GESTIONE DEI FILE

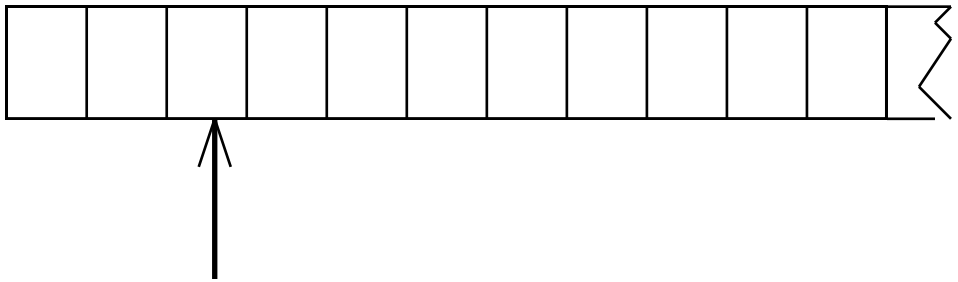
Per poter mantenere disponibili i dati tra le diverse esecuzioni di un programma (***persistenza*** dei dati) è necessario poterli ***archiviare*** *su memoria di massa*

- 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***

FILES IN C

- In C gli streem (flussi) di dati sono gestite in modo uniforme mediante l'astrazione file, lavorando o su dispositivi di memoria di massa (disco rigido…) o device fisici (stampante tastiera..).

- **Il tipo `FILE` è una struttura definita in header standard `<stdio.h>`,** che l'utente non ha necessità di conoscere nei dettagli (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.h`**
- **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***
  - I file di testo sono sequenze ordinate di caratteri organizzati in righe
  - I file binari sono sequenze ordinate di bytes

➢ **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

Gli stream aperti (tranne quelli standard) dovrebbero essere chiusi prima della terminazione del programma

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*** (il buffer è una memoria intermediaria che compensa differenze di velocità nel trasferimento o nella trasmissione di dati fra computer e periferiche e viene gestita normalmente come una coda)

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*

Attenzione: la costante 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

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 monitor**
- ***stderr*** **è un altro file di testo aperto in scrittura, di norma agganciato al monitor**

Lettura e Scrittura files di testo

Funzione da console	*Funzione da file*
`int getchar(void);`	`int fgetc(FILE* f);`
`int putchar(int c);`	`int fputc(int c, FILE* f);`
`char* gets(char* s);`	`char* fgets(char* s, int n, FILE* f);`
`int puts(char* s);`	`int fputs(char* s, FILE* f);`
`int printf( … );`	`int fprintf(FILE* f, … );`
`int scanf( … );`	`int fscanf(FILE* f, … );`

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

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

```
getchar()  ≡ fgetc(stdin)
putchar(c) ≡ fputc(c, stdout)
```

I/O con FORMATO

La libreria standard offre due funzioni di I/O *di uso generale*, che compendiano tutte le necessità precedenti: printf() e scanf()

```
int printf(...);
```

- scrive sul canale di output una serie di valori, *effettuando le conversioni richieste ove necessario*
- restituisce il numero di ***caratteri emessi*** oppure `EOF` in caso di errore

```
int scanf(...);
```

- legge dal canale di input una serie di *campi*, *effettuando le conversioni richieste ove necessario*
- restituisce il numero di ***campi letti con successo*** oppure `EOF` in caso di errore

Esempio di Lettura su Files di Testo

```
N=fscanf(fd, "%d %d", &x, &y);
```

Ha i seguenti possibili valori di ritorno per `N`:

- **`2`** sono stati letti (convertiti con successo) due interi, memorizzati nelle varabili x e y;
- **`1`** un solo intero è stato letto con successo, ed è stato memorizzato nella variabile x; non è stato possibile leggere il secondo intero: questo può essere dovuto al fallimento di conversione oppure al fatto che è stata incontrata la fine del file;
- **`0`** non è stato possibile leggere nessuno dei due numeri interi: questo è dovuto alla presenza di caratteri non interpretabili come interi su file;
- **`EOF`** nessun intero è stato letto da file, perchè si è incontrata la fine del file prima ancora di riuscire a leggere il primo intero.

ESEMPIO 1

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

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

```
#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);
  }
}
```

La funzione exit

`void exit (int status)`

- Si trova in `stlib.h`
- Termina il processo che richiama questa funzione e ritorna il valore di uscita "status".
- Il sistema operativo è in grado di leggere il valore dello stato.
- Per convenzione, uno stato uguale a 0 significa "fine normale", mentre qualsiasi altro valore indica un errore o un evento insolito.

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

Scrive un intero in coda a un file

```
#include <stdlib.h>
#include<stdio.h>
int main() {
      FILE *fd; int x=-32;
/* apre il file in scrittura in coda */
      fd=fopen("scrivi.txt", "a");
      if( fd==NULL )
      { perror("Errore in apertura del file");
      exit(1); }
/* scrive il numero */
fprintf(fd, "%d\n", x);
/* chiude il file */
fclose(fd);
return 0; }
```

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, ad esempio **`'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
 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>

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++;
}
```

Dichiara la procedura **exit()**

ESEMPIO 3: VARIANTE

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

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

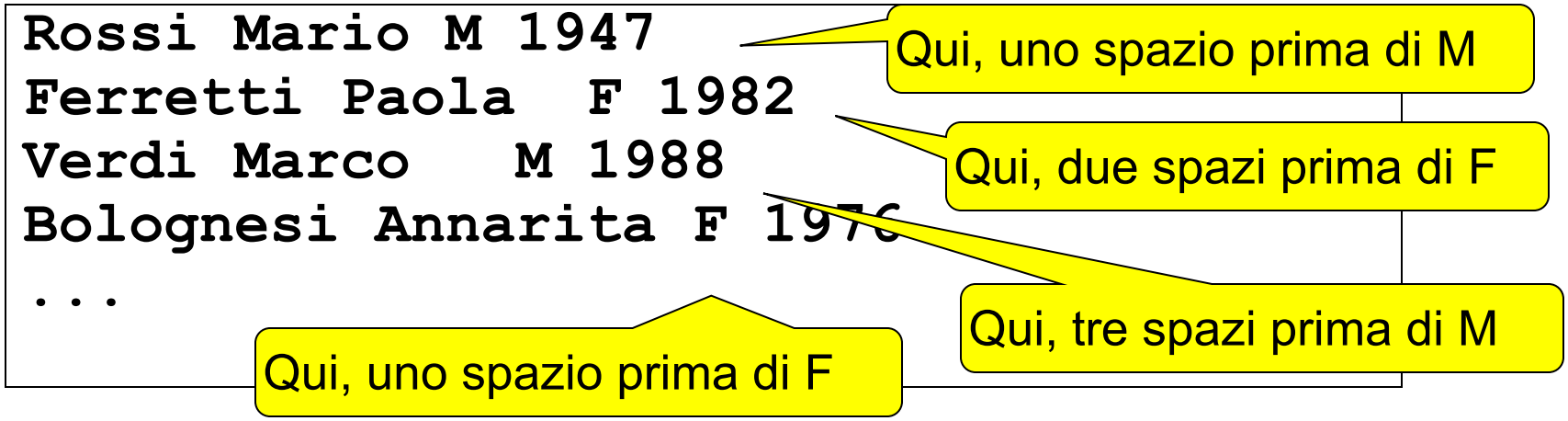
ESEMPIO 3: VARIANTE

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:



- 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;

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++; }
}
```

Stringa ausiliaria

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 mai
 persona                                          ch;
 if ((f=                                          {
  perror(                                         }
 while(fscanf(f,          s", v[k].cognome,
                          v[k].nome) != EOF){
  do fscanf(f,"%c", &ch); while (ch==' ');
   v[k].sesso = ch;
  fscanf(f,"%
}
```

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)
- **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

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 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()` 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",…)`**
  -
  - ssario
- **pe altra modalità, *specificando ti caratteri leggere*.** Ad esempio, per leggerne dieci:

ATTENZIONE: viene riempito un array di caratteri, *senza inserire alcun terminatore*
Occorre aggiungerlo a parte

```
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"))==
  perror("Il file non esiste!");
 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!