4 - Metodo di utilizzo dei file in C

I file svolgono un ruolo fondamentale nella programmazione, possono essere usati:

- per acquisire **input** in modo automatico senza l'utilizzo della tastiera
- per **memorizzare in modo persistente degli output** del programma
  Nel C i file vengono gestiti attraverso il concetto di **flusso o stream**; uno stream è una **sequenza di dati** (come stringhe, struct, ecc...), tipicamente i file si usano per memorizzare a lungo termine i dati contenuti nelle struct, ma possono essere usati anche per scopi differenti.

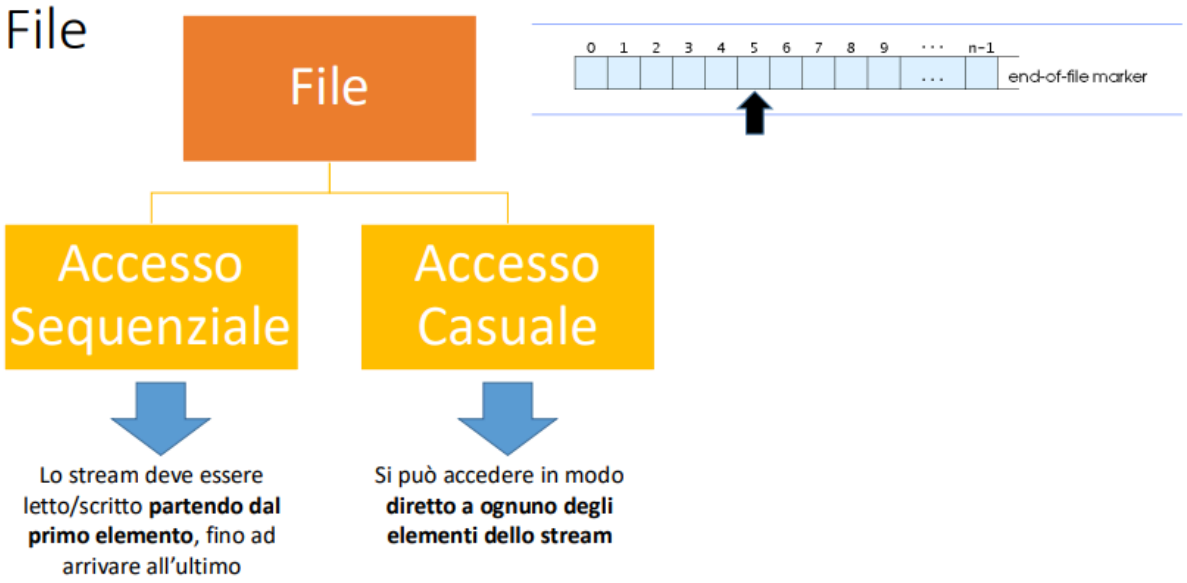
Il file

Lo stream è un concetto **astratto**, di fatto anche la navigazione sul web avviene tramite stream, ovvero flussi di dati che vengono scambiati tra un client ed un server.
Lo stream si divide in due possibilità:

- **input stream**: esso è il flusso di dati che può essere **letto**:
  - Apertura di uno stream
  - Lettura del dato
  - Avanzamento al dato successivo
  - Verifica di fine stream
  - Chiusura dello stream
- **output stream**: esso è il flusso di dati che può essere **scritto**:
  - Apertura di uno stream
  - Scrittura di un dato
  - Accodamento di un dato
  - Chiusura

Lo stream non è un concetto totalmente nuovo, lo si usa regolarmente con il `printf()` e lo `scanf()`, attraverso le corrispettive librerie:

- **`stdin`** per l'input da tastiera
- **`stdout`** per l'output sullo schermo
  La scrittura e lettura nei/sui file si basa sempre su delle varianti quasi omonime del printf e dello scanf.



I file ad **accesso sequenziale** sono anche detti **file testuali**, poiché il loro contenuto può essere aperto tramite un editor di testo e mostrato a schermo.
I file ad **accesso casuale** o diretto, sono anche detti **file binari**, poiché il loro contenuto non può essere visualizzato attraverso un editor di testo.

I file in C

Per poter utilizzare i file nel *C* si deve in primis includere la libreria `<stdio.h>` (che già includiamo a prescindere nella maggior parte dei progetti) questo perché si deve dichiarare una variabile di tipo **FILE**: `FILE *fileName`
Questa variabile è un puntatore che punta proprio alla struttura di tipo **FILE**, definita nella libreria `<stdio.h>`, per questo è d'obbligo includerla.
Questa struttura contiene:

- le **informazioni di sistema**, in modo tale da affacciarsi con il SO e visualizzare le varie informazioni dei file, come il recupero del **File Control Block** del file stesso.
- le **informazioni del file**, ovvero se è in scrittura o lettura, la data di modifica, la **locazione fisica** dei blocchi di memoria.

La struttura della struct FILE:

```
typedef struct {
    char *fpos; /* Current position of file pointer (absolute address) */
    void *base; /* Pointer to the base of the file */
    unsigned short handle; /* File handle */
    short flags; /* Flags (see FileFlags) */
    short unget; /* 1-byte buffer for ungetc (b15=1 if non-empty) */
    unsigned long alloc; /* Number of currently allocated bytes for the
file */
    unsigned short buffincrement; /* Number of bytes allocated at once */
} FILE;
```

Operazioni sui file

Le operazioni eseguibili sui file sono:

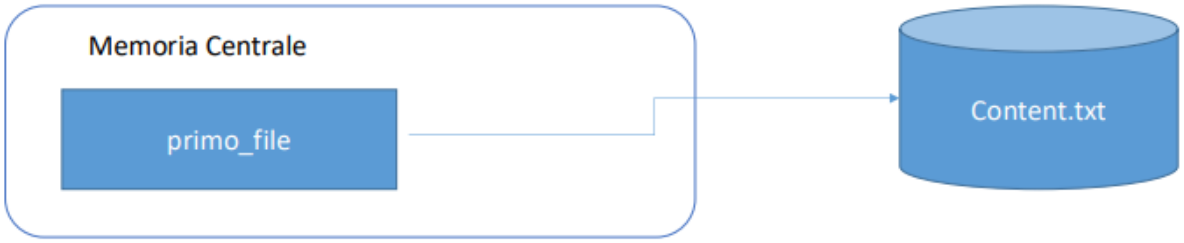
- Apertura del file
- Lettura dei dati
- Scrittura dei dati
- Chiusura del file
- Verifica di fine stream
- Riavvolgimento dello stream
- Collocare il puntatore in un punto preciso del file (nei file ad accesso casuale)

Apertura dei file

```
FILE *fopen(const char* filename, const char* mode);
```

Dove *filename* sta ad indicare il nome del file da aprire e *mode* è la modalità di apertura
Se il file esiste restituisce un puntatore, altrimenti restituirà **NULL**.

- **Esempio**
  - `FILE* primo_file = fopen("content.txt", "w");`



Apertura File - Modalità

	Modalità	Cosa fa
File **Testuali**	r	open a text file for **reading**
	w	truncate to zero length or **create a text file for writing**
	a	append; open or create text file for **writing at end-of-file**
	r+	**open text file for update (reading and writing)**
	w+	truncate to zero length or create a text file for update
	a+	append; open or create text file for update
File **Binari**	rb	open a binary file for **reading**
	wb	truncate to zero length or **create a binary file for writing**
	ab	append; open or create binary file for **writing at end-of-file**
	rb+	**open binary file for update (reading and writing)**
	wb+	truncate to zero length or create a binary file for update
	ab+	append; open or create binary file for update

22/04/2025 31

Chiusura dei file

`int fclose(const char* filename);`

Dove in questo caso il nome del file indica il file da chiudere. Ogni file aperto ha associato a se un **buffer**, ovvero un area di memoria temporanea, la fase di chiusura di un file permette di spostare il contenuto del buffer all'intero dello stream.
Inoltre la chiusura è fondamentale per dissociare il descrittore "FILE" dallo stream e rilasciare le risorse trattenute.

Esempio apertura e chiusura di un file:

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

int main(void) {
    puts("Utilizzo dei File");
    FILE *file;

    // Apertura del File. Stampa messaggio di conferma o di errore
    if( (file = fopen("test.txt", "r") ) == NULL) {
        puts("Errore nell'apertura");
    }
    else {
        puts("File aperto con successo");
    }

    // Chiusura del file
    if (file!=NULL && !fclose(file) ) {
        puts("File Chiuso");
    }
}
```

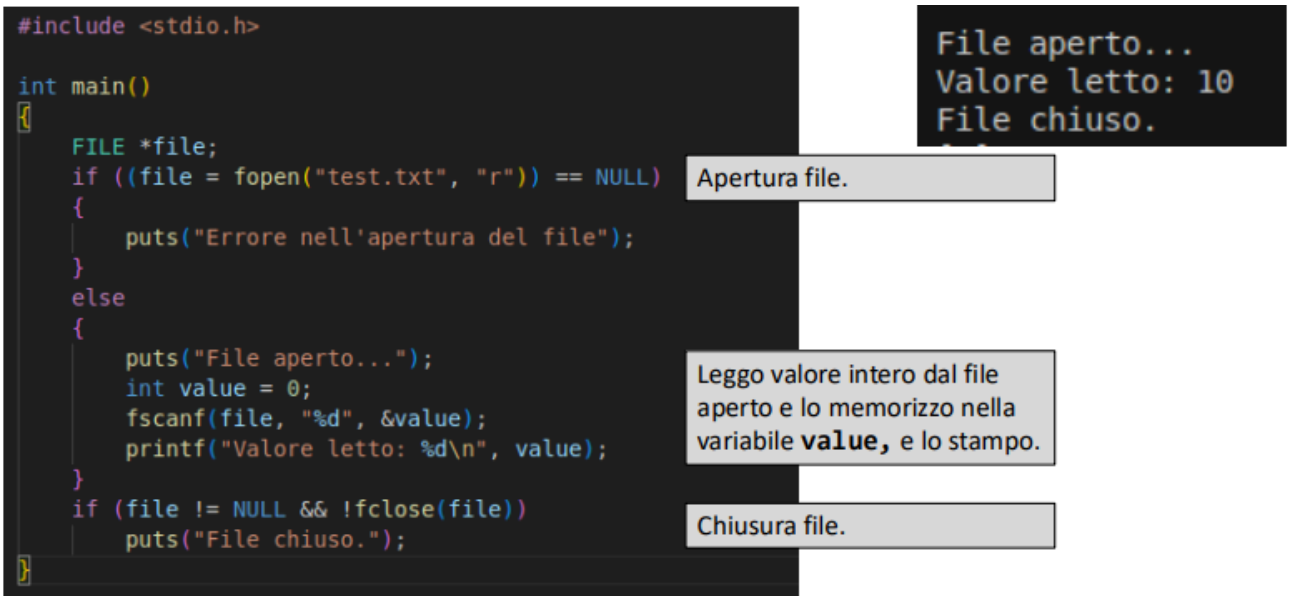
E' sicuramente fondamentale prestare attenzione alla root del file, in questo caso `test.txt` è un esempio relativo, il file dev'essere presente nella stessa root del progetto.
Inoltre fopen si preoccupa di creare il file nel momento in cui esso non esista, se quest'ultimo dovesse esistere stamperà un messaggio di errore, perché il suo puntatore sarà diverso da NULL.

Lettura dei file

`int fscanf(FILE* stream, const char* format, ...);`

Dove *stream* è il nome del file da cui si vogliono leggere i dati, mentre *format* è lo specificatore del formato dei dati.
Il suo metodo di utilizzo segue quello della *scanf* classica, solo con l'aggiunta del puntatore al file: ```

```c
int value=0; FILE *file;
fscanf(file, "%d", &value);
```

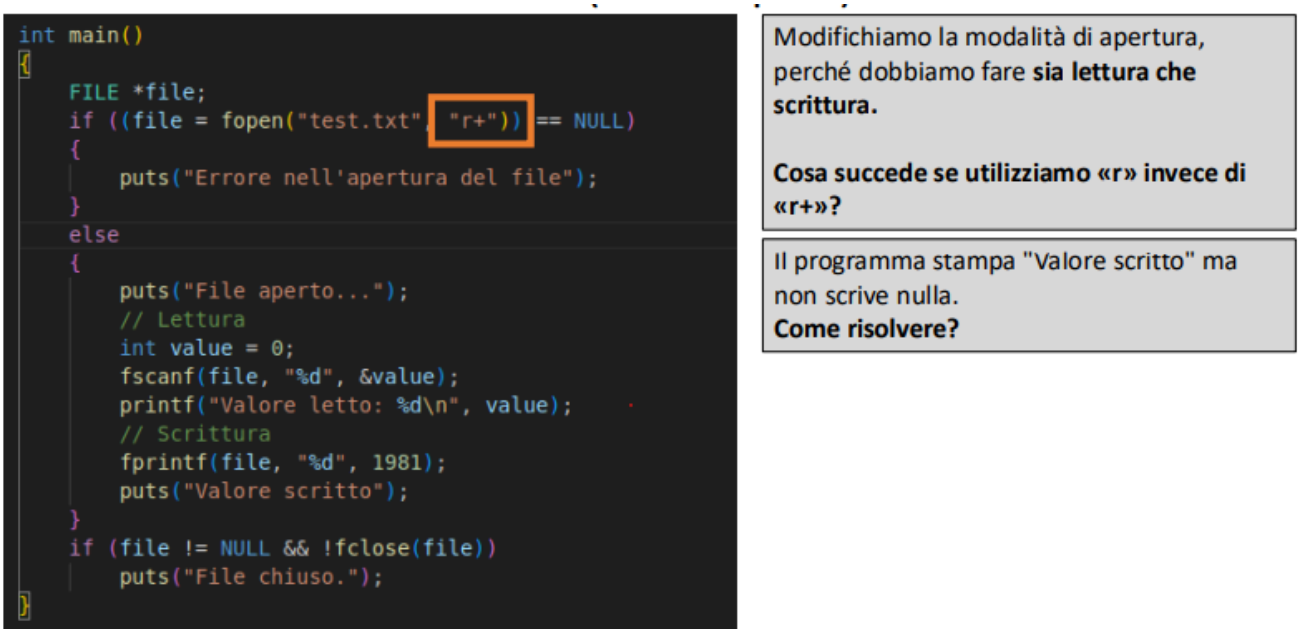


Scrittura dei file

`int fprintf(FILE* stream, const char* format, ...);`

Seguo lo stesso formato implementativo di *printf*, solo con l'aggiunta del puntatore al file:

```
int value=0; FILE *file;
fprintf(file, "%d", value);
```



Si può aggiungere un controllo sull'istruzione fprintf(), fprintf restituisce un intero pari al numero di caratteri scritti. Se non ha scritto caratteri o c'è stato un errore, il valore restituito è negativo. Modificando l'istruzione al rigo 16 in: if(fprintf(file, "%d", 123) > 0).
Aggiungiamo un controllo che aumenta la solidità del programma.

In questo programma la cifratura 1981 sarà posta alla fine di tutti i dati presenti già nel file, questo per ogni sua esecuzione, se eseguissimo due volte il codice leggeremo il primo valore, dato da *fscanf* (ipotizzando che sia 10), successivamente tutto attaccato vedremo 1981, dopo di che alla seconda esecuzione il valore finale letto sarà: 1019811981. Questo è

dato dall'**accesso sequenziale**,
i nuovi dati inseriti dipendono dalla posizione del puntatore.

Verifica di fine stream

`int feof(FILE* stream);`

Dove *stream* è il nome dello stream da verificare e il tipo restituito è **true** se il file è terminato.
Possiamo infatti utilizzare più stream nelle funzioni dei file, questo perché generalmente queste funzioni prendono in input un **generico stream**, ma possono essere ridirezionate verso gli standard di input/output, che come detto precedentemente anch'essi sono dei flussi. Quindi scrivere `fprintf(stdout, "%d",10)` è pari a scrivere `printf("%d", 10)`.
Questa funzione trova il suo utilizzo nei cicli che scorrono tra i contenuti di un file.

