Gestione dei file nel linguaggio C

Parte 1: utilizzo dei file *nel linguaggio C*

Per memorizzare permanentemente informazioni all'interno di un calcolatore occorre usare un **file**. Il linguaggio C fornisce al programmatore tutti gli strumenti necessari per l'utilizzo di file.

Poiché è il sistema operativo ad occuparsi della gestione dei dispositivi fisici utilizzati dal calcolatore per gestire i file, un programma C che utilizza file deve interfacciarsi con il sistema operativo in modo da utilizzarne le parti necessarie. Dato che queste parti *cambiano al cambiare del sistema operativo*, il compilatore deve tenere conto del sistema operativo presente sulla macchina nella quale il programma viene compilato e deve produrre un codice eseguibile che utilizzi le corrette chiamate al sistema operativo.

In C il programmatore non deve in nessun modo preoccuparsi di questi problemi: infatti egli non opera direttamente sui file, bensì utilizza sempre, per farlo, opportune *funzioni di libreria*. Sarà il processo di compilazione di queste funzioni ad occuparsi di produrre un codice adatto al particolare sistema operativo scelto. (Gli sviluppatori delle funzioni di libreria hanno invece dovuto tenere conto, nel loro sviluppo, delle caratteristiche dei diversi sistemi operativi.)

Dunque il linguaggio C, inteso in senso stretto, *non* definisce le operazioni di gestione dei file; queste sono rese disponibili al programmatore attraverso l'uso di opportune funzioni della *standard library*.
La standard library del C contiene l'intestazione delle funzioni dedicate alla gestione dei file nello *header file* **stdio.h**, che va quindi incluso (con una direttiva `#include`) in ogni programma che utilizza file. Come vedremo tra breve, "utilizza file" include alcune operazioni che non sembrano riguardare ciò che si intende con "file" nel linguaggio comune.

* * * * *

In realtà noi abbiamo già visto ed utilizzato, anche se in modo "mascherato", dei file. Le funzioni **printf** e **scanf**, che non a caso sono definite in *stdio.h*, sono infatti funzioni che operano su file. Il motivo di ciò risiede nel fatto che il sistema operativo "mostra" al programmatore le periferiche standard di ingresso (tastiera) e di uscita (schermo) **esattamente come se fossero file**.

In altri termini, dal punto di vista di chi scrive un programma C la scrittura sullo schermo o su un file posto sul disco fisso utilizza esattamente le stesse operazioni; analogamente la lettura di dati da tastiera o da un file sul disco fisso utilizza le stesse identiche operazioni.
Come vedremo, infatti, le funzioni printf e scanf sono semplicemente forme semplificate (perché si riferiscono ad un file prestabilito, invece che ad un file qualsiasi scelto dal programmatore) di funzioni più generali di scrittura su file: **fprintf** e **fscanf**.

* * * * *

Per rappresentare un file (inclusi quelli che corrispondono a dispositivi di *input/output*) il

linguaggio C utilizza uno strumento chiamato **stream**, o **flusso di comunicazione**. Il compilatore C rappresenta ogni stream con una struttura dati interna (nel senso che non viene direttamente manipolata dal programmatore) il cui tipo è chiamato **FILE**. FILE (in maiuscolo) è una parola chiave del C, e rappresenta appunto il tipo di dato che corrisponde ad uno stream. Si tratta di un tipo di dato predefinito (inbuilt) del C, al pari di *int*, *char*, *float*, ... Il tipo FILE si può immaginare come una struct che contiene diversi campi, ciascuno dei quali contiene una delle informazioni che riguardano lo stream.

Uno stream non viene ***mai*** utilizzato dal programmatore manipolando direttamente i campi della variabile di tipo FILE che rappresenta lo stream. In effetti non è necessario, né utile, per il programmatore conoscere le caratteristiche interne del tipo FILE. Tale variabile viene sempre e solo manipolata tramite opportune chiamate a funzioni di libreria (ad esempio la funzione `printf`). Sono queste ultime a compiere effettivamente sullo stream le operazioni richieste.

Più precisamente, a ciascuno degli stream utilizzati da un programma viene associata una variabile di tipo **FILE*** (puntatore a FILE); quest'ultima viene passata come parametro alle funzioni di libreria tutte le volte che lo stream va utilizzato. Come vedremo più avanti, l'associazione tra la variabile di tipo FILE* e lo stream avviene tramite una apposita funzione di libreria chiamata `fopen`. Spesso la variabile di tipo FILE* associata ad uno stream viene chiamata a sua volta, per semplicità ma in modo un po' confuso, “stream”.

