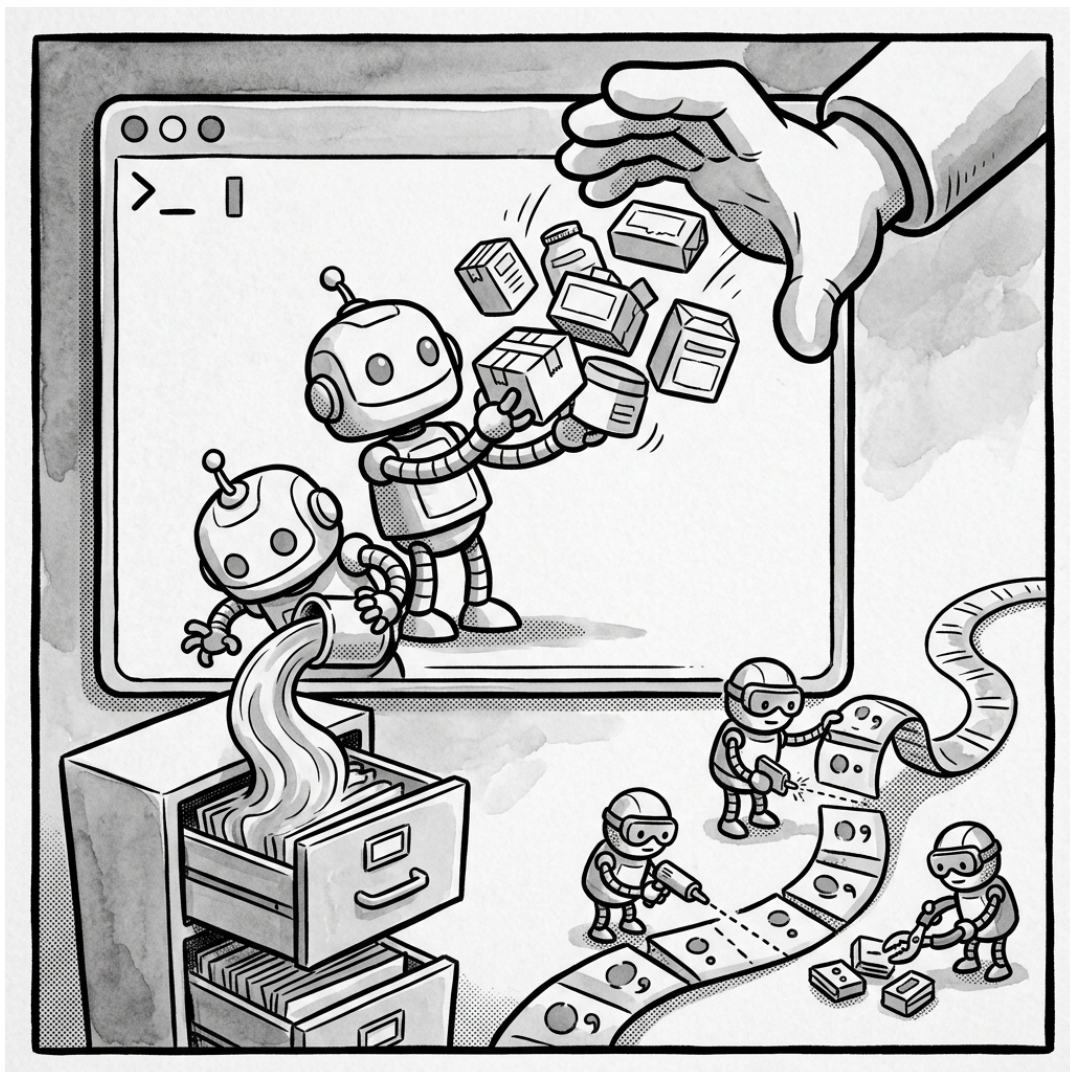


LEZIONE 11

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



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

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

Fino ad oggi, i nostri programmi hanno interagito con l'utente in modo esclusivamente interattivo, fermandosi ad attendere dati digitati da tastiera a run-time. Tuttavia, nello sviluppo software professionale e nell'automazione di sistema, i programmi devono poter ricevere parametri direttamente all'avvio dal terminale (Command Line Interface) ed elaborare file in modo robusto senza bloccarsi davanti a dati corrotti.

In questa lezione vedremo come il Sistema Operativo passa parametri al processo (l'anatomia di `argc` e `argv` in memoria), come differenziare i dati dall'output diagnostico usando lo **Standard Error (`stderr`)**, come implementare letture stabili prevenendo le trappole intrinseche di `scanf` e, infine, come effettuare il parsing robusto e la tokenizzazione in memoria RAM tramite `sscanf` e `strtok`.

