Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Input e Output su file

I prototipi delle funzioni per la gestione dei file sono dichiarate in `stdio.h`.

2 tipologie di file:

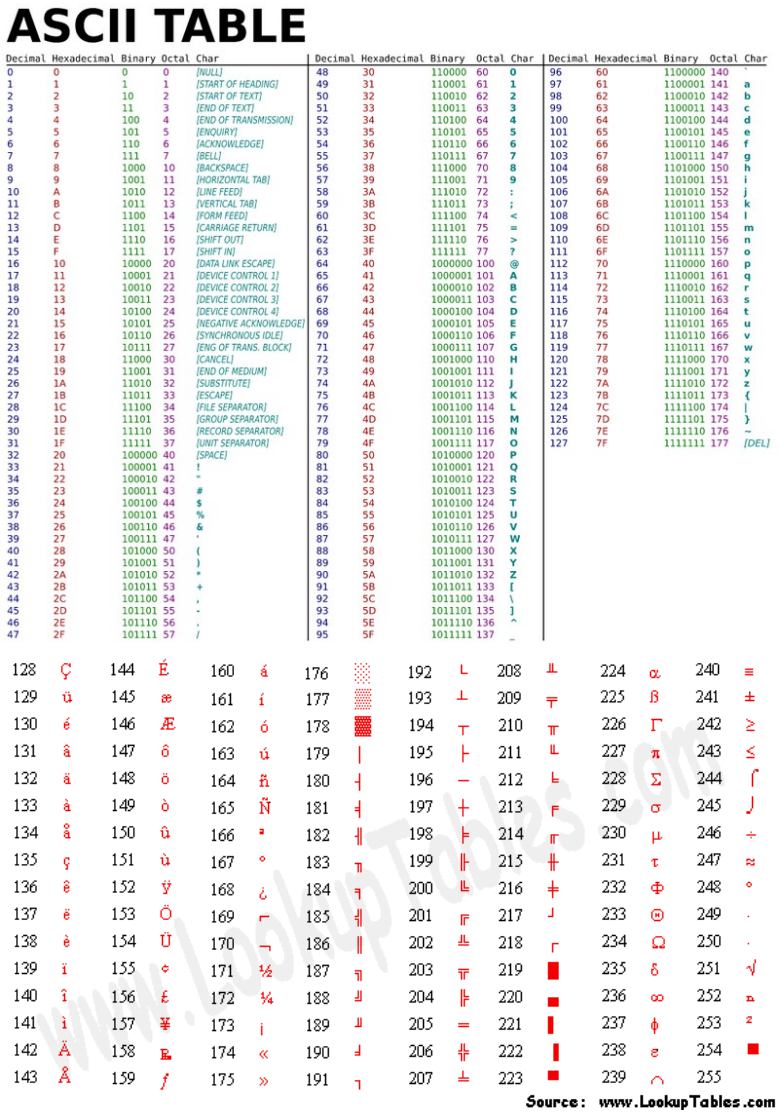
- file *ASCII* (o di testo)
- file *binari*

Esempio di creazione di un file **binario**:

In [3]:
```
#include <stdio.h>
int main()
{
    int v[] = {64, 77, 48, 92};
    FILE* fp;
    fp = fopen("vettore.bin","wb");

    fwrite(v, sizeof(int), 4, fp);

    fclose(fp);

    return 0;
}
```

`cat vettore.bin`

```
@M0
```



Da terminale:

`xxd -b vettore.bin`

```
00000000: 01000000 00000000 00000000 00000000 01001101
00000000  | @...M.
00000006: 00000000 00000000 00110000 00000000 00000000
00000000  | ..0...
0000000c: 01011100 00000000 00000000 00000000
| \...
```

- 00000000 00000000 00000000 01000000 è la codifica binaria di 64 su 4 byte
- 00000000 00000000 00000000 01001101 è la codifica binaria di 77 su 4 byte
- 00000000 00000000 00000000 00110000 è la codifica binaria di 48 su 4 byte
- 00000000 00000000 00000000 01011100 è la codifica binaria di 92 su 4 byte

Nota: si può notare che i byte in memoria sono rappresentati in ordine inverso (dal meno significativo al più significativo). Questo dipende da come l'architettura utilizzata memorizza le informazioni in memoria. Generalmente ci sono due modalità:

- Big-endian: memorizzazione/trasmissione che inizia dal byte più significativo per finire col meno significativo (generalmente usata nei processori Motorola);
- little endian: memorizzazione/trasmissione che inizia dal byte meno significativo per finire col più significativo (generalmente usata nei processori Intele AMD).