* * * * *

Come già accennato, in C i dispositivi standard di input e output del sistema operativo (ovverosia tastiera -input- e schermo -output-) vengono rappresentati come file, e come tali vengono utilizzati. Per tale ragione, ai dispositivi standard corrispondono, nel C, tre **stream standard**:

- lo **standard input**, automaticamente associato alla tastiera e manipolabile attraverso il puntatore **stdin**, di tipo FILE*;
- lo **standard output**, automaticamente associato allo schermo e manipolabile attraverso il puntatore **stdout**, di tipo FILE*;
- lo **standard error**, automaticamente associato anch'esso allo schermo e manipolabile attraverso il puntatore **stderr**, di tipo FILE*.

stdin, *stdout* e *stderr* sono parole chiave del C, definite automaticamente dal compilatore C (purché *stdio.h* sia stato incluso nel programma).

Come vedremo, per poter utilizzare qualsiasi stream occorre prima **aprirlo**; dopo l'utilizzo occorre invece ricordarsi di **chiudere** lo stream. Nel caso degli stream *standard input*, *standard output* e *standard error* le operazioni di apertura e chiusura vengono gestite automaticamente dal compilatore ed il programmatore non deve preoccuparsene affatto, diversamente da quanto accade per tutti gli altri stream.

* * * * *

In C esistono due tipi di stream: gli stream **di tipo testo** e quelli **di tipo binario**.
Gli stream di tipo testo sono usati in tutti i casi in cui il programmatore desidera che il relativo file venga interpretato dal C come una successione di dati di tipo *char*, alcuni dei quali saranno caratteri "stampabili" (come le lettere o i numeri) mentre altri saranno caratteri "speciali" che rappresentano comandi da eseguire in fase di stampa. Ad esempio, in un file di tipo testo il carattere indicato con '\n' (*new line*) viene interpretato come la richiesta di eseguire un'andata a capo.
Al contrario, gli stream di tipo binario sono usati quando si vuole leggere o scrivere il file byte per byte, senza occuparsi in alcun modo del significato di tali dati. Ad esempio i file binari sono utili quando si usa un file per memorizzare uno o più dati di tipo *struct*. Ciascuno di tali dati, infatti, occuperà un determinato numero di byte per contenere tutti i suoi campi: alcuni (o tutti) di tali byte non potranno, in genere, essere interpretati come caratteri.

Gli stream predefiniti ***standard input, standard output*** **e** ***standard error*** **sono di tipo testo[1].**

* * * * *

In un programma C, ogniqualvolta si desidera usare uno o più file occorre:

1. Includere la libreria *stdio.h*, inserendo nel programma la direttiva

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

2. Dichiarare una variabile di tipo FILE* per ciascuno dei file da utilizzare, come nell'esempio seguente:

```
FILE* MioFile;
```

3. Aprire ciascun file, associando ad esso una delle variabili di tipo FILE*: ciò avviene tramite una istruzione del tipo

```
MioFile = fopen("file_da_elaborare.txt", "r");
```

4. Elaborare i file utilizzando le funzioni della libreria *stdio.h.* Per identificare quale file vada di volta in volta elaborato, alla funzione chiamate andrà passata la variabile di tipo FILE* che è stata associata a tale file. Considerando l'esempio del punto 3, per applicare una funzione al file chiamato "file_da_elaborare.txt" andrà passata alla funzione la variabile MioFile.

5. Una volta terminata l'elaborazione di ciascun file, chiuderlo. Ciò viene eseguito tramite una opportuna chiamata alla funzione `fclose`. Ad esempio, per chiudere il file chiamato "file_da_elaborare.txt" al quale era stata associata la variabile di tipo FILE* chiamata MioFile, si utilizzerà l'istruzione

1 Nella pratica la distinzione tra stream testuali e binari è significativa soltanto per alcuni sistemi operativi. Per i sistemi operativi compatibili con lo standard POSIX, infatti, gli stream dei due tipi (pur se gestiti diversamente dal C) vengono trattati nello stesso modo dal sistema operativo. Linux e Windows sono esempi di sistemi operativi, rispettivamente, compatibile e non compatibile con POSIX: di conseguenza, per Linux la differenza tra stream binari e testuali non è significativa mentre per Windows lo è.

```
fclose(MioFile);
```

La chiusura dei file utilizzati dal programma è particolarmente importante, poiché garantisce che essi non vengano danneggiati e che tutto ciò che è stato scritto in essi giunga effettivamente a destinazione. Ulteriori informazioni in merito saranno fornite in seguito, quando sarà descritta la funzione `fflush`.