```
FILE file;
while(!feof(file)){
    //fai questo
}
```

Riavvolgimento dello stream

`void rewind(FILE* stream);`

Dove *stream* è il nome dello stream da riavvolgere, ovvero il puntatore del file viene riportato **all'inizio del file stesso**, in modo tale da ricominciare la scrittura/lettura, utile in casi di modifiche, ecc...

Seguendo la logica degli esempi precedenti, 10 sarà sostituito dalla successiva riga di scrittura sul file, ovvero 123, avendo infine come risultato **12323**, con il puntatore fermo al primo *3*, dopo di che il file sarà chiuso e quindi salvato.

File binari

I file binari sono detti anche file **ad accesso casuale**, poiché il puntatore non scorre sequenzialmente i contenuti, come succedeva nel caso dei file sequenziali; questo ci permette di risolvere il problema della sovrascrittura di contenuti, ad ogni esecuzione la riscrittura degli stessi elementi per modificare solo una piccola parte richiesta, garantendo così maggior **flessibilità**, i contenuti vengono modificati senza sovrascrivere ciò che era memorizzato precedentemente.

Pro e contro

I dati nei file binari sono memorizzati come **raw bytes**, ovvero non vengono convertiti in un tipo o in un testo ma memorizzati nella memoria direttamente in formato grezzo nel file.
Inoltre i dati dello stesso tipo utilizzano la stessa quantità di memoria, per esempio viene assegnata ad un **int** possibilità di occupare x byte, mentre ad un **char** una quantità $k < x$.
Questo aiuta perché **sapendo il tipo di dato** sappiamo **quanto spazio occupa** e dove trovarlo.
Questo principio non cede eccezione ai record, infatti anch'essi hanno una dimensione fissa per ogni tipo, ogni volta che si cerca un record di x byte sarà sicuramente quello e ciò rende la ricerca molto più semplice, senza controllare i separatori come nei file di testo.

Un lato **negativo** dei file binari è che non sono **human readable**, questo perché sono salvati come byte **puri** e aprendo il file non sarà possibile leggere un testo comprensibile.

Operazione di scrittura sui file binari

`size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream);`
Questa funzione scrive nello `stream` un numero di elementi pari a `nmemb`, ognuno dei quali ha dimensione pari a `size`, attualmente memorizzati in `ptr`.
Dove:

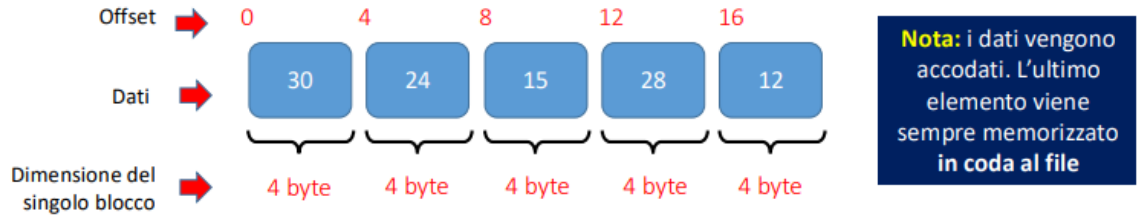
- **ptr** è il puntatore alla variabile da copiare nel blocco dati
- **size** è la dimensione del blocco dati da scrivere
- **nmemb** è il numero dei blocchi di memoria da scrivere
- **stream** è il puntatore al file
  Indichiamo con **size_t** un **nuovo tipo di dato** intero senza segno a 16bit. Utilizzato principalmente per memorizzare le **dimensioni delle variabili** o i contatori.

Rispetto ai file sequenziali abbiamo **due nuovi parametri formali**, ovvero `size` e `nmemb`, essi ci richiedono quindi di sapere quanto sono **grandi** i dati che vogliamo scrivere sui file.
Per capire quanto sono grandi i dati utilizziamo la funzione **`sizeof`**, la quale restituisce la dimensione di una variabile in byte. Nel nostro caso, per ogni variabile da scrivere, bisogna utilizzare `sizeof( )` sulla variabile per spiegare al compilatore di quanta memoria abbiamo bisogno.
Il numero dei blocchi che ci servono è invece pari **al numero di elementi** che vogliamo memorizzare (1 se è una variabile singola, n se è un vettore di dimensione n).

Esempio:
Supponiamo che vogliamo inserire in un file binario un voto di un esame, la chiamata della funzione `fwrite()` diventa:

```
FILE* file;
int vote=30;
fwrite(&vote, sizeof(vote),1,file);
```

- In questo modo ad ogni elemento che **memorizziamo viene associata una dimensione predefinita**. Supponendo di memorizzare una sequenza di cinque voti, la situazione in memoria sarà la seguente



Lettura dei file binari

Il funzionamento della lettura dei file binari è analoga al funzionamento di `fwrite`, il suo prototipo è: `size_t fread(const void* ptr, size_t size, size_t nmemb, FILE* stream);`
Supponendo di voler leggere una variabile intera da file la sua chiamata diventa:

```
int value=0;
FILE* file;
fread(&value, sizeof(value),1,file);
```

Basandoci sull'esempio precedente il risultato ottenuto sarà il valore **30**. Se volessimo leggere un altro dato, grazie alla definizione di file binario ciò è possibile dato che possiamo posizionare il puntatore ad un elemento particolare, sapendo già tutta la memoria allocata per ogni singolo dato, ciò è possibile tramite la funzione di **posizionamento del puntatore**.

Posizionare il pointer

`int fseek(FILE* stream, log int offset, int whence);`
Dove:

- **stream** è il puntatore al file su cui scrivere i dati
- **offset** è lo spostamento all'interno del file
- **whence** è la posizione iniziale del puntatore che viene determinata in base alle tre **costanti** che può assumere whence:
  - **SEEK_SET**: dall'inizio del file
  - **SEEK_END**: dalla fine del file
  - **SEEK_CUR**: dalla posizione corrente

la funzione `fseek` sposta il puntatore `steam` di **offset** byte a partire dalla posizione iniziale di **whence**.
Se volessi quindi raggiungere la posizione k devo puntare al **k-esimo** elemento del file: `offset=k*(sizeof(var))`, dove *var* indica la dimensione delle variabili allocate.

Esempio di seek:

Main:

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

#define NUMERI 10 //scrivere 10 numeri randomt ra 0 e 99 in un file

void scrivi(char *nomeFile, int n);
int leggiInt(char *nomeFile, int pos); //leggi un numero in una specifica
posizione

int main(){
    scrivi("int.dat",NUMERI);
    int v=leggiInt("int.dat",3);
    printf("LETTO: %d\n", v);
}
```

Scrivi:

```
void scrivi(char *nomeFile, int n){
    int seed=time(NULL);
    srand(seed);
    FILE *file=fopen(nomeFile,"wb");
    if(file!=NULL){
        for(int i=0;i<n;i++){
            int r=rand()%100;
            if(fwrite(&r,sizeof(int),1,file)>0) printf("SCRITTO: %d\n",
r);
        }
        fclose(file);
    }
    else printf("ERROR\n");
}
```

LeggiInt:

```
int leggiInt(char *nomeFile, int pos){
    FILE *file=fopen(nomeFile, "rb");
    if(file!=NULL){
```

```
        fseek(file,pos*sizeof(int),SEEK_SET); //3*dimensione dell'int
        int v;
        fread(&v,sizeof(int),1,file);
        fclose(file);
        return v;
    }
    else printf("ERROR\n");
}
```

Appendice: Elaborazione di file

Lettura di singoli caratteri

- **Funzioni**:
  - `int getc(FILE* stream);`
  - `int fgetc(FILE* stream);`
  - `int getchar(void);`
- **Descrizione**:
  - `getc` e `fgetc` leggono il successivo carattere da uno stream e avanzano la posizione di un byte.
  - Restituiscono il carattere letto (come `int`) oppure `EOF` in caso di fine file o errore.
  - `getchar` è equivalente a `getc(stdin)` (lettura da tastiera).

Scrittura di singoli caratteri

- **Funzioni**:
  - `int fputc(int c, FILE* stream);`
  - `int putc(int c, FILE* stream);`
  - `int putchar(int c);`
- **Descrizione**:
  - `fputc` e `putc` scrivono un carattere nello stream specificato.
  - Restituiscono il carattere scritto oppure `EOF` in caso di errore.
  - `putchar` è equivalente a `putc(stdout)` (scrittura su schermo).

Lettura e scrittura di stringhe

- **Funzioni di lettura**:
  - `char* fgets(char* s, int n, FILE* stream);`
  - `char* gets(char* s);`
- **Funzioni di scrittura**:
  - `int fputs(const char* s, FILE* stream);`
  - `int puts(const char* s);`
- **Descrizione**:

- `fgets` legge fino a `n-1` caratteri da uno stream, aggiunge `'\0'` alla fine. Include anche il newline (`\n`) se presente.
- `gets` legge da `stdin` fino al newline (poi sostituito da `'\0'`). **Non sicura**, può causare overflow.
- `fputs` scrive una stringa su uno stream; ritorna un valore positivo o `EOF` in caso di errore.
- `puts` scrive su `stdout` e aggiunge automaticamente un newline.