1.1 La Riga di Comando e lo Standard Error (CLI & STDERR)

1.1.1 Il Limite dell'Isolamento interattivo

I comandi di sistema professionali (come `cp`, `ls` o `git`) non avviano dialoghi interattivi con domande del tipo *"Inserisci la sorgente:"* o *"Inserisci la destinazione:"*. Essi ricevono le istruzioni operative direttamente al momento del lancio, ad esempio scrivendo `cp foto.png backup/`. Questo modello consente di concatenare ed automatizzare i comandi all'interno di script. In C, questa interazione viene gestita tramite due parametri speciali passati alla funzione `main`.

1.1.2 L'Anatomia di `argc` e `argv` nello Stack

Per accogliere i parametri di lancio, definiamo il `main` con la seguente firma:

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

- **`argc` (Argument Count):** Un intero che conta il numero totale di argomenti digitati sul terminale, incluso il nome stesso del programma. È sempre ≥ 1 .

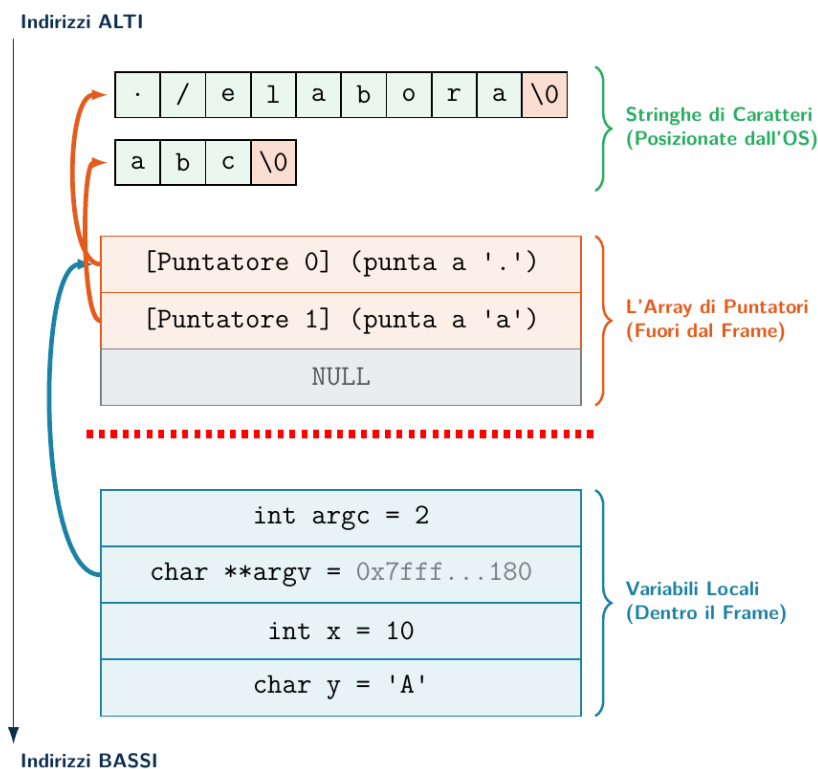


Figura 1.1: Anatomia di argc e argv nello Stack

- **argv (Argument Vector):** Un array di puntatori a caratteri (stringhe).
 - argv[0] contiene sempre la stringa con il nome dell'eseguibile (ad es. ". / e l a b o r a").
 - argv[1], argv[2], ..., argv[argc-1] contengono i parametri di lancio veri e propri.
 - argv[argc] è un puntatore nullo (NULL), che serve come sentinella di fine array.

La mutabilità di argv

Un errore comune è ritenere che le stringhe lette da argv si trovino nell'area di memoria a sola lettura (RoData). In realtà, all'avvio del processo, il *loader* del Sistema Operativo deposita i testi degli argomenti **all'inizio assoluto della memoria Stack** (agli indirizzi fisici più alti), seguiti dall'array di puntatori argv. Solo dopo viene allocato lo Stack Frame della funzione main.

Essendo allocate nello Stack, le stringhe di input sono **mutabili**: il programma può scriverci dentro e alterarle (es. argv[1][0] = 'X') senza provocare crash di segmentazione.