Esempio di creazione di un file di **testo**:

In [5]:
```
#include <stdio.h>
int main()
{
    int v[] = {64, 77, 48, 92};
    FILE* fp;
    fp = fopen("vettore.txt","w");

    for(int i=0; i<4; i++)
        fprintf(fp,"%d ", v[i]);

    fclose(fp);

    return 0;
}
```

Da terminale:

`cat vettore.txt`

```
64 77 48 92
```

`xxd -b vettore.txt`

```
00000000: 00110110 00110100 00100000 00110111 00110111
00100000  | 64 77
00000006: 00110100 00111000 00100000 00111001 00110010
00100000  | 48 92
```

- 00110110 è la codifica ASCII del 6
- 00110100 è la codifica ASCII del 4
- 00100000 è la codifica ASCII dello spazio

....

- In un **file di testo**, le cifre di ogni numero vengono convertite nei *caratteri* ASCII corrispondenti prima di essere scritte, così da poter essere facilmente lette da un essere umano attraverso un editor di testo
- In un **file binario**, ogni valore è scritto nella sua codifica originaria, senza che sia effettuata alcuna trasformazione

Apertura e chiusura di un file

Le funzioni per la gestione dei file sono generalmente dichiarate (o comunque incluse) in `stdio.h`.

In generale, per aprire e chiudere un file si usano rispettivamente le funzioni `fopen(...)` ed `fclose(...)`.

`fopen(...)`

La funzione `fopen(...)` ha il seguente prototipo:

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

Tale funzione accetta due parametri: il nome del file (stringa), e la *modalità di accesso* (stringa).

La modalità di accesso permette di stabilire due cose:

1. se vogliamo leggere/scrivere sul file
2. se vogliamo leggere o scrivere un file in modalità binaria o testuale

mode (text)	mode (binary)	descrizione
"r"	"rb"	**read:** Apre il file in lettura (input).Il file deve esistere, altrimenti restituisce NULL
"w"	"wb"	**write:** Crea un file in scrittura (output). Se il file già esiste, il suo contenuto viene eliminato.
"a"	"ab"	**append:** Crea un file in scrittura (output). Se il file già esiste, la scrittura partirà dalla fine del file. Repositioning operations (fseek, fsetpos, rewind) are ignored.
"r+"	"r+b"	**read/update:** Apre il file sia in lettura che scrittura. Il file deve esistere, altrimenti restituisce NULL.
"w+"	"w+b"	**write/update:** Crea un file per lettura/scrittura. Se il file già esiste, il contenuto viene eliminato.
"a+"	"a+b"	**append/update:** Crea un file in lettura/scrittura. Se il file già esiste, eventuali operazioni di scrittura verranno effettuate a partire dalla fine del file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file.

Tale funzione restituisce un puntatore di tipo `FILE`.

> `FILE` : [struct] that identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator

> and all its state indicators.

Tale struttura è definita (o comunque inclusa) in `stdio.h`.

Generalmente la struttura FILE di un dato file è chiamata anche *file handler*.

```
struct _IO_FILE
{
  int _flags;           /* High-order word is _IO_MAGIC; rest is
flags. */

  /* The following pointers correspond to the C++ streambuf
protocol. */
  char *_IO_read_ptr;   /* Current read pointer */
  char *_IO_read_end;   /* End of get area. */
  char *_IO_read_base;  /* Start of putback+get area. */
  char *_IO_write_base; /* Start of put area. */
  char *_IO_write_ptr;  /* Current put pointer. */
  char *_IO_write_end;  /* End of put area. */
  char *_IO_buf_base;   /* Start of reserve area. */
  char *_IO_buf_end;    /* End of reserve area. */

  /* The following fields are used to support backing up and
undo. */
  char *_IO_save_base; /* Pointer to start of non-current get
area. */
  char *_IO_backup_base;  /* Pointer to first valid character of
backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area.
*/

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  __off_t _old_offset; /* This used to be _offset but it's too
small.  */

  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

typedef struct _IO_FILE FILE;
```

I campi di questa struttura saranno utilizzati dalle funzioni di scrittura/lettura per accedere materialmente al file.

Tra tutte le informazioni contenute nella struct e che possono essere utili al programmatore, le più importanti sono:

- un cursore che tiene traccia fin dove il file è stato letto (nel caso di file aperto in modalità scrittura);
- un cursore che tiene traccia fin dove il file è stato scritto (nel caso di file aperto in modalità lettura);
- se il cursore di lettura ha raggiunto (o, meglio, *provato a superare*) la fine del file (condizione di `end-of-file`, se ne parlerà in dettaglio in seguito).

Tali cursori verranno quindi modificati dalle funzioni di lettura/scrittura su file. Esempio, man mano che si andrà avanti nella lettura attraverso l'utilizzo delle funzioni `fread(...)` o `fscanf(...)`, il cursore di lettura andrà avanti per tenere traccia di fin dove si è arrivati nella lettura.

fclose(...)

