

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*.
 - dischi
 - nastri
 - cd
 - ...
- I file possono essere manipolati (aperti, letti, scritti...) all'interno di programmi C

IL CONCETTO DI FILE

- Un file è una *astrazione fornita dal sistema operativo*, il cui scopo è consentire la memorizzazione di informazioni su memoria di massa.
- Concettualmente, un file è una *sequenza di registrazioni (record) uniformi*, cioè dello stesso tipo.
- Un file è un'astrazione di memorizzazione di *dimensione potenzialmente illimitata* (ma non infinita), *ad accesso sequenziale*.

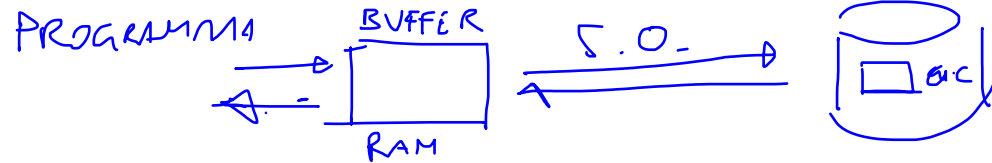
OPERARE SUI FILE

- 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
hard disk

- *in UNIX e Linux:*

/usr/temp/prova1.c
percorso *nome*



APERTURA DI FILE

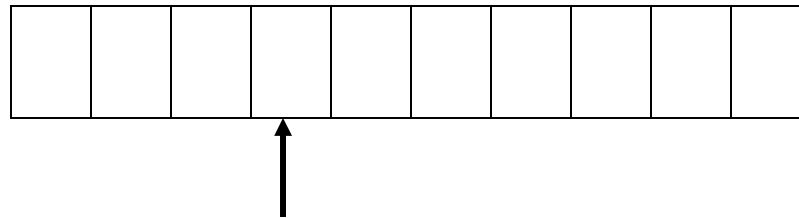
- Poiché un file è un'entità del sistema operativo, per agire su esso dall'interno di un programma occorre *stabilire una corrispondenza* fra:
 - il nome del file come risulta al sistema operativo
 - un nome di variabile definita nel programma.
- Questa operazione si chiama *apertura del file*
- Durante la fase di apertura si stabilisce anche la *modalità* di apertura del file
 - apertura in lettura
 - apertura in scrittura
 - *apertura in append*

APERTURA E CHIUSURA DI FILE

- Una volta aperto il file, il programma può operare su esso *operando formalmente sulla variabile definita al suo interno*
 - il sistema operativo provvederà a effettuare realmente l'operazione richiesta sul file associato a tale simbolo.
- Al termine, la corrispondenza fra *nome del file* e *variabile usata dal programma per operare su esso* dovrà essere *soppressa*, mediante l'operazione di *chiusura del file*.

ASTRAZIONE: testina di lettura/scrittura

- Una testina di lettura/scrittura (concettuale) 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.-> **accesso sequenziale al file**



- È illecito operare oltre la fine del file.

FILE IN C

STRUTTURA

- Per gestire i file, il C definisce il tipo **FILE**.
- **FILE** è una struttura definita nello header standard **stdio.h**, che l'utente non ha necessità di conoscere nei dettagli – e che spesso cambia da un compilatore all'altro!
- 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 *puntatori a FILE*.

L u i g i \0

0 1 0 0 1 0 0
2 byte

Come rappresentiamo i dati?

Ad esempio, vogliamo scrivere su un file i dati di una persona

```
char Nome[]="Luigi"
```

```
unsigned short int stipendio = 2000
```

- possiamo immaginare di ricopiare le celle di memoria che rappresentano le variabili direttamente sul file (6+2 byte): **File Binario**
- oppure possiamo immaginare di “stampare” il contenuto delle celle e scrivere sul file il risultato (6+4byte): **File di Testo**

Come rappresentiamo i dati?