Codice C: Stampare e modificare gli argomenti a riga di comando

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

```

3
4 int main(int argc, char *argv[]) {
5     printf("Hai lanciato il programma con %d argomenti.\n", argc);
6
7     for (int i = 0; i < argc; i++) {
8         printf("argv[%d]: \"%s\"\n", i, argv[i]);
9     }
10
11     // Intercettare un'opzione di aiuto
12     if (argc > 1 && strcmp(argv[1], "--help") == 0) {
13         printf("\nUso: ./programma [valore_1] [operatore] [valore_2]\n");
14     }
15
16     // Dimostrazione di mutabilit  nello Stack
17     if (argc > 1) {
18         argv[1][0] = 'X';
19         printf("Argomento 1 modificato fisicamente: %s\n", argv[1]);
20     }
21
22     return 0;
23 }

```

1.1.3 Lo Standard Error (stderr)

Se un'applicazione stampa un messaggio di errore (come *"Impossibile aprire il file"*) usando una normale `printf`, questo messaggio finisce sullo **Standard Output (stdout)** insieme ai dati regolari prodotti dal programma. Ci  rende difficile separare i dati puliti dai diagnostici d'errore.

Per ovviare a questo problema, ogni programma C dispone fin dall'avvio di tre flussi standard (identificati a basso livello da numeri detti *File Descriptor*):

1. **stdin (File Descriptor 0)**: Canale di ingresso dei dati (tastiera).
2. **stdout (File Descriptor 1)**: Canale di uscita standard dei dati (schermo del terminale).
3. **stderr (File Descriptor 2)**: Canale di uscita indipendente dedicato a errori e messaggi diagnostici.

Utilizzando la funzione `fprintf` (dove la *f* sta per file/stream) possiamo deviare i messaggi sul canale desiderato:

```

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

```

Reindirizzamento dei Flussi da Terminale

L'esistenza di canali separati permette al terminale di effettuare operazioni di redirectione:

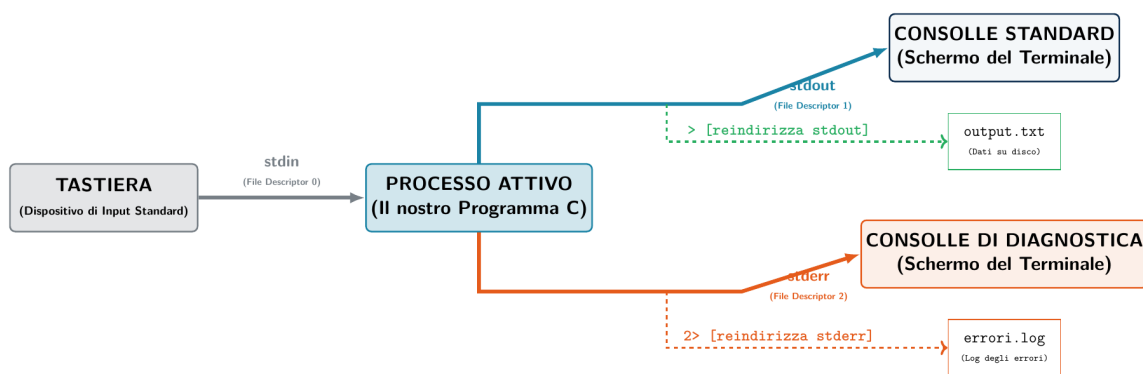


Figura 1.2: Architettura dei Flussi Standard Unix

- **Devviare solo stdout su file:** `make > output.txt` (lo schermo rimarrà pulito, i dati andranno nel file, ma gli eventuali errori verranno comunque stampati a schermo).
- **Devviare solo stderr su file:** `./programma 2> errori.log` (i dati corretti vanno a schermo, gli errori vengono isolati nel log).
- **Devviare entrambi su file separati:** `./programma > output.txt 2> errori.log`
- **Unire stderr a stdout su unico file:** `./programma > tutto.txt 2>\{\}&1` (devia il canale 2 nel canale 1).

1.2 I Limiti dell'Input e l'Acquisizione Robusta

1.2.1 Le insidie nascoste di scanf

La funzione `scanf` è pensata per l'acquisizione formattata rapida, ma presenta gravi limiti di stabilità in scenari reali:

1. **Il problema del tipo errato:** Se il codice richiede un numero (`scanf("\{%d", \{\}&eta)`) e l'utente inserisce la lettera 'A', la funzione fallisce la conversione e restituisce immediatamente il controllo al programma senza valorizzare la variabile. Se quest'ultima non è inizializzata, conterrà dati casuali dello stack ("spazzatura"). Inoltre, la 'A' rimarrà incastrata nel buffer di sistema, mandando in blocco o in loop infinito le successive chiamate di lettura.
2. **Il limite degli spazi:** La lettura di stringhe tramite `scanf("\{%s", buffer)` si interrompe sistematicamente davanti al primo carattere di spaziatura (spazio, tabulazione, invio). Non è quindi possibile acquisire stringhe contenenti frasi complete.