* * * * *

E' interessante notare che in fase di esecuzione di un programma già compilato le associazioni predefinite tra *standard input*, *standard output* e *standard error* e le periferiche tastiera e schermo possono essere modificate dal sistema operativo, senza modificare il codice sorgente del programma. Ciò si ottiene utilizzando la cosiddetta **ridirezione dell'input e dell'output**. Questa *non* è una operazione C, bensì una caratteristica del sistema operativo (dunque ha forme diverse a seconda del sistema utilizzato).

Ad esempio, se *nomefile* è il nome di un file di testo e *programma* è il nome di un programma eseguibile (contenuto nella directory corrente) ottenuto dalla compilazione di un programma C, scrivere da terminale

```
./programma < percorsofile
```

esegue *programma* associando lo *standard input* al file su disco raggiungibile seguendo il percorso *percorsofile* (ad esempio *percorsofile* potrebbe essere */home/nomeutente/dati/filedati.txt*) anziché alla tastiera. In questo modo, ogni volta che *programma* leggerà dallo *standard input*, i dati richiesti dal programma verranno letti da *percorsofile* e non dalla tastiera.
Questa ridirezione dell'input è possibile perché sia `percorsofile` sia la tastiera sono gestiti come file: il primo perché si tratta effettivamente di un file fisico situato (ad esempio) sul disco fisso, la seconda perché, come abbiamo visto, dal punto di vista del C essa corrisponde ad un file.
Analogamente scrivendo

```
./programma > percorsofile
```

programma viene eseguito associando lo *standard output* al file su disco raggiungibile seguendo il percorso *percorsofile* anziché alla tastiera. Dunque ogni volta che *programma* scriverà sullo *standard output* i dati verranno scritti in *percorsofile* e non sullo schermo. Se *percorsofile* esiste già, il suo contenuto verrà sovrascritto; altrimenti verrà creato un nuovo file di testo con tale nome.
Infine, se *programma1* e *programma2* sono due programmi eseguibili contenuti nella directory corrente, scrivere

```
./programma1 | ./programma2
```

esegue *programma1* e *programma2* utilizzando come *standard output* del primo lo *standard input* del secondo. In pratica *programma1* e *programma2* verranno eseguiti

contemporaneamente: lo stream di output di *programma1* (normalmente associato allo schermo) verrà inviato a *programma2*, che lo utilizzerà come stream di input (normalmente associato alla tastiera). L'operatore | del sistema operativo, detto **pipe** (“tubo”) serve cioè a collegare l'uscita di un programma all'ingresso di un altro, ovverosia a far lavorare il secondo sui dati prodotti dal primo.
Nei sistemi Linux, il comando *grep* serve ad estrarre da un file di testo le sole linee contenenti una stringa specifica. Pertanto, un uso tipico dell'operatore | è l'estrazione con *grep* delle sole parti utili dell'output di un comando. Ad esempio

```
ls | grep “a”
```

lancia il comando ls (che elenca tutti i file contenuti nella directory corrente) ma mostra a schermo i soli nomi di file che contengono la lettera *a*. Analogamente,

```
gcc -Wall -o test programmaC.c | grep “warning”
```

lancia la compilazione con gcc del programma C chiamato *programmaC.c* e mostra a schermo, anziché l'intero output di gcc, le sole linee contenenti la parola “warning”.

Le operazioni di ridirezione dell'input e dell'output sopra descritte (e la relativa sintassi) sono utilizzabili con tutti i sistemi operativi più diffusi: Windows, MacOS X e Linux.

Parte 2: funzioni C per la gestione dei file

Qui sotto sono elencati i *prototipi* (ovverosia le sole intestazioni) delle principali funzioni di gestione file contenute nella libreria *stdio.h*, suddivisi per categorie. Alcune delle funzioni sotto elencate sono già note (`printf`, `scanf`) o sono già state anticipate nella Parte 1 di questo documento (`fopen`, `fclose`).
Nel seguito della Parte 2, ciascuna delle funzioni sotto elencate sarà brevemente descritta.