Rappresentazione interna

- Più sintetica
- Non c'è bisogno di effettuare conversioni ad ogni lettura/scrittura
- Si può vedere il contenuto del file solo con un programma che conosce l'organizzazione dei dati

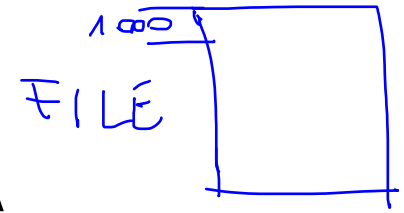
FILE BINARI

Rappresentazione esterna

- Meno sintetica
- ^{1 2 ' 5 ' 9 ' 0 ' 1} Necessità di conversione ad ogni lettura/scrittura
- Si può verificare il contenuto del file con un semplice editor di testo

ASCII
UTF8 1 & 8PC9-1
FILE di TESTO

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).

- **NB:** il nome del file (in particolare il path) è indicato in maniera diversa nei diversi sistemi operativi (\ nei percorsi oppure /, presenza o assenza di unità, etc).
In C per indicare il carattere '\ ' si usa la notazione '\\ '

FILE IN C: APERTURA

"r" lettura
"rt" testo

Per aprire un file si usa la funzione:

"wb" scrittura
binaria

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

modo specifica *come* aprire il file:

- **r** apertura in lettura (**read**). Se il file non esiste → **fallimento**.
- **w** apertura di un file vuoto in scrittura (**write**). Se il file esiste il suo contenuto viene cancellato.
- **a** apertura in aggiunta (**append**). Crea il file se non esiste.
- seguito opzionalmente da:
 - **t** apertura in modalità **testo** (default)
 - **b** apertura in modalità **binaria**
- ed eventualmente da
 - **+** apertura con possibilità di *lettura e scrittura*.

FILE IN C: APERTURA

Modi:

- **r+** apertura in lettura e scrittura. Se il file non esiste → **fallimento**.
- **w+** apertura un file vuoto in lettura e scrittura. Se il file esiste il suo contenuto viene distrutto.
- **a+** apertura in lettura e aggiunta. Se il file non esiste viene creato.

Nota: non si può passare da lettura a scrittura e viceversa se non si fa una operazione di **fflush**, **fseek** o **rewind** (V. prossimi lucidi)

FILE IN C: APERTURA

- Il *puntatore a FILE* restituito da **fopen()** si deve usare in tutte le successive operazioni sul file.
 - esso assume il valore **NULL** in caso l'apertura sia fallita
 - controllarlo è *il solo modo per sapere se il file è stato davvero aperto: non dimenticarlo!*
 - se non è stato aperto, il programma *non può proseguire* → **procedura exit()**

```
(#include <stdlib.h>)
```

```
... FILE *fp;
```

```
fp = fopen("esempio.txt", "rt");
```

```
if (fp==NULL)
```

```
{ printf("file esempio.txt non trovato");
```

```
exit(-1);
```

```
}
```

```
..... fclose(fp);
```

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 in caso di errore.
- Prima della chiusura, tutti i buffer vengono svuotati.

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 ASCII ≥ 32), altri no
 - la sequenza di caratteri è chiusa dal **carattere speciale EOF**

FILE DI TESTO (segue)

<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>

- tutte le funzioni da file acquistano una “f” davanti nel nome (qualcuna però cambia leggermente nome)
- tutte le funzioni da file hanno un *parametro in più*, che è appunto il puntatore al **FILE** aperto
- per riferimento: da terminale, **man 3 <nome funzione>**

ESERCIZIO

- Si scriva su un file di testo **prova.txt** quello che l'utente inserisce da tastiera parola per parola, finché non inserisce la parola "**FINE**".

FILE DI TESTO E CONSOLE

- In realtà, anche per leggere da tastiera e scrivere su video, il C usa le procedure per i file.
- Ci sono 3 file, detti canali di I/O standard, che sono 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.

