



Programmazione 1 (Canale O-Z)

Lezione 12: Basso Livello, File Binari e l'Abisso della Ricorsione

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

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

Testo vs Binario: L'Illusione ASCII

- **Testo (`fprintf`):** `1000000` diventa 7 caratteri ASCII ➡ **7 byte**. Un miliardo ➡ **10 byte**.
- In RAM un `int` occupa sempre **4 byte**, qualunque sia il valore.
- La CPU spreca cicli per convertire i bit in cifre (potenze di dieci).
- **File binario:** copia esatta byte per byte (*dump*) della RAM, senza traduzione.



? Se scrivo un milione su un file di testo con `fprintf`, quanto spazio occupa sul disco?

La Scrittura Binaria: `fwrite`

Si apre il file aggiungendo `b` alla modalità: `"wb"` (scrittura), `"rb"` (lettura).

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- `ptr`: indirizzo di memoria di partenza da cui leggere i dati.
- `size`: dimensione in byte di un singolo elemento (es. `sizeof(int)`).
- `nmemb`: numero di elementi consecutivi da scrivere.
- `stream`: puntatore allo stream del file (`FILE *`).



Testo vs Binario: il Codice

```
#include <stdio.h>

int main(void) {
    int numero = 1000000; // 1 milione

    // 1. Scrittura Testuale (fprintf)
    FILE *fp_txt = fopen("numero.txt", "w");
    if (fp_txt != NULL) {
        fprintf(fp_txt, "%d", numero);
        fclose(fp_txt);
    }

    // 2. Scrittura Binaria (fwrite)
    FILE *fp_bin = fopen("numero.bin", "wb");
    if (fp_bin != NULL) {
        fwrite(&numero, sizeof(int), 1, fp_bin);
        fclose(fp_bin);
    }

    printf("File generati correttamente.\n");
    return 0;
}
```



Passo 1: Compiliamo, Eseguiamo, Misuriamo

```
gcc -Wall numeri.c -o numeri  
./numeri
```

File generati correttamente.

- `numero.txt` ➡ **7 byte** (le cifre `'1'`, `'0'`, ..., `'0'`).
- `numero.bin` ➡ **4 byte** (la rappresentazione binaria dell'`int`).



Live: `ls -l numero.txt numero.bin` e confrontare la colonna delle dimensioni. Poi aprire `numero.bin` con un editor di testo: compaiono caratteri strani (come `@B`).



Passo 2: Hexdump e Little Endian

L'editor interpreta i byte come caratteri ASCII. Guardiamo i byte veri:

```
hexdump -C numero.bin
```

```
00000000  40 42 0f 00
```

```
|@B..|
```

- 1.000.000 in esadecimale è `0x000F4240`, ma nel file leggiamo `40 42 0F 00`.
- **Little Endian** (x86-64 Intel/AMD): i byte di una parola sono in RAM dal **meno significativo** al più significativo.
- `fwrite` copia fedelmente la sequenza di byte presente in RAM.

? Perché i byte appaiono "al contrario"?



Scrittura e Lettura di Array in Blocco

Gli elementi di un array sono **contigui** in RAM: basta una sola `fwrite`, niente cicli.

```
int dati[5] = {10, 20, 30, 40, 50};  
// Copia diretta di (5 * sizeof(int)) = 20 byte consecutivi  
fwrite(dati, sizeof(int), 5, fp_bin);
```

Specularmente, con il file aperto in `"rb"`:

```
int letti[5];  
fread(letti, sizeof(int), 5, fp_bin);
```



Navigazione nel File: `fseek` e `ftell`

- **Lettura sequenziale:** per il centesimo record di un file di testo leggiamo a vuoto le prime 99 righe.
- **File binari:** la dimensione dei dati è costante e nota ➡ saltiamo **direttamente** al punto voluto.

```
int fseek(FILE *stream, long offset, int origin);
```

- `SEEK_SET`: relativo all'**inizio** del file (offset positivo).
- `SEEK_CUR`: relativo alla **posizione corrente** del cursore.
- `SEEK_END`: relativo alla **fine** del file (offset generalmente negativo).
- `long posizione = ftell(fp);` ➡ byte in cui si trova il cursore.



Lettura Diretta di un Elemento

```
FILE *fp = fopen("array.bin", "rb");
if (fp != NULL) {
    // Saltiamo i primi due interi (2 * 4 = 8 byte) a partire dall'inizio del file
    fseek(fp, 2 * sizeof(int), SEEK_SET);

    int terzo_numero;
    fread(&terzo_numero, sizeof(int), 1, fp);
    printf("Terzo elemento dell'array letto direttamente: %d\n", terzo_numero);
    fclose(fp);
}
```

Terzo elemento dell'array letto direttamente: 30

 **Live:** scrivere prima `array.bin` con `{10, 20, 30, 40, 50}`, poi stampare `ftell(fp)` dopo la `fread` (vale 12). Compilare con `gcc -Wall`.

Record Variabili: Addio `fseek`

Record di testo a lunghezza variabile, preceduti dalla loro lunghezza:

Record 1		Record 2		Record 3	
4	"Etna"	7	"Catania"	4	"Roma"

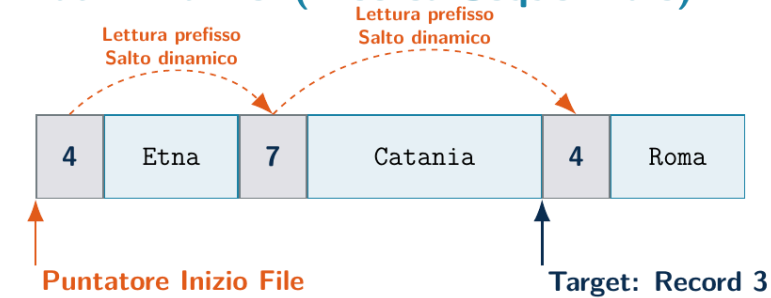
- Per raggiungere il terzo record dobbiamo **leggere i prefissi** dei primi due: leggo 4, salto 4; leggo 7, salto 7.
- ➡ Scansione sequenziale: il vantaggio di `fseek` è perso.

? Posso usare `fseek` per saltare direttamente a "Roma"? Come dovremmo salvare i record?

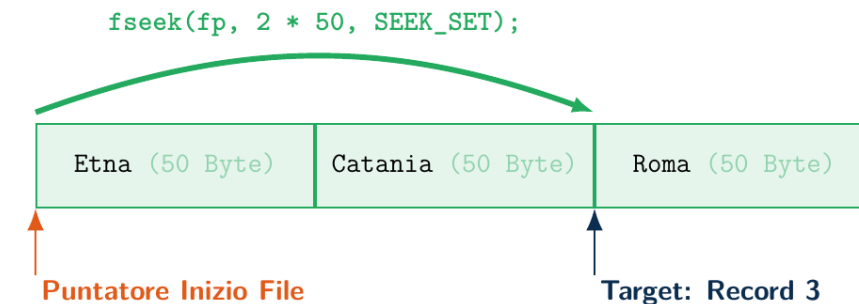
Record a Dimensione Fissa

- Ogni record occupa **sempre** la stessa dimensione (es. `char nome[50]`).
- La posizione del record i si calcola:
$$\text{Offset} = i \times \text{Dimensione Record}$$
- Es. terzo nome ($i = 2$): offset
 $2 \times 50 = 100$ byte.
- Con dati eterogenei (ID, media, nome) in un unico blocco? Lo vedremo con le **struct**.

Dati Dinamici (Ricerca Sequenziale)



Dati a Blocco Fisso (Accesso Diretto con fseek)



Il Concetto di Ricorsione

Una funzione risolve un problema **richiamando se stessa** su sotto-problemi più piccoli dello stesso tipo.

- **Le due regole d'oro:**
 - i. **Caso Base** (condizione di arresto): la situazione più semplice, con soluzione immediata. È l'ancora di salvataggio.
 - ii. **Passo Ricorsivo**: la chiamata a se stessa, con parametri che **convergono** verso il caso base.

? Finora abbiamo ripetuto azioni con `for` e `while`. Cosa succede se dimentichiamo il caso base?



Il Fattoriale ($N!$)

- **Caso Base:** $0! = 1! = 1$.
- **Passo Ricorsivo:** $N! = N \times (N - 1)!$

```
long long fattoriale(int n) {  
    // Caso Base  
    if (n == 0 || n == 1) {  
        return 1;  
    }  
    // Passo Ricorsivo  
    return n * fattoriale(n - 1);  
}
```



fattoriale(4) : la Discesa

Chiamata	Caso base?	Resta in sospeso
fattoriale(4)	no	4 * fattoriale(3)
fattoriale(3)	no	3 * fattoriale(2)
fattoriale(2)	no	2 * fattoriale(1)
fattoriale(1)	sì	return 1

- Ogni chiamata crea un nuovo **Stack Frame**, che resta "appeso" finché non si raggiunge il caso base.



fattoriale(4) : la Risalita

Frame che si chiude	Calcolo	Restituisce
<code>fattoriale(1)</code>	caso base	1
<code>fattoriale(2)</code>	<code>2 * 1</code>	2
<code>fattoriale(3)</code>	<code>3 * 2</code>	6
<code>fattoriale(4)</code>	<code>4 * 6</code>	24

 **Live:** eseguire `fattoriale(4)` su Python Tutor (modalità C) e osservare i frame che si impilano e poi collassano.

Stampa Inversa di una Stringa

Sfruttando lo Stack, stampiamo una stringa al contrario **senza** manipolare indici:

```
void stampa_inverso(char *str) {  
    // Caso Base: Carattere terminatore raggiunto  
    if (*str == '\0') {  
        return;  
    }  
    // Passo Ricorsivo: Avvia la chiamata per i caratteri successivi  
    stampa_inverso(str + 1);  
  
    // Al ritorno dallo srotolamento delle funzioni, stampa il carattere corrente  
    putchar(*str);  
}
```

- Con `stampa_inverso("ciao")`: le chiamate vedono `"ciao"`, `"iao"`, `"ao"`, `"o"`, `"` (caso base).
- Al ritorno si stampa a cascata dall'ultima alla prima ➡ `oaic`.



La Sequenza di Fibonacci

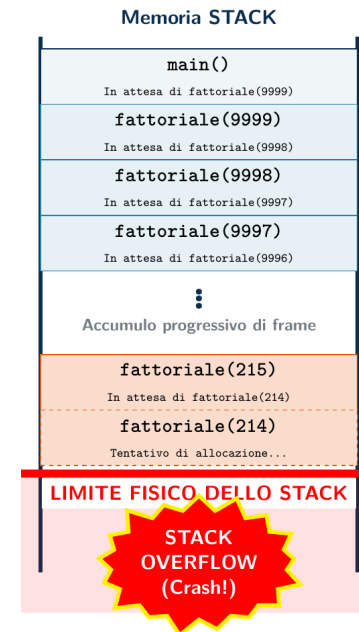
- **Casi Base:** se $N = 0$ ritorna 0; se $N = 1$ ritorna 1.
- **Passo Ricorsivo:** $\text{Fibonacci}(N - 1) + \text{Fibonacci}(N - 2)$.

```
int fibonacci(int n) {  
    // Casi Base  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    // Passo Ricorsivo (Doppia chiamata)  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

? Elegante... ma quanto costa? Prima vediamo cosa succede nello Stack a ogni chiamata.

● Lo Stack Frame e lo Stack Overflow

- Ogni chiamata riserva in cima allo Stack un **Record di Attivazione (Stack Frame)**: variabili locali, parametri, indirizzo di ritorno.
- Nella ricorsione i frame si **accumulano**, sospesi fino al caso base.
- **Stack Overflow**: senza caso base, o con ricorsione troppo profonda, lo Stack (circa **8 MB**) si esaurisce.
- Il sistema operativo termina il programma con *Segmentation Fault*.



🎬 **Live:** senza caso base nel fattoriale, `gcc -Wall fatt.c -o fatt && ./fatt` ➡ *Segmentation fault*.



Fibonacci: l'Inefficienza Esponenziale

- `fibonacci(45)` provoca un visibile **blocco** della CPU.
- La doppia chiamata crea un albero che cresce in modo **esponenziale** ($O(2^N)$).
- Lo stesso sotto-problema viene ricalcolato molte volte su rami diversi.
- Es. in `fibonacci(5)`: `fibonacci(3)` è calcolato **2 volte**, `fibonacci(2)` **3 volte**.

 **Live:** disegnare alla lavagna l'albero delle chiamate di `fibonacci(5)`; poi lanciare `fibonacci(45)` e attendere.

La Ricorsione di Coda (Tail Recursion)

- **Tail recursive:** la chiamata ricorsiva è l'**ultimissima** istruzione, nessuna computazione resta in sospeso.
- Con le ottimizzazioni (`gcc -O3`) si attiva la **Tail Call Optimization (TCO)**: lo stesso Stack Frame viene riutilizzato ➡ spazio sullo Stack $O(1)$.
- Con variabili **accumulatrici**, Fibonacci diventa $O(N)$ nel tempo:

```
// Funzione ausiliaria ottimizzata per la ricorsione di coda
long long fib_tail(int n, long long a, long long b) {
    if (n == 0) return a;
    if (n == 1) return b;
    return fib_tail(n - 1, b, a + b); // Nessuna operazione rimasta dopo la chiamata
}

long long fibonacci_efficiente(int n) {
    return fib_tail(n, 0, 1);
}
```

Ricorsione o Cicli?

- In C la ricorsione espone a rischi strutturali legati ai **limiti fisici dello Stack**.
- Per elaborazioni sequenziali lineari (Fattoriale, Fibonacci), i cicli `for` / `while` restano la scelta ottimale:
 - occupazione di memoria costante $O(1)$;
 - nessun rischio di Stack Overflow.

? Come scrivereste `fattoriale` con un semplice ciclo `for`?

Cosa Abbiamo Imparato Oggi

- **File binari:** `fwrite` / `fread` copiano i byte della RAM così come sono (`"wb"` , `"rb"`).
- `hexdump -C` mostra i byte reali: x86-64 è **Little Endian**.
- `fseek ( SEEK_SET , SEEK_CUR , SEEK_END )` e `ftell`: accesso diretto con **record a dimensione fissa**.
- **Ricorsione:** caso base + passo ricorsivo; ogni chiamata è uno **Stack Frame**.
- Rischi: **Stack Overflow** e costo esponenziale di Fibonacci; rimedi: **tail recursion** o cicli.

 **Per casa:** esercizi della Lezione 12 sul sito del corso.

Grazie per l'attenzione!

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