Apertura e chiusura:

FILE ***fopen**(const char **path*, const char **mode*)
int **fclose**(FILE **stream*)

Gestione degli errori (argomento non richiesto all'esame):

int **ferror**(FILE **stream*)
int **feof**(FILE **stream*)
void **clearerr**(FILE **stream*)

Lettura e scrittura formattata [usata in genere per gli stream testuali]:

int **scanf**(const char **format*, ...)
int **fscanf**(FILE **stream*, const char **format*, ...)
int **printf**(const char **format*, ...)
int **fprintf**(FILE **stream*, const char **format*, ...)

Lettura e scrittura di un singolo carattere [usata in genere per gli stream binari]:

int **getchar**(void)
int **putchar**(int *c*)
int **fgetc**(FILE **stream*) [o l'analoga int **getc**(FILE **stream*)]
int **fputc**(int *c*, FILE **stream*) [o l'analoga **putc**(int *c*, FILE **stream*)]

Lettura e scrittura di stringhe o linee di testo [usata in genere per gli stream testuali]:

char ***gets**(char **s*)
int **puts**(const char **s*)
char ***fgets**(char **s*, int *size*, FILE **stream*)
int **fputs**(const char **s*, FILE **stream*)

Lettura e scrittura per blocchi di dati (usata in genere per gli stream binari):

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

Accesso diretto ad un punto di un file:

int **fseek**(FILE **stream*, long *offset*, int *whence*)
long **ftell**(FILE **stream*)
void **rewind**(FILE **stream*)

Funzioni ausiliarie:

int **fflush**(FILE* stream)

APERTURA E CHIUSURA

Prima di poter utilizzare un file, esso va *aperto*. La funzione

FILE ***fopen**(const char **path*, const char **mode*)

apre il file il cui nome (o il path completo) è contenuto nella stringa *path* (o lo crea se non esiste già) ed associa uno stream ad esso, restituendo il puntatore alla struttura di tipo FILE associata allo stream. In caso di errori la funzione restituisce NULL.
Notare che la sintassi con la quale viene espresso *path* dipende dal sistema operativo usato. Se non viene specificato il percorso completo (che, ad esempio, potrebbe avere una forma del tipo *C:\Documenti\MieiDocumenti\Nomefile.estensione* per un sistema Windows) il programma cerca il file nella directory corrente, ovvero quella che contiene il programma compilato.
Il file viene aperto con modalità diverse a seconda del valore della stringa *mode*:

"r": lettura come stream di tipo testo, dall'inizio
"w": scrittura come stream di tipo testo, dall'inizio
"a": scrittura come stream di tipo testo, dalla fine (*append*)
"r+", "w+": lettura e scrittura come stream di tipo testo, dall'inizio
"a+": scrittura come stream di tipo testo, dalla fine (*append*) e contemporaneamente lettura come stream di tipo testo, dall'inizio
"rb", "wb", "ab", "rb+", "wb+", "ab+": come sopra ma come stream di tipo binario[1]

Se il file aperto con `fopen` esiste già, nelle modalità di apertura w e wb esso viene troncato a lunghezza 0 (ovvero il suo contenuto viene eliminato). Se invece si utilizzano le modalità w+, rb+ e wb+ il file rimane inalterato a meno di eventuali operazioni di scrittura dati all'interno di esso. Quando tali operazioni di scrittura avvengono, esse sovrascrivono la parte di file dove il nuovo dato viene inserito, ma non alterano le altre parti del file.

Si noti che se il file di cui viene richiesta l'apertura non esiste, `fopen` restituisce NULL solo quando il file viene aperto in lettura. Se il file viene aperto in scrittura, `fopen` crea un nuovo file (vuoto). Viene restituito NULL solo se tale creazione è impossibile, ad esempio perché la directory di lavoro è protetta da scrittura.

E' buona norma controllare che il file venga aperto correttamente, utilizzando codice simile a quello dell'esempio seguente:

1 Nei sistemi Unix/Linux non esiste differenza tra le modalità di apertura come stream di tipo testo e come stream di tipo binario; dunque in tali sistemi l'eventuale carattere "b" nel parametro *mode* viene ignorato. Inoltre in certi casi "wb+" non viene riconosciuta come apertura in lettura e scrittura, per cui è preferibile usare, in sua vece, "rb+".

```
FILE* Stream;

Stream = fopen("nomefile.dat", "r");

if ( NULL == Stream )
{
   printf("Errore di apertura file!");
}
else
{
   ...elaborazione file...
}
```

Una forma alternativa (spesso usata) di questo frammento di programma è la seguente:

```
if ( NULL == (Stream = fopen("nomefile.dat", "r")) )
{
   printf("Errore di apertura file!");
}
else
{
   ...elaborazione file...
}
```

La prima forma è decisamente migliore perché evita di "nascondere" un assegnamento all'interno della condizione da verificare. La seconda forma (da evitare!) funziona perché l'assegnamento, *valutato come espressione* (come in questo caso), restituisce il valore assegnato.

* * * * *

Ciascuno dei file aperti con fopen deve essere chiuso prima che il programma termini. A tale scopo occorre che venga chiamata la funzione

int **fclose**(FILE ***stream*)

passandole il puntatore alla struttura FILE associata al file.
La funzione restituisce 0 in assenza di errori, **EOF** in caso di errori. *EOF* è una costante intera definita in *stdio.h*, utilizzata da molte funzioni di *stdio.h*.

GESTIONE DEGLI ERRORI

La struttura di tipo FILE che rappresenta uno *stream* contiene due campi che registrano le eventuali condizioni di errore incontrate durante l'elaborazione del relativo file, chiamati

eof (*end of file*): registra un tentativo di lettura oltre la fine del file;
error: registra altri tipi di errore.

Per verificare lo stato di tali campi, e dunque se si sono verificati errori, si usano le seguenti funzioni:

int **feof**(FILE *_stream_),

che restituisce un valore diverso da 0 se il campo eof ha registrato un errore, e 0 altrimenti;

int **ferror**(FILE *_stream_),

che restituisce un valore diverso da 0 se si sono verificati errori durante la gestione del file, o 0 se non se ne sono verificati.

Dopo la rilevazione di una condizione di errore, i campi error ed eof di una struttura FILE (e dunque i valori restituiti dalle funzioni feof e ferror) mantengono i loro valori fino a che vengono riportati alla condizione iniziale di "nessun errore". A tale scopo si usa la funzione

void **clearerr**(FILE *_stream_)

LETTURA E SCRITTURA FORMATTATA [usata in genere per gli stream testuali]

Le operazioni di lettura e scrittura formattata consentono di specificare il formato dei dati da leggere e scrivere e di indicare operazioni di *conversione di formato*, attraverso l'utilizzo di opportuni *caratteri di conversione* posti nella stringa *format*. Non ci dilunghiamo su di esse perché le funzioni scanf e printf sono già ben note, e fscanf ed fprintf differiscono da esse solo perché lo stream considerato è esplicitamente fornito dall'utente anziché essere implicito (*stdin* per scanf e *stdout* per printf).

int **scanf**(const char *_format_, ...)
int **fscanf**(FILE *_stream_, const char *_format_, ...)

int **printf**(const char *_format_, ...)
int **fprintf**(FILE *_stream_, const char *_format_, ...)

Le funzioni scanf e fscanf restituiscono il numero di dati effettivamente assegnati alle variabili specificate dopo la virgola che segue la stringa di formattazione, secondo le conversioni di tipo specificate da quest'ultima. In altri termini si dice che restituiscono il *numero di conversioni* effettuate. Questo numero può essere inferiore al numero di variabili, o perfino nullo, ad esempio perché si è verificato un *matching failure*: una differenza tra il tipo del dato fornito alla funzione ed il tipo specificato dal corrispondente campo della stringa di formattazione.
Se avviene un errore di lettura del file oppure si raggiunge la fine del file la funzione restituisce *EOF*. Il fatto che queste condizioni si siano verificate può essere controllato con le funzioni feof e/o ferror.

Le funzioni printf e fprintf restituiscono il numero di caratteri effettivamente scritti con successo nello stream (*non* incluso il '\0' finale nel caso si stampino stringhe di caratteri). Se avviene un errore di scrittura del file la funzione restituisce un numero negativo. Il fatto che

questa condizione si sia verificata può essere controllato con la funzione ferror.

> **Nota.** Nel caso in cui *stdin* corrisponda alla tastiera, per terminare la lettura l'utente deve premere il tasto "enter". Ciò corrisponde all'inserimento da tastiera del carattere '\n' (a capo). Tale carattere *non* viene eliminato, ma rimane a disposizione per eventuali future letture di dati da tastiera. Se la lettura immediatamente successiva è una lettura di dati di tipo di tipo numerico, il carattere '\n' viene ignorato; al contrario, se avviene la lettura di caratteri, il primo carattere ad essere letto è proprio il '\n' residuo della lettura precedente. Spesso ciò causa problemi: per evitarli è possibile eliminare il '\n' utilizzando l'istruzione
>
> ```
> scanf("%*c");
> ```
>
> che legge un carattere e lo elimina senza memorizzarlo da nessuna parte.

Per completezza osserviamo che la libreria stdio.h contiene anche una funzione, sscanf, del tutto analoga a scanf e fscanf, ma da applicare ad una stringa (array di caratteri terminati da '\0') anziché ad un file. Essa è definita come

```
int sscanf ( const char * str, const char * format, ...)
```

dove str è la stringa dalla quale leggere. I valori restituiti da sscanf sono gli stessi di scanf e fscanf. Questa funzione può essere utile, ad esempio, per estrarre dati dagli elementi dell'array *argc[ ]* che contiene gli argomenti passati alla funzione *main* tramite la linea di comando.

LETTURA E SCRITTURA DI SINGOLI CARATTERI [usata in genere per gli stream binari]

Un altro modo per eseguire la lettura e la scrittura di un file è procedere un singolo carattere alla volta. Tale operazione corrisponde alla lettura di un singolo byte, ovverosia di 8 bit, dato che in C un carattere è rappresentato da 1 byte. Nel caso di un *unsigned char*, gli 8 bit sono utilizzati per rappresentare i valori binari da 0 a 255.
Per la lettura di un singolo carattere si utilizza la funzione

int **fgetc**(FILE **stream*) [o l'analoga int **getc**(FILE **stream*)]

Essa legge un carattere dallo stream *stream* e lo restituisce come unsigned char trasformato (attraverso una operazione di conversione di tipo, detta operazione di *type casting*) in un integer. Non viene restituito direttamente un unsigned char perché nel caso in cui venga incontrata la fine del file o si verifichi un errore di lettura fgetc restituisce *EOF*, che non è rappresentabile con un unsigned char[1]. In tutti i casi in cui il dato restituito non è EOF, il suo valore numerico coincide con il valore numerico del carattere corrispondente. Una conseguenza di ciò è che è possibile eseguire confronti del tipo

```
if ( fgetc(...) == 'a')
```

o assegnamenti del tipo

```
VariabileChar = fgetc(...)
```

1 Infatti il valore di EOF è l'intero negativo -1. Questo valore numerico non va mai usato, preferendo invece l'uso dell'identificatore EOF! In teoria, infatti, il valore numerico di EOF potrebbe essere modificato in future implementazioni del linguaggio C.

esattamente come se fgetc restituisse direttamente un dato di tipo *char*. La cosa è resa possibile dal meccanismo di conversione automatica da *int* a *char* del C.
La funzione

int **getchar**(void) equivale a int **fgetc**(stdin).

Si noti che, nel caso in cui la lettura di carattere avvenga da tastiera, sia fgetc che getchar lasciano nel buffer associato al file il carattere '\n' inserito per terminare la lettura (si veda la precedente sezione "LETTURA E SCRITTURA FORMATTATA"). Tale carattere rimane dunque a disposizione di eventuali successive operazioni di lettura di caratteri, a meno che venga eliminato con il metodo descritto in precedenza.
La funzione

int **fputc**(int *c*, FILE **stream*) [o l'analoga **putc**(int *c*, FILE **stream*)]

scrive nello stream *stream* l'intero *c*, dopo averlo trasformato (con una operazione di *type casting*) in un unsigned char. Restituisce l'intero scritto oppure, in caso di errore, *EOF*.

int **putchar**(int *c*) equivale a int **fputc**(int *c*, stdout)

LETTURA O SCRITTURA DI STRINGHE O LINEE DI TESTO [usata in genere per gli stream testuali]

Spesso nel caso di file contenenti testo la lettura e la scrittura vengono eseguite trasferendo in blocco intere stringhe o linee di testo.

char ***gets**(char *s)

continua a leggere caratteri dallo *standard input* fino a che raggiunge un carattere *newline* ('\n') o la fine del file, e li registra uno dopo l'altro nell'array di caratteri *s*; subito dopo l'ultimo carattere inserito, gets aggiunge un carattere di terminazione '\0'. Il valore restituito è *s* in caso di lettura corretta, NULL in caso contrario.