1.2.2 Tecniche di Lettura Sicura

Per garantire la stabilità del software, si applicano due regole fondamentali:

Regola 1: Verificare sempre il valore di ritorno di scanf

La funzione `scanf` restituisce il numero di elementi convertiti con successo. Dobbiamo sempre verificarne il valore:

```
1 int eta = 0;
2 printf("Inserisci la tua eta': ");
3 if (scanf("%d", &eta) != 1) {
4     fprintf(stderr, "Errore: Inserire un valore numerico valido!\n");
5     // Gestione dell'errore e svuotamento del buffer...
6 }
```

Regola 2: Usare fgets per le stringhe

Per acquisire testi completi, si deve usare `fgets(buffer, dimensione, stream)`, che legge un'intera riga (inclusi gli spazi intermedi) e previene attacchi di *Buffer Overflow* rispettando rigorosamente la dimensione massima specificata:

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

1.2.3 EOF, getchar e la lettura carattere per carattere

Per avere il controllo completo sull'input o svuotare il buffer, si ricorre alla lettura a basso livello tramite `getchar()`.

Perché getchar() restituisce int e non char?

`getchar()` estrae un singolo byte dallo stream di ingresso. Essa deve poter restituire tutti i 256 possibili valori di un carattere (da 0 a 255), più una condizione speciale che indichi la fine dell'input o del file: la costante **EOF (End Of File)**, definita solitamente come `-1`.

Se `getchar()` restituisse un tipo `char`, il comportamento varierebbe a seconda del compilatore, poiché lo standard C non specifica se il `char` base sia `signed` o `unsigned`:

- **Se char è signed:** il byte `0xFF` (valore ASCII 255) verrebbe letto come `-1`, venendo scambiato erroneamente per il segnale di EOF.
- **Se char è unsigned:** il segnale di EOF (`-1`) verrebbe convertito automaticamente in 255, venendo scambiato per un carattere valido e causando un ciclo infinito.

L'uso di un tipo `int` permette di mappare in sicurezza EOF al di fuori dell'intervallo dei caratteri validi, mantenendo distinti tutti i 256 valori positivi dal segnale di fine file.

Codice C: Costruire stringhe carattere per carattere invertendole

```
1  #include <stdio.h>
2
3  int main(void) {
4      char stringa[50];
5      printf("Inserisci la stringa (max 49 car. - premi invio per
   ↳ confermare): ");
6
7      int len = 0;
8      int c;
9
10     // Lettura manuale a basso livello con getchar()
11     do {
12         c = getchar();
13         // Escludiamo il newline e l'EOF, rispettando il limite fisico
   ↳ del buffer
14         if (c != '\n' && c != EOF && len < 49) {
15             stringa[len] = (char)c;
16             len++;
17         }
18     } while (c != EOF && c != '\n' && len < 49);
19
20     // Inserimento manuale del terminatore di stringa
21     stringa[len] = '\0';
22
23     printf("\nStringa invertita: ");
24     for (int i = len - 1; i >= 0; i--) {
25         printf("%c", stringa[i]);
26     }
27     printf("\n");
28
29     return 0;
30 }
```

1.3 Persistenza, File Stream e Tokenizzazione in Memoria

1.3.1 Apertura e Chiusura dei File: il tipo FILE

La persistenza delle informazioni viene gestita tramite puntatori a descrittori di file (FILE *). La struttura FILE racchiude i dettagli di stato del file gestiti dal Sistema Operativo (posizione corrente del cursore, buffer, flag di errore).

- **fopen("nome", "modo")**: Apre un file. I modi principali sono "r" (lettura, restituisce NULL se il file è assente), "w" (scrittura, crea un file vuoto cancellando l'eventuale contenuto preesistente) e "a" (append, scrive in coda senza cancellare).
- **fclose(fp)**: Chiude lo stream. Questa chiamata è fondamentale per forzare lo svuotamento dei buffer in memoria e garantire che i dati vengano fisicamente scritti sul disco prima che il programma termini.

1.3.2 Il Pericolo di `fscanf` nei File Reali

Cercare di interpretare un file di dati strutturati (es. un elenco CSV) tramite istruzioni dirette quali `fscanf(fp, "\\{}%s \\{}%d", ...)` è rischioso. Se una singola riga del file risulta malformata o contiene una stringa al posto di un numero (ad es. `Analisi\\{}_1 9 ASSENTE` invece di `Analisi\\{}_1 9 27`), la conversione fallisce, i dati incongruenti rimangono memorizzati nel buffer del file stream e il programma entrerà in loop infinito o salterà erroneamente tutte le righe rimanenti del file.

1.3.3 Robust Line-by-Line Parsing

La soluzione ingegneristica standard consiste nel dividere il processo di lettura dal processo di decodifica:

1. Si acquisisce l'intera riga "grezza" come stringa indipendente in memoria RAM usando la sicura `fgets()`.
2. Si effettua il parsing e la decodifica (tokenizzazione) della sola stringa in memoria RAM. In questo modo, se una riga è corrotta, la decodifica fallirà localmente senza bloccare il file stream. Il programma potrà segnalare l'anomalia su `stderr` e passare tranquillamente alla riga successiva.

1.3.4 Strumenti di Tokenizzazione: `sscanf` e `strtok`

Strumento A: La funzione `sscanf`

`sscanf` funziona in modo speculare a `scanf/fscanf`, ma estrae i dati analizzando una stringa in memoria RAM invece che un canale fisico:

```
1 char riga[] = "Programmazione_1 9 27";
2 char materia[50];
3 int cfu, voto;
4
5 if (sscanf(riga, "%49s %d %d", materia, &cfu, &voto) == 3) {
6     printf("Letto: %s (%d CFU) - Voto: %d\\n", materia, cfu, voto);
7 }
```

Strumento B: La funzione `strtok` (In-Place Tokenizer)

Definita in `<string.h>`, `strtok` divide una stringa in sotto-stringhe (token) separate da uno o più caratteri delimitatori (es. la virgola per file CSV).

Attenzione

Attenzione `strtok` è una funzione **distruttiva**: opera *in-place* alterando la stringa originaria. Ogni volta che individua un delimitatore, lo sovrascrive inserendovi il terminatore di stringa `'\\{}\\{}0'`. Per questa ragione, passargli come argomento una stringa costante non modificabile (ad es. `char *s = "Etna,Corso,Catania"`) provocherà un crash immediato per violazione di accesso in scrittura (*Segmentation*

0. Stato Iniziale in RAM

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

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

Puntatore Statico (Prossima partenza)

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

↑ tok
(ritorna "Etna")

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

Puntatore Statico

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

↑ tok
(ritorna "Corso")

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

Puntatore Statico

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

↑ tok
(ritorna "Catania")

NULL

Figura 1.3: Funzionamento interno di `strtok` in RAM

Fault).

Per mantenere traccia della posizione durante l'analisi, `strtok` adotta una logica specifica:

- Alla **prima chiamata**, si passa la stringa da dividere come primo argomento: `strtok(riga, ",")`.
- Alle **chiamate successive**, si passa `NULL` come primo parametro per indicare alla funzione di riprendere l'analisi dallo stato memorizzato internamente: `strtok(NULL, ",")`.

La Trappola del Terminatore

Cosa stampa l'istruzione `printf` alla fine di questo codice?

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char str[] = "Ciao,sono,io";
6     char *a = strtok(str, ",");
7     while (a != NULL) {
8         printf("%s\n", a);
    