Una volta che il file non deve più essere utilizzato, è buona norma chiuderlo.

La chiusura di un file può essere effettuata attraverso la funzione `fclose(...)`

- `int fclose(FILE *stream)`

esempio:

```
// apro il file
FILE* file_handler = fopen(...);
// effettuo operazione sul file...

// terminate le operazioni sul file, chiudo il file
fclose(file_handler);
```

é buona norma chiudere **sempre** un file una volta finito di utilizzarlo. Sia per liberare risorse (e.g., la struct `FILE` occupa spazio in memoria), sia per permettere di finalizzare eventuali operazioni sospese (e.g., le operazioni di scrittura sul file non è detto che avvengano all'atto dell'invocazione della funzione di scrittura, a causa del meccanismo del buffering).

Se l'operazione va a buon fine, `fclose(...)` restituisce 0.

File Binari

Scrittura e lettura in un file binario

- `fwrite ( const void* ptr, size_t size, size_t nitems, FILE * stream );`

La funzione `fwrite(...)` scrive, sul file puntato da `stream`, (fino a) `nitems` elementi di dimensione `size` (in Byte), a partire dall'indirizzo specificato in `ptr`. tale funzione restituisce il numero di elementi correttamente scritti. Se tutto è andato bene, tale valore sarà pari a `nitems`.

Otherwise, if a write error occurs, the error indicator for the stream is set and errno is set to indicate the error.

- `fread(void *ptr, size_t size, size_t nitems, FILE *stream);`

La funzione `fread(...)` legge, dal file puntato da `stream`, (fino a) `nitems` di dimensione `size` (in Byte). La funzione restituisce il numero di elementi correttamente letti. Tale valore sarà `< nitems` solo se c'è un errore oppure il file è finito prima di riuscire a leggere `nitems` elementi.

In [ ]:

In [1]:
```
#include <stdio.h>
#define MAX_LEN 100

void stampa_vett(int v[], int n)
{
    printf("( ");
    for(int i=0; i<4; i++)
        printf("%d, ", v[i]);
    printf("\b\b)\n");

}

int main()
{
    int v[] = {64, 77, 48, 92};
    FILE* fp;
    fp = fopen("vettore.bin","wb");

    int n_scritti = fwrite(v, sizeof(int), 4, fp);

    fclose(fp);

    printf("scritto: ");
    stampa_vett(v, n_scritti);

    // lettura
    // supponiamo che non conosciamo quanti sono i valori da leggere,
    // quindi decidiamo di leggere un valore per volta
    fp = fopen("vettore.bin","rb");
    int v2[MAX_LEN];
    int n2 = 0;

    int readed = 1;
    while(readed >=1) // mentre l'ultima invocazione di fread mi ha fatto
    {
        readed = fread(&v2[n2], sizeof(int),1, fp);
        n2++;

    }
    fclose(fp);

    printf("letto: ");
    stampa_vett(v2, n2);


    return 0;
}
```

```
scritto: ( 64, 77, 48, 92)
letto: ( 64, 77, 48, 92)
```

L'esempio precedente non sfrutta appieno la funzione `fread(...)` in quanto:

- necessita di più invocazioni, una per ogni elemento da leggere dal file
- basandosi unicamente sull numero di valori letti dalla funzione `fread(...)` per terminare, non permette di avere nello stesso file più strutture diverse (e.g., più array) memorizzati nello stesso file

Possibile soluzione: Salvare all'interno del file *informazioni addizionali* in testa, ad esempio la lunghezza dell'array

In [2]:
```
#include <stdio.h>
#define MAX_LEN 100

void stampa_vett(int v[], int n)
{
    printf("( ");
    for(int i=0; i<4; i++)
        printf("%d, ", v[i]);
    printf("\b\b)\n");

}

int main()
{
    int v[] = {64, 77, 48, 92};
    int n   = 4;
    FILE* fp;
    fp = fopen("vettore.bin","wb");

    // scrivo il numero di elementi dell'array in testa al file
    fwrite(&n, sizeof(int), 1, fp);

    // provo a scrivere n*sizeof(int) Bytes nel file
    int n_scritti = fwrite(v, sizeof(int), n, fp);

    fclose(fp);

    printf("scritto: ");
    stampa_vett(v, n_scritti);

    // lettura
    // supponiamo non conosciamo quanti sono i valori da leggere,
    // ma sappiamo che tale quantità è contenuta in testa al file
    fp = fopen("vettore.bin","rb");
    int v2[MAX_LEN];
    int n2 = 0;
    // leggo il numero di elementi dell'array. Tale elemento so che si tr
    fread(&n2, sizeof(int), 1, fp);

    // leggo n2 elementi (ossia n2*sizeof(int) Bytes)
    fread(v2, sizeof(int), n2, