

GESTIONE DEI FILE

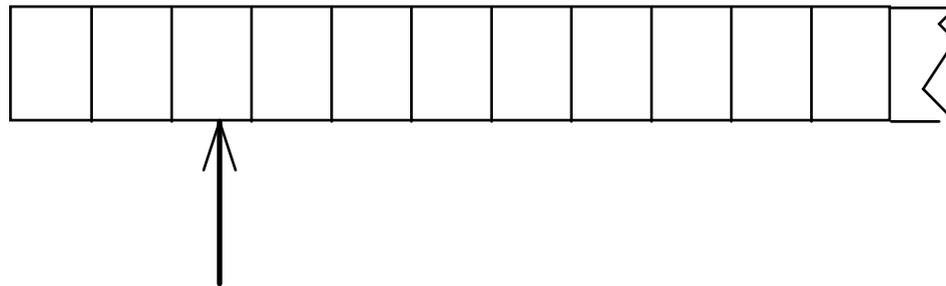
Per poter mantenere disponibili i dati tra le diverse esecuzioni di un programma (*persistenza* dei dati) è necessario poterli *archiviare su memoria di massa*

- Un **file** è una ***astrazione fornita dal sistema operativo***, per consentire la memorizzazione di informazioni su memoria di massa
- Un file è un'astrazione di memorizzazione di ***dimensione potenzialmente illimitata ad accesso sequenziale***

IL CONCETTO DI FILE

Una testina di lettura/scrittura (ideale) indica in ogni istante il record corrente:

- inizialmente, la testina si trova per ipotesi sulla ***prima posizione***
- dopo ***ogni operazione di lettura/scrittura***, essa si ***sposta*** sulla registrazione successiva



È illecito operare oltre la fine del file

OPERARE SUI FILE

A livello di sistema operativo un file è denotato univocamente dal suo **nome assoluto**, che comprende il **percorso** e il **nome relativo**

In certi sistemi operativi il percorso può comprendere anche il **nome dell'unità**

- *in DOS o Windows:*

C: \temp\prova1.c

- *in UNIX e Linux:*

/usr/temp/prova1.c

APERTURA E CHIUSURA DI UN FILE

Poiché un file è un'entità del sistema operativo, **per agire su esso dall'interno di un programma occorre *stabilire una corrispondenza* fra:**

- **il nome del file** come risulta al sistema operativo
- **un nome di variabile** definita nel programma

Dopo l'apertura, il programma **opera sul file *utilizzando la variabile* che lo rappresenta**: il sistema operativo provvederà a effettuare l'operazione richiesta sul file associato a tale simbolo

Al **termine**, la corrispondenza dovrà essere ***eliminata***:
operazione di ***chiusura del file***

FILE IN C

- Il tipo **FILE** è una **struttura definita in header standard `<stdio.h>`**, che l'utente non ha necessità di conoscere nei dettagli (che spesso cambia da una piattaforma all'altra)
- Le strutture **FILE** non sono ***mai gestite direttamente dall'utente***, ma solo dalle funzioni della libreria standard **`stdio`**
- **L'utente** definisce e usa, nei suoi programmi, solo dei ***puntatori a FILE***

IL MODELLO DI FILE IN C

- **Libreria standard stdio**
- l'input avviene da un **canale di input associato a un file aperto in lettura**
- l'output avviene su un **canale di output associato a un file aperto in scrittura**
- **Due tipi di file: file binari e file di testo**
 - **basterebbero i file binari**, ma sarebbe scomodo fare tutto con solo questi
 - i file di testo, *pur non indispensabili*, rispondono a un'esigenza pratica molto sentita

FILE IN C: APERTURA

Per aprire un file si usa la funzione:

```
FILE* fopen(char fname[], char modo[])
```

Questa funzione *apre il file di nome `fname` nel modo specificato*, e restituisce un puntatore a **FILE** (che punta a una nuova struttura **FILE** appositamente creata)

ATTENZIONE alle convenzioni dipendenti dal sistema operativo usato (\ oppure / nei percorsi, presenza o assenza di unità, ...)

FILE IN C: APERTURA

Per aprire un file si usa la funzione:

```
FILE* fopen(char fname[], char modo[])
```

modo specifica *come* aprire il file:

- **r** apertura in lettura (read)
- **w** apertura in scrittura (write)
- **a** apertura in aggiunta (append)

seguita opzionalmente da:

- **t** apertura in modalità testo (default)
- **b** apertura in modalità binaria

ed eventualmente da:

- **+** apertura con possibilità di *modifica*

FILE IN C: APERTURA

Per aprire un file si usa la funzione:

FILE* fopen(char fname[], char modo[])

Il valore restituito da `fopen()` è un *puntatore a FILE*, da usare in tutte le successive operazioni sul file

- NULL in caso l'apertura sia fallita
- controllarlo è il solo modo per sapere se il file si sia davvero aperto
- se non si è aperto, il programma usualmente *non deve proseguire* → chiamata a funzione di libreria `exit()`

I canali predefiniti standard (stdin, stdout, stderr) sono dei file già aperti: quindi, il loro tipo è **FILE***

FILE IN C: CHIUSURA

Per chiudere un file si usa la funzione:

```
int fclose(FILE*)
```

- Il **valore restituito da `fclose()`** è un **intero**
 - 0 se tutto è andato bene
 - EOF (valore intero negativo) in caso di errore
- Prima della chiusura, ***tutti i buffer vengono svuotati***

FINE DEL FILE

La fine del file può essere rilevata:

- in base ***all'esito delle operazioni di lettura***

È l'approccio standard del C: prima si tenta una operazione di lettura, poi si guarda se è andata a buon fine, controllando *il valore da essa restituito*

- oppure perché **si intercetta il carattere di EOF**, come vedremo, nel solo caso di file di testo

Attenzione: lo speciale carattere EOF (End-Of-File) varia da una piattaforma all'altra (spesso **EOF=-1**; ANSI C prescrive un qualunque intero negativo; definizione di **EOF** in `<stdio.h>`)

FILE DI TESTO

Un **file di testo** è un file che contiene ***sequenze di caratteri***

È un caso *estremamente frequente*, con ***caratteristiche proprie***:

- esiste un concetto di *riga* e di *fine riga* ('\n')
- certi caratteri sono *stampabili a video* (quelli di codice ≥ 32), altri no
- la sequenza di caratteri è terminata dal **carattere speciale EOF**

FILE DI TESTO E CANALI STANDARD

I canali di I/O standard *non sono altro che file di testo già aperti*

- **stdin** è un file di testo aperto in lettura, di norma agganciato alla tastiera
- **stdout** è un file di testo aperto in scrittura, di norma agganciato al video
- **stderr** è un altro file di testo aperto in scrittura, di norma agganciato al video

Le funzioni di I/O disponibili per i file di testo sono una *generalizzazione di quelle già note* per i canali di I/O standard

CONFRONTO

<i>Funzione da console</i>	<i>Funzione da file</i>
<code>int getchar(void);</code>	<code>int fgetc(FILE* f);</code>
<code>int putchar(int c);</code>	<code>int fputc(int c, FILE* f);</code>
<code>char* gets(char* s);</code>	<code>char* fgets(char* s, int n, FILE* f);</code>
<code>int puts(char* s);</code>	<code>int fputs(char* s, FILE* f);</code>
<code>int printf(...);</code>	<code>int fprintf(FILE* f, ...);</code>
<code>int scanf(...);</code>	<code>int fscanf(FILE* f, ...);</code>

`getchar()` e `putchar()` sono semplicemente delle scorciatoie linguistiche per `fgetc()` e `fputc()`

`getchar()` \equiv `fgetc(stdin)`

`putchar(c)` \equiv `fputc(c, stdout)`

ESEMPIO 1

Salvare su un file di testo `prova.txt` ciò che viene digitato sulla tastiera

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

int main(void) {
    FILE* fp = fopen("prova.txt", "w");
    if (fp==NULL) exit(1); /* Non si è aperto */
    else {
        int c;

        while ((c=getchar()) != EOF) fputc(c, fp);
        fclose(fp);
    }
}
```

`fp` è NULL se non c'è spazio su disco o protetto da scrittura

ESEMPIO 2

Stampare a video il contenuto di un file di testo `prova.txt`

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    FILE *fp;
    if ((fp = fopen("prova.txt", "r")) == NULL)
        exit(1); /* Errore di apertura */
    else {
        int c;
        while ((c=fgetc(fp)) != EOF) putchar(c);
        fclose(fp);
    }
}
```

`fp` può essere NULL se il file richiesto non esiste o non può essere aperto

ESEMPIO 3

È dato un file di testo `people.txt` le cui righe rappresentano ciascuna i dati di una persona, secondo il seguente formato:

- **cognome** (al più 30 caratteri)
- uno o più spazi
- **nome** (al più 30 caratteri)
- uno o più spazi
- **sesso** (un singolo carattere, 'M' o 'F')
- uno o più spazi
- **anno di nascita**

ESEMPIO 3

Si vuole scrivere un programma che

- legga riga per riga i dati dal file
 - e ponga i dati in un array di persone
 - ... *(poi svolgeremo elaborazioni su essi)*
-

Un possibile file `people.txt`:

```
Rossi Mario M 1947
Ferretti Paola F 1982
Verdi Marco M 1988
Bolognesi Annarita F 1976
...
```

ESEMPIO 3

1) Definire il tipo `persona`

Occorre definire una `struct` adatta a ospitare i dati elencati:

- **cognome** → array di 30+1 caratteri
- **nome** → array di 30+1 caratteri
- **Sesso** → array di 1+1 caratteri
- **anno di nascita** → un intero

non è la sola scelta possibile (ma è **comoda...**)

```
typedef struct {  
    char cognome[31], nome[31], sesso[2];  
    int anno;  
} persona;
```

ESEMPIO 3

Poi, nel main:

2) definire un array di **persona**

3) aprire il file in lettura

```
int main(void) {  
    persona v[DIM];  
    FILE* f = fopen("people.txt", "r");  
    if (f==NULL) {  
        . . .  
    }  
    . . .  
}
```

apertura in lettura

ESEMPIO 3

Poi, nel main:

2) definire un array di **persona**

3) aprire il file in lettura

```
int main(void) {  
    persona v[DIM];  
    FILE* f = fopen("file.txt", "r");  
    if (f==NULL) {  
        perror("Il file non esiste!");  
        exit(1);  
    }  
    . . .  
}
```

perror(msg) stampa un messaggio d'errore sul canale standard stderr

exit(n) fa terminare il programma, restituendo al SO il valore n come codice di errore

ESEMPIO 3

Poi, nel main:

4) leggere una riga per volta, e porre i dati di quella persona in una cella dell'array

Come organizzare la lettura?

- Dobbiamo leggere delle stringhe separate una dall'altra da spazi
- Sappiamo che ogni singola stringa (cognome, nome, sesso) non contiene spazi

Scelta più pratica: fscanf ()

ESEMPIO 3

Che cosa far leggere a `fscanf()` ?

- Tre stringhe separate una dall'altra da spazi
→ si ripete *tre volte* il formato `%s`
- Un intero → si usa il formato `%d`
- Il fine riga → occorre specificare in fondo `\n`

```
fscanf(f, "%s%s%s%d\n", ...)
```

Fino a quando si deve leggere?

- Quando il file termina, `fscanf()` restituisce `EOF`
→ basta controllare il valore restituito
- Si continua fintanto che è diverso da `EOF`

```
while (fscanf(...) != EOF)
```

```
...
```

ESEMPIO 3

Dove mettere quello che si legge?

- Abbiamo definito un array di **persona**, **v**
- Struttura fatta di *cognome*, *nome*, *Sesso*, *anno* → ciò che si estrae da una riga va nell'ordine in **v[k].cognome**, **v[k].nome**, **v[k]. Sesso**, **v[k].anno**

E dopo aver letto una riga?

- La testina di lettura sul file è già andata a capo, perché il formato di lettura prevedeva esplicitamente di *consumare il fine linea* (**\n**)
- L'indice **k** invece indica ancora la cella appena occupata → occorre incrementarlo, affinché indichi la prossima cella libera

ESEMPIO 3

4) leggere una riga per volta, e porre i dati di quella persona in una cella dell'array

```
int main(void) {  
    int k=0; /* indice per array */  
    ...  
    while (fscanf(f, "%s%s%s%d\n",  
        v[k].cognome, v[k].nome,  
        v[k].sesso, &(v[k].anno) ) != EOF) {  
        k++; /* devo incrementare k */  
    }  
}
```

Ricorda: l'intero richiede l'estrazione esplicita dell'indirizzo della variabile

ESEMPIO 3

4) leggere una riga per volta, e porre i dati di quella persona in una cella dell'array

Ricordare:

- **fscanf ()** elimina automaticamente gli spazi che separano una stringa dall'altra → non si devono inserire spazi nella stringa di formato
- **fscanf ()** considera finita una stringa al primo spazio che trova → non si può usare questo metodo per leggere stringhe contenenti spazi

ESEMPIO 3: programma completo

```
#define DIM 30
#include <stdio.h>
#include <stdlib.h>
```

Dichiara la procedura `exit()`

```
typedef struct {
    char cognome[31], nome[31], sesso[2];
    int anno;
} persona;
```

```
int main(void) {
    persona v[DIM]; int k=0; FILE* f;
    if ((f=fopen("people.txt", "r"))==NULL) {
        perror("Il file non esiste!"); exit(1); }
    while (fscanf(f, "%s%s%s%d\n", v[k].cognome,
                 v[k].nome, v[k].sesso, &(v[k].anno)) != EOF)
        k++;
}
```

ESEMPIO 3: VARIANTE

E se usassimo *un singolo carattere*
per rappresentare il sesso?

```
typedef struct {  
    char cognome[31], nome[31], sesso;  
    int anno;} persona;
```

ESEMPIO 3: VARIANTE

Che cosa cambierebbe?

- `fscanf` elimina *automaticamente* gli spazi prima di leggere una stringa o un numero (intero o reale)... ***ma non prima di leggere un singolo carattere***, perché se lo facesse non riuscirebbe a leggere il carattere spazio
- **Ma noi non sappiamo quanti spazi ci sono fra nome e sesso**
- Quindi, non possiamo sapere a priori dov'è il carattere che ci interessa

ESEMPIO 3: VARIANTE

Infatti, il nostro file potrebbe essere fatto così:

```
Rossi Mario M 1947
Ferretti Paola F 1982
Verdi Marco M 1988
Bolognesi Annarita F 1976
...
```

The diagram shows a list of names and initials with callouts pointing to specific spaces. The callouts are:

- Callout 1: "Qui, uno spazio prima di M" (pointing to the space before 'M' in the first line)
- Callout 2: "Qui, due spazi prima di F" (pointing to the two spaces before 'F' in the second line)
- Callout 3: "Qui, tre spazi prima di M" (pointing to the three spaces before 'M' in the third line)
- Callout 4: "Qui, uno spazio prima di F" (pointing to the space before 'F' in the fourth line)

- prima, dicendo a `fscanf()` di leggere una stringa, gli spazi (uno, due, ...) erano eliminati comunque
- adesso, dicendo a `fscanf()` di leggere un carattere singolo, **dobbiamo decidere che cosa fare all'interno del programma**

ESEMPIO 3: VARIANTE

Due possibilità:

- **scelta 1**: *introdurre comunque una stringa di due caratteri* e usarla per far leggere il carattere relativo al sesso a `fscanf()`
Poi, copiare il primo carattere al suo posto
- **scelta 2**: costruirsi un ciclo che *salti tutti gli spazi* fino al primo carattere non-spazio, poi recuperare quest'ultimo
→ non consente più di usare `fscanf()` per gestire tutta la fase di lettura

VARIANTE 1

```
#define DIM 30
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct {
    char cognome[31], nome[31], sesso;
    int anno;
} persona;
```

Stringa ausiliaria

```
int main(void) {
    persona v[DIM]; int k=0; FILE* f; char s[2];
    if ((f=fopen("people.txt", "r"))==NULL) {
        perror("Il file non esiste!"); exit(1); }
    while(fscanf(f, "%s%s%s%d\n", v[k].cognome,
                v[k].nome, s, &v[k].anno) != EOF) {
        v[k].sesso = s[0]; k++; }
}
```