```

```

9      a = strtok(NULL, ",");
10     }
11     printf("Stringa finale: %s\n", str);
12     return 0;
13 }
```

L'istruzione finale stamperà a schermo solo: **Ciao**. Poiché `strtok` ha modificato la prima virgola in `\{\}\{\}\0`, qualunque funzione standard (come la `printf` con `\{\}%s`) arresterà la lettura all'indirizzo iniziale non appena incontra la prima sentinella `\{\}\{\}\0`, ignorando le parole successive che pure risiedono ancora intatte in memoria RAM.

1.4 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Dove vengono collocate fisicamente in memoria le stringhe passate come argomenti tramite `argv`?

- A) Nel segmento RoData (sola lettura), trattandosi di costanti letterali.
- B) All'inizio assoluto dello Stack, subito prima dello Stack Frame del `main()`.
- C) Nello Heap, allocate dinamicamente dal sistema operativo.
- D) Nel segmento BSS, in quanto variabili globali non inizializzate.

Spiegazione: Le stringhe dei parametri di riga di comando vengono caricate nello Stack dal loader dell'OS prima dell'inizio dell'esecuzione del programma. Questo le rende mutabili (possono essere modificate scrivendo in `argv[i]`).

Risposta corretta: B

Soluzione

Quiz 2

Domanda

2. Cosa accade se si reindirizza l'output di un programma tramite comando `myapp > out.txt` ed il programma riscontra un errore che stampa su `stderr`?

- A) L'errore viene scritto nel file `out.txt` insieme ai dati ordinari.
- B) L'errore viene stampato comunque sullo schermo del terminale.
- C) Il programma viene interrotto istantaneamente dall'OS.
- D) L'errore viene scartato silenziosamente.

Spiegazione: Il reindirizzamento con `>` (o `>>`) devia solo lo standard output (`stdout`), lasciando invariata la destinazione di `stderr` che continua a essere diretta allo schermo del terminale (salvo reindirizzone esplicita con `>>`).

Risposta corretta: B

Soluzione

Quiz 3

Domanda

3. Perché il tipo di ritorno della funzione standard `getchar()` è `int` e non `char`?

- A) Perché deve consentire l'acquisizione di numeri interi completi in formato testuale.
- B) Perché deve poter restituire un valore speciale (EOF, pari a `-1`) distinto da tutti i caratteri validi, evitando che su macchine con `char` con segno il byte `0xFF` venga scambiato per fine file, o che con `char` senza segno EOF si trasformi in 255.
- C) Per uniformità con lo standard POSIX che richiede solo tipi a 32 o 64 bit.
- D) Per evitare overflow aritmetici nello stack frame.

Spiegazione: Se `getchar()` restituisse `char`, lo standard C non specifica se sia con o senza segno. Se fosse signed, il carattere valido 255 (`0xFF`) verrebbe promosso a `-1`, confondendosi con EOF. Se fosse unsigned, EOF (`-1`) verrebbe convertito in 255, impedendo il rilevamento di fine file e causando cicli infiniti.

Risposta corretta: B

Soluzione

Quiz 4

Domanda

4. Cosa succede se si passa una costante di stringa definita con le virgolette (ad es. `char *s = "dati,csv";`) come primo parametro della funzione `strtok`?

- A) La stringa viene duplicata nello Heap e tokenizzata in sicurezza.
- B) La funzione sostituisce la prima virgola con un terminatore senza effetti collaterali.
- C) Il programma subisce un crash istantaneo (*Segmentation Fault*) per tentativo di scrittura su memoria protetta.
- D) Viene restituito un puntatore nullo (`NULL`).

Quiz 7

Domanda

7. Da dove legge i dati la funzione standard `sscanf`?

- A) Dalla tastiera, bloccando il terminale fino alla pressione di Invio.
- B) Da uno stream di file precedentemente aperto con `fopen`.
- C) Dallo standard error.
- D) Da una stringa di caratteri (buffer) già presente in memoria RAM.

Soluzione
Risposta corretta: D
Spiegazione: La prima "s" in `sscanf` sta per "string". Essa analizza il contenuto di una stringa in memoria RAM formattandolo secondo gli specificatori di formato forniti, esattamente come `scanf` fa con lo standard input.

Quiz 8

Domanda

8. Se si apre un file in scrittura con `fopen("dati.txt", "w")` e il file esiste già sul disco, qual è il comportamento del sistema?

- A) Il file viene troncato, ovvero svuotato completamente diventando di dimensione zero.
- B) La funzione fallisce restituendo `NULL`.
- C) I nuovi dati vengono aggiunti in coda a quelli già esistenti.
- D) Viene sollevato un errore di runtime non intercettabile.

Soluzione
Risposta corretta: A
Spiegazione: La modalità "w" (write) cancella l'eventuale contenuto preesistente del file. Per scrivere in coda preservando i dati presenti occorre utilizzare la modalità "a" (append).

Quiz 9

Domanda

9. Perché è strettamente necessario invocare `fclose(fp)` al termine delle operazioni di scrittura su file?

- A) Per rilasciare i puntatori ed evitare integer overflow nello stack.
- B) Perché la scrittura avviene inizialmente in un buffer in memoria RAM gestito dall'OS; `fclose` forza il riversamento (*flush*) fisico dei dati sul disco rigido.

- C) Per consentire a `fscanf` di riposizionare il cursore all'inizio del file.
- D) Per forzare la compilazione del file stream.

Soluzione
Risposta corretta: B
Spiegazione: Per ragioni di prestazioni, le scritture su file sono "bufferizzate". I dati non vengono inviati al disco per ogni singolo carattere. La chiusura del file chiude il descrittore ed esegue il flush del buffer, garantendo l'integrità fisica dei dati memorizzati.

Quiz 10

Domanda

10. Nel dividere una stringa con `strtok("a,b,c", ",")`, in cosa vengono convertite le virgole all'interno dell'array originario?

- A) In caratteri di spazio.
- B) Nel carattere terminatore di stringa `'\0'`.
- C) Rimangono inalterate, la funzione lavora su una copia nascosta.
- D) Vengono eliminate compattando la stringa in memoria.

Soluzione
Risposta corretta: B
Spiegazione: `strtok` opera *in-place* e sostituisce fisicamente ciascun delimitatore individuato con il carattere sentinella `'\0'`. Questo spezza l'array in più sotto-stringhe disgiunte ma allocate nella stessa area di memoria RAM.

1.5 Esercizi Pratici

Esercizio 1: Calcolatrice da Riga di Comando

Domanda

(cli\{}_calc.c)

Si scriva un programma in C che accetti esattamente tre argomenti da riga di comando: due numeri interi e un operatore matematico (+, -, x). Il programma deve stampare a schermo il risultato dell'operazione. Se il numero di argomenti è errato o l'operatore non è supportato, stampare un opportuno messaggio d'errore su `stderr` e terminare con exit code 1.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      if (argc != 4) {
7          fprintf(stderr, "Usage: %s [n1] [op] [n2]\n", argv[0]);
8          return 1;
9      }
10     int n1 = atoi(argv[1]), n2 = atoi(argv[3]);
11     if (strcmp(argv[2], "+") == 0) printf("%d\n", n1 + n2);
12     else if (strcmp(argv[2], "-") == 0) printf("%d\n", n1 - n2);
13     else if (strcmp(argv[2], "x") == 0) printf("%d\n", n1 * n2);
14     else { fprintf(stderr, "Operato\n"); return 1; }
15     return 0;
16 }