FUNZIONE `feof()`

- Durante la fase di accesso ad un file è possibile verificare la presenza della marca di fine file con la funzione di libreria:

```
int feof(FILE *fp);
```

- **`feof(fp)`** controlla se è stata raggiunta la fine del file **`fp`** nella operazione di lettura precedente.
- Restituisce il valore
 - 0 (falso logico) se non è stata raggiunta la fine del file,
 - un valore diverso da zero (vero logico), se è stata raggiunta la fine del file

ESEMPIO

- Si mostri a video il contenuto di un file di testo il cui nome viene inserito da tastiera

```
Inserisci il nome di un file: infinito.txt
```

```
Sempre caro mi fu quest'ermo colle,  
E questa siepe, che da tanta parte  
De l'ultimo orizzonte il guardo esclude.  
Ma sedendo e mirando, interminati  
Spazi di la` da quella, e sovrumani  
Silenzi, e profondissima quiete  
Io nel pensier mi fingo, ove per poco  
Il cor non si spaura. E come il vento  
Odo stormir tra queste piante, io quello  
Infinito silenzio a questa voce  
Vo comparando: e mi sovvien l'eterno,  
E le morte stagioni, e la presente  
E viva, e 'l suon di lei. Così tra questa  
Immensita` s'annega il pensier mio:  
E 'l naufragar m'è dolce in questo mare.
```

ESERCIZIO

Un file di testo `rubrica.txt` contiene una rubrica del telefono, in cui per ogni persona è memorizzato

- `nome` (stringa di 20 caratteri senza spazi, incluso terminatore)
- `indirizzo` (stringa di 30 caratteri senza spazi, incluso `'\0'`)
- `numero` (stringa di 15 caratteri incluso `'\0'`)

Si scriva un programma C che legge da tastiera un nome, cerca la persona corrispondente nel file `rubrica.txt` e visualizza sullo schermo i dati della persona (se trovata)

ESEMPIO

Salvare su un file di testo `prova.txt` ciò che viene battuto sulla tastiera fino alla stringa `"fine"` usando la `fprintf`.

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

main()
{ FILE *fp;
  if ((fp = fopen("prova.txt", "w"))==NULL)
    exit(1); /* Errore di apertura */
  else
  { char s[81];
    scanf("%s", s);
    while (strcmp(s, "fine"))
    { fprintf(fp, "%s", s);
      scanf("%s", s);
    }
    fclose(fp);
  }
}
```

ESEMPIO

Stampare a video il contenuto di un file di testo **prova.txt** usando **fscanf**.

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

fp può essere NULL se il file richiesto non esiste

```
main()
{ FILE *fp;
  if ((fp = fopen("prova.txt", "r"))==NULL)
    exit(1); /* Errore di apertura */
  else
  {
    char s[81];
    while (!feof(fp))
      {fscanf(fp, "%s", s);
        printf("%s\n", s);
      }
    fclose(fp);
  }
}
```

clao\n

c | l | a | o | \n | \0

s

LETTURA DI STRINGHE

n

10

s[]

```
char *fgets (char *s, int n, FILE *fp);
```

- Trasferisce nella stringa s i caratteri letti dal file puntato da fp, fino a quando ha letto n-1 caratteri, oppure ha incontrato un newline, oppure la fine del file.
- Il carattere newline, se letto, e' mantenuto nella stringa s.
- Restituisce la stringa letta in caso di corretta terminazione; NULL in caso di errore o fine del file.

SCRITTURA DI STRINGHE

```
int fputs (char *s, FILE *fp);
```

- Trasferisce la stringa s (terminata da '\0') nel file puntato da fp. Non copia il carattere terminatore '\0' ne` aggiunge un newline finale.
- Restituisce un numero non negativo in caso di terminazione corretta; EOF altrimenti.

ESEMPIO COMPLETO FILE TESTO

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 COMPLETO FILE TESTO

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 COMPLETO FILE TESTO

Come organizzarsi?

- 1) Definire una struttura **persona**

Poi, nel main:

- 2) Definire un array di strutture **persona**
- 3) Aprire il file in lettura
- 4) Leggere una riga per volta, e porre i dati di quella persona in una cella dell'array
 - Servirà un indice per indicare la prossima cella libera nell'array.

ESEMPIO COMPLETO FILE TESTO

1) Definire una struttura di 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

ricordarsi lo spazio per il terminatore

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

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

- 2) definire un array di **struct persona**
- 3) aprire il file in lettura