ATTENZIONE: gets non fa alcun controllo sulla *quantità* di dati scritti nell'array *s*: se le dimensioni di *s* sono insufficienti, la funzione continua a scrivere nelle regioni di memoria adiacenti (problema di *buffer overrun*). Poiché starebbe a chi *usa* il programma limitare il numero di caratteri alla dimensione di *s* (che invece non è nota all'utente), questa situazione è evidentemente pericolosa. Di conseguenza **la funzione gets non va mai usata**. Per gestire un input di linee di testo dallo *standard input* va utilizzata invece fgets.
Per la stessa ragione occorre assolutamente evitare di eseguire l'input di linee di testo utilizzando scanf (o fscanf). Il codice

```
scanf("%s",NomeArray)
```

legge da tastiera caratteri e li inserisce (seguiti da un carattere di terminazione '\0') in un array di caratteri il cui identificatore è NomeArray; ma, come gets, non esegue alcun controllo sul

numero di caratteri effettivamente inseriti dall'utente e dunque presenta forti rischi.

char ***fgets**(char **s*, int *size*, FILE **stream*)

si comporta in modo analogo a gets ma richiede di specificare la dimensione *size* del vettore destinato a ricevere i caratteri letti e può essere usata per stream diversi da *stdin*. Legge caratteri fino a che non raggiunge un '\n' o la fine del file oppure fino a che ne ha letti *size*-1 (l'ultimo carattere di *s* serve per il '\0'). Notare che se viene letto un '\n' esso *viene* memorizzato in *s* (e viene seguito dal '\0' conclusivo).

int **puts**(const char **s*)

scrive la stringa contenuta in *s* (privata del '\0' finale) nello *standard output*, facendola seguire da un *newline* ('\n'). La funzione restituisce un numero intero non negativo nel caso l'operazione si concluda con successo, *EOF* in caso di insuccesso.

int **fputs**(const char **s*, FILE **stream*)

scrive la stringa contenuta in *s* (privata del '\0' finale) nello stream *stream* (senza farla seguire da un *newline*). La funzione restituisce un numero intero non negativo nel caso l'operazione si concluda con successo, *EOF* in caso di insuccesso.

LETTURA E SCRITTURA PER BLOCCHI [usata in genere per gli stream binari]

La lettura per blocchi consente di prelevare un numero di byte specificato e di inserirli nello spazio di memoria associato ad una data variabile. Il significato che i dati assumeranno sarà perciò determinato dal tipo della variabile. Questo meccanismo consente, ad esempio, di leggere facilmente dati di tipo *struct*.

size_t **fread**(void **ptr*, size_t *size*, size_t *nmemb*, FILE **stream*)

numero memory blocks

legge dallo stream *nmemb* blocchi di byte, ciascuno di dimensione *size*, e li memorizza a partire dall'indirizzo specificato da *ptr*.
Il tipo **size_t** è un tipo dipendente dallo specifico sistema operativo utilizzato, assimilabile al tipo int per i nostri scopi (per sistemi Linux, size_t corrisponde al tipo long unsigned int).
Il tipo di *ptr* è indicato come "puntatore a void" per significare che la funzione fread non tiene conto del tipo di *ptr*: si limita a copiare da file a memoria. Nell'utilizzo di fread *ptr* sarà un puntatore ad uno specifico tipo di variabile, in genere diverso da chiamata a chiamata della funzione.
fread restituisce il numero di blocchi effettivamente letti: se tale valore differisce da *nmemb* è possibile usare le funzioni feof ed ferror per capire quale problema si sia verificato.

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

scrive nello stream *stream nmemb* blocchi di byte, ciascuno di dimensione *size*, letti a partire dall'indirizzo specificato da *ptr*.
fread restituisce il numero di blocchi effettivamente scritti: se tale valore differisce da *nmemb* è

possibile usare le funzioni feof ed ferror per capire quale problema si sia verificato.

In genere fread ed fwrite sono usate nella forma

```
...dichiarazione del tipo di dato TipoDato...
FILE* Stream;
TipoDato Dato;
Stream = fopen("nomefile.dat", "rb+");
/* copia di un TipoDato da file a memoria RAM: */
fread(&Dato, sizeof(TipoDato), 1, Stream);
/* copia di un TipoDato da memoria RAM a file: */
fwrite(&Dato, sizeof(TipoDato), 1, Stream);
```

dove **`sizeof()`** è un operatore C da usare specificando, tra le parentesi tonde, il nome di un *tipo di dato* (attenzione: NON il nome di una variabile, `sizeof` non è una funzione!) e restituisce la dimensione del singolo dato di quel tipo. `sizeof` può essere applicato sia ai tipi di dato predefiniti che ai tipi di dato definiti dall'utente.
Nel caso vadano letti o scritti più blocchi di dati, ciascuno di essi corrisponda ad un elemento di un array e gli elementi dell'array da utilizzare siano consecutivi è possibile eseguire un'unica operazione di lettura o scrittura. Ad esempio:

```
TipoDato ArrayDati[5];
/* copia di 5 TipoDato nei 5 elementi dell'array: */
fread(ArrayDati, sizeof(TipoDato), 5, Stream);
/* copia degli elementi di indici 2...4 da array a file: */
fwrite(ArrayDati+2, sizeof(TipoDato), 3, Stream);
```

ACCESSO DIRETTO

Uno stream è visibile come una sequenza di byte posti uno dopo l'altro; in ogni istante la struttura FILE associata allo stream mantiene memoria della posizione corrente del "punto di accesso" al file, detto *indicatore di posizione*.
Ogni operazione di lettura o scrittura *sposta* automaticamente in avanti l'indicatore di posizione, in modo tale che se si accede al contenuto del file sequenzialmente non è richiesto al programmatore alcuno spostamento esplicito. Tuttavia esistono casi in cui occorre spostarsi all'interno di un file "saltando" ad un punto specifico[1]: è possibile usare la funzione fseek per compiere tale operazione.

int **fseek**(FILE **stream*, long *offset*, int *whence*)

posiziona l'indicatore di posizione *offset* bytes dopo la posizione specificata da *whence* nello stream *stream*. Quest'ultimo parametro assume i valori predefiniti SEEK_SET, SEEK_CUR, or SEEK_END a seconda che la posizione alla quale viene aggiunto l'offset sia l'inizio, la posizione corrente dell'indicatore di posizione o la fine del file.

1 Naturalmente non tutti i dispositivi fisici associati a stream ammettono la possibilità di saltare in un punto diverso dello stream: ad esempio è impossibile "tornare indietro nel tempo" e rileggere i caratteri prodotti nel passato da una tastiera.

fseek restituisce 0 in caso di corretta esecuzione dell'operazione, un valore diverso da 0 altrimenti. Nel primo caso il campo eof della struttura FILE associata a *stream* viene resettato.

void **rewind**(FILE **stream*)

equivale alla chiamata

```
fseek(stream, 0L, SEEK_SET)
```

(dove 0L rappresenta il valore 0 in formato long int).

long **ftell**(FILE **stream*)

restituisce il valore corrente dell'indicatore di posizione. Per file binari, questo valore corrisponde al numero di byte che precedono quello corrispondente all'indicatore di posizione; per file di testo il valore restituito può non essere significativo ma è comunque univoco (ovverosia può essere utilizzato con *fseek* per ritornare alla posizione attuale dopo uno spostamento dell'indicatore di posizione).

FUNZIONI AUSILIARIE

I file fisici interagiscono con i programmi attraverso **buffer**, ovvero zone di memoria utilizzate come "magazzino temporaneo" di dati. I dati vengono letti o scritti sul file *a blocchi* di dimensione prestabilita. Perciò, in particolare, durante un'operazione di scrittura su file viene effettivamente modificato il file fisico *solo quando il buffer è pieno*. Questo vuole dire che se il programma si blocca inaspettatamente (per un qualsiasi motivo, ad esempio a causa di un errore) quando il buffer non è vuoto, i dati nel buffer sono persi e *non* vengono scritti nel file (ad esempio a schermo, se il file è quello associato allo stream *stdout*). E' possibile forzare la scrittura dei dati dal buffer al file fisico con la funzione

int **fflush(FILE*** *stream***)**

La funzione restituisce 0 nel caso l'operazione si concluda con successo, **EOF** (costante intera definita in *stdio.h*) in caso di insuccesso.
fflush è utile in fase di *debugging* per assicurarsi che tutte le chiamate alla funzione *printf* che sono state eseguiteabbiano effettivamente portato ad un risultato a schermo. Infatti a causa del meccanismo di buffering può accadere che una scrittura a schermo ordinata con printf non venga eseguita immediatamente, e dunque non venga eseguita affatto se subito dopo la chiamata a printf il programma si interrompe o viene interrotto (ad esempio perché sta venendo eseguito in modalità step-by-step). Inserendo l'istruzione

```
fflush(stdout))
```

subito dopo una chiamata a printf, si è certi che -se la linea di codice corrispondente a tale istruzione viene superata durante l'esecuzione del programma- la printf abbia effettivamente generato il suo effetto su stdout. Se tale effetto non è visibile, non è possibile che ciò dipenda dal buffer, che è stato svuotato da `fflush`: la causa della mancata stampa a schermo va

cercata altrove.