```

Soluzione

Esercizio 2: Funzione di Lettura Robusta Interi

Domanda

(robust_input.c)

Si scriva una funzione C `int read_int(const char *prompt, int *val)` che implementi l'acquisizione robusta di un numero intero tramite `fgets` e `sscanf`.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int read_int(const char *prompt, int *val) {
5      char buf[128];
6      while (true) {
7          printf("%s", prompt);
8          if (!fgets(buf, stdin)) return 0;
9          if (sscanf(buf, "%d", val) == 1) return 1;
10         fprintf(stderr, "Input non valido\n");
11     }
12 }

```

Soluzione

Esercizio 3: Parser In-Memory di Configurazione

Domanda

(config_parser.c)

Si realizzi un programma che legga coppie chiave=valore saltando commenti (`\{\}`).

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char line[256];
7      while (fgets(line, sizeof(line), stdin)) {
8          if (line[0] == '#' || line[0] == '\n')
9              continue;
10         char *k = strtok(line, "\"");
11         char *v = strtok(NULL, "\"");
12         if (k && v) printf("Key: %s, Val: %s\n", k, v);
13     }
14     return 0;
15 }

```

Soluzione**Esercizio 4: Filtro CSV****Domanda**

(csv\{}_filter.c)

Si scriva un filtro CSV che stampa solo i cognomi con voto $\geq$ soglia CLI.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      long soglia = atol(argv[1]);
7      char line[256];
8      FILE *fp = fopen("studenti.csv", "r");
9      while (fgets(line, sizeof(line), fp)) {
10         char *nome = strtok(line, ",");
11         char *voto_s = strtok(NULL, ",");
12         if (atol(voto_s) >= soglia) printf("%s\n", nome);
13     }
14     fclose(fp);
15     return 0;
16 }

```

Soluzione**Esercizio 5: Inversione degli argomenti CLI****Domanda**

(cli\{}_reverse.c)

Stampa gli argomenti CLI al contrario, in maiuscolo.

```

1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      for(int i = argc - 1; i > 0; i--) {
7          for(int j = 0; argv[i][j]; j++) putchar(toupper(argv[i][j]))
8      }
9      putchar(',');
10     return 0;
11 }

```

Soluzione**Esercizio 6: Word & Character Counter****Domanda**

(my\{}_wc.c)

Conta caratteri, parole e righe da stdin.

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4  int main() {
5      int c, chars = 0, words = 0, lines = 0, in_w = 0;
6      while((c = getchar()) != EOF) {
7          chars++;
8          if(c == '\n') lines++;
9          if(isspace(c)) in_w = 0;
10         else if(in_w) { in_w = 1; words++; }
11     }
12     printf("Righe: %d, Parole: %d, Caratteri: %d\n", lines, words,
13         chars);
14     return 0;
15 }

```

Soluzione**Esercizio 7: Copia File con Error Logging****Domanda**

(file\{}_copy.c)

Si scriva un programma C che accetti due nomi di file come parametri (sorgente e destinazione). Copia il contenuto del primo nel secondo. Gestire tutti i casi di errore (argomenti mancanti, file non apribili) stampando messaggi significativi su stderr e terminando con exit code 1.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 3) {
6          fprintf(stderr, "Usa: %s [sorgente] [destinazione]\n", argv
          ↪ [0]);
7          return 1;
8      }
9
10     FILE *in = fopen(argv[1], "r");
11     if (in == NULL) {
12         fprintf(stderr, "Errore: impossibile aprire il file
          ↪ sorgente '%s'\n", argv[1]);
13         return 1;
14     }
15     FILE *out = fopen(argv[2], "w");
16     if (out == NULL) {
17         fprintf(stderr, "Errore: impossibile creare il file di
          ↪ destinazione '%s'\n", argv[2]);
18         fclose(in);
19         return 1;
20     }
21     fclose(in);
22
23     int c;
24     while ((c = fgetc(in)) != EOF) {
25         fputc(c, out);
26     }
27     fclose(in);
28     fclose(out);
29     return 0;
30 }
31

```

Soluzione**Esercizio 8: Validatore di Indirizzi IP****Domanda**

(ip\{}_parser.c)

Si scriva un programma C che prenda un indirizzo IP in formato A.B.C.D come parametro e lo stampi in formato esadecimale, validando che gli ottetti siano tra 0 e 255.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          fprintf(stderr, "Uso: %s [indirizzo_IP]\n", argv[0]);
7          return 1;
8      }
9
10     int o[4];
11     char residuo;
12     if (sscanf(argv[1], "%d.%d.%d.%d%c", &o[0], &o[1], &o[2], &o[3], &residuo) != 4) {
13         fprintf(stderr, "Errore: formato IP non valido. Richiesto A
14         ↪ .B.C.D\n");
15         return 1;
16     }
17     for (int i = 0; i < 4; i++) {
18         if (o[i] < 0 || o[i] > 255) {
19             fprintf(stderr, "Errore: ottenuto %d fuori dal
20             ↪ range [0..255]\n", i, o[i]);
21             return 1;
22         }
23     }
24     printf("Hex: %02X.%02X.%02X.%02X\n", o[0], o[1], o[2], o[3]);
25     return 0;
26 }

```

Soluzione**Esercizio 9: Tokenizzazione di Frase****Domanda**

(word\{}_tokenizer.c)

Estrae parole da stringa inserita tramite standard input usando come delimitatori spazi e punteggiatura.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5      char s[256];
6      printf("Inserisci una frase: ");
7      if (!fgets(s, sizeof(s), stdin))
8          return 0;
9
10     }
11     char *t = strtok(s, " ,.?! \\n\\t");
12     int i = 1;
13     while (t) {
14         printf("Parola %d: %s\\n", i++, t);
15         t = strtok(NULL, " ,.?! \\n\\t");
16     }
17     return 0;
18 }

```

Soluzione**Esercizio 10: Unione Alternata di File****Domanda**

(file\{}_merge.c)

Prende come parametri due file sorgente ed un file destinazione. Copia alternativamente una riga del primo e del secondo file nel file destinazione.


```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 4) {
6          printf(stderr, "Uso: %s [file1] [file2] [destinazione]\n",
7              argv[0]);
8          return 1;
9      }
10     FILE *f1 = fopen(argv[1], "r");
11     if (f1 == NULL) {
12         printf(stderr, "Errore: impossibile aprire '%s'\n", argv
13             [1]);
14         return 1;
15     }
16     FILE *f2 = fopen(argv[2], "r");
17     if (f2 == NULL) {
18         printf(stderr, "Errore: impossibile aprire '%s'\n", argv
19             [2]);
20         fclose(f1);
21         return 1;
22     }
23     FILE *o = fopen(argv[3], "w");
24     if (o == NULL) {
25         printf(stderr, "Errore: impossibile creare '%s'\n", argv
26             [3]);
27         fclose(f1);
28         fclose(f2);
29         return 1;
30     }
31     char l1[256], l2[256];
32     char *r1, *r2;
33     do {
34         r1 = fgets(l1, sizeof(l1), f1);
35         if (r1) {
36             fputs(l1, o);
37         }
38         r2 = fgets(l2, sizeof(l2), f2);
39         if (r2) {
40             fputs(l2, o);
41         }
42     } while (r1 || r2);
43     fclose(f1);
44     fclose(f2);
45     fclose(o);
46     return 0;
47 }

```