```
main() {  
    struct persona v[DIM];  
    FILE* f = fopen("people.txt", "r");  
    if (f==NULL) {  
        .../* controllo che il file sia  
           effettivamente aperto */  
    }  
    ...  
}
```

Hp: massimo DIM
persone

ESEMPIO COMPLETO FILE TESTO

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

Uso fscanf

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

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

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**
- **fscanf(f, "%s%s%s%d", ...)**

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

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

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 COMPLETO FILE TESTO

Poi, nel main:

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

Dove mettere quello che si legge?

- Abbiamo definito un array di **struct persona**, **v**
- L'indice **k** indica la prima cella libera → **v[k]**
- Tale cella è una struttura fatta di *cognome*, *nome*, *sex*, *anno* → ciò che si estrae da una riga va nell'ordine in **v[k].cognome**, **v[k].nome**, **v[k].sex**, **v[k].anno**

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

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

E dopo aver letto una riga?

- L'indice **k** indica ancora la cella appena occupata → occorre incrementarlo, affinché indichi la prossima cella libera.

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

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

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

indica la prima
cella libera

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

ESEMPIO COMPLETO FILE TESTO

Poi, nel main:

- 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 COMPLETO FILE TESTO

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

Dichiara la procedura `exit()`

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

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

VARIANTE

E se usassimo un singolo carattere per rappresentare il sesso?

Non più:

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

Ma:

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

VARIANTE

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!** La specifica non lo dice!
- Quindi, non possiamo sapere a priori dov'è il carattere che ci interessa!

VARIANTE

Infatti, il nostro file potrebbe essere fatto così:

Rossi Mario M 1947	Qui, uno spazio prima di M
Ferretti Paola F 1982	Qui, due spazi prima di F
Verdi Marco M 1988	
Bolognesi Annarita F 1976	Qui, tre spazi prima di M
...	Qui, uno spazio prima di F

- prima, dicendo a **fscanf** si leggere una stringa, gli spazi (uno, due, tre..) erano eliminati comunque
- adesso, dicendo a **fscanf** si leggere un carattere singolo, **dobbiamo decidere noi cosa fare.**

VARIANTE

Due possibilità:

- scelta "furba": *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 "fai da te": 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 - VERSIONE "FURBA"

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

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

Un singolo carattere

```
main() {
    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", v[k].cognome,
        v[k].nome, s, &v[k].anno ) != EOF){
        v[k].sesso = s[0]; k++; }
}
```

Stringa ausiliaria

Copiatura carattere

VARIANTE - VERSIONE "FAI DA TE"

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

Un singolo carattere

Carattere ausiliario

```
main() {  
    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", &v[k].anno); k++; }  
}
```

Legge solo co-
gnome e nome

Salta spazi

Copia il
carattere

Legge l'anno

VARIANTE - VERSIONE "FAI DA TE"

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

```
main() {  
    persona v[100];  
    if ((f = fopen("dati.txt", "r")) == NULL) {  
        perror("Il file non esiste!"), exit(1); }  
    while(fscanf(f, "%s", v[k].cognome,  
                v[k].nome) != EOF){  
        do fscanf(f, "%c", &ch); while (ch==' ');  
        v[k].sesso = ch;  
        fscanf(f, "%s", v[k].cognome, v[k].nome);  
        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 dall'indirizzo

ESEMPIO 4

È dato un file di testo **e1enco.txt** le cui righe rappresentano ciascuna i dati di una persona, secondo il seguente formato:

- **cognome** (esattamente 10 caratteri)
- **nome** (esattamente 10 caratteri)
- **sesso** (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

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 **e**elenco.txt:

```
Rossi      Mario      M1947
Ferretti   Paola      F1982
De Paoli   Gian Marco M1988
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` in un'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 aggiungerlo a parte. per altra 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>
```

Sappiamo esattamente la dimensione: 10 +1

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

Legge esattamente 10 caratteri (spazi inclusi)

```
main() {
    persona v[DIM]; int k=0; FILE *f;
    if ((f=fopen("elenco.txt", "r"))==NULL)
        perror("Il file non esiste!");
```

Legge 1 carattere e un intero (ricordare &)

```
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++;}
}
```

Ricordare il terminatore!