Copiatura carattere

VARIANTE 2

```
typedef struct {
    char cognome[31], nome[31], sesso;
    int anno;
} persona;

int main(void) {
    persona v[DIM]; int k=0; FILE* f; char ch;
    if ((f=fopen("people.txt", "r"))==NULL) {
        perror("Il file non esiste!"); exit(1); }
    while(fscanf(f, "%s%s", v[k].cognome,
                v[k].nome) != EOF) {
        while((ch=fgetc(f))==' ');
        v[k].sesso = ch;
        fscanf(f, "%d\n", &v[k].anno); k++; }
}
```

Carattere ausiliario

Salta spazi

VARIANTE 2

```
typedef struct {
    char cognome[31], nome[31], sesso;
    int anno;
} persona;

int main() {
    persona v[10];
    if ((f = fopen("dati.txt", "r")) != NULL) {
        perror("Errore di apertura del file");
        while (fscanf(f, "%s", v[k].cognome,
                    v[k].nome) != EOF) {
            do fscanf(f, "%c", &ch); while (ch == ' ');
            v[k].sesso = ch;
            fscanf(f, "%d", &v[k].anno);
            k++;
        }
    }
}
```

Alternativa: anziché `fgetc`, si può usare `fscanf` per leggere il singolo carattere
→ occorre un ciclo `do/while` (prima si legge, poi si verifica cosa si è letto)

Ricorda: il singolo carattere richiede l'estrazione esplicita dell'indirizzo

ESEMPIO 4

È dato un file di testo `elenco.txt` le cui righe rappresentano ciascuna i dati di una persona, secondo il seguente formato:

- **cognome** (esattamente 10 caratteri)
- **nome** (esattamente 10 caratteri)
- **sex** (esattamente un carattere)
- **anno di nascita**

I primi due possono contenere spazi al loro interno

NB: non sono previsti spazi espliciti di separazione

ESEMPIO 4

Che cosa cambia rispetto a prima?

- sappiamo esattamente dove iniziano e dove finiscono i singoli campi
- non possiamo sfruttare gli spazi per separare cognome e nome

Un possibile file `elenco.txt`:

```
Rossi      Mario      M1947
Ferretti   Paola      F1982
De Paoli   Gian MarcoM1988
Bolognesi Anna Rita  F1976
...
```

I vari campi possono essere "attaccati": tanto, sappiamo a priori dove inizia l'uno e finisce l'altro

ESEMPIO 4

Come fare le letture?

non possiamo usare `fscanf (f, "%s", ...)`

- si fermerebbe al primo spazio
- potrebbe leggere più caratteri del necessario (si pensi a Gian MarcoM1988)

però possiamo usare `fscanf ()` nell'altra modalità, specificando quanti caratteri leggere. Ad esempio, per leggerne dieci:

`fscanf (f, "%10c", ...)`

Così legge esattamente 10 caratteri, spazi inclusi

ESEMPIO 4

Come fare le letture?

- non possiamo usare `fscanf (f, "%s", ...)`

- **ATTENZIONE:** viene riempito un array di caratteri, senza inserire alcun terminatore

- per Occorre aggiungerlo a parte modalità, specificando quanti caratteri leggere. Ad esempio, per leggerne dieci:

`fscanf (f, "%10c", ...)`

Così legge esattamente 10 caratteri, spazi inclusi

ESEMPIO 4: PROGRAMMA COMPLETO

```
#define DIM 30
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char cognome[11], nome[11], sesso; int anno;
} persona;

int main(void) {
    persona v[DIM]; int k=0; FILE* f;
    if ((f=fopen("elenco.txt", "r")) == NULL) {
        perror("Il file non esiste!");
        return 1;
    }
    while (fscanf(f, "%10c%10c%c%d\n", v[k].cognome,
        v[k].nome, &v[k].sesso, &v[k].anno) != EOF) {
        v[k].cognome[10]=v[k].nome[10]='\0'; k++;
    }
}
```

Legge esattamente 10
caratteri (spazi inclusi)

Legge 1 carattere e un
intero (ricordare &)

Ricordare il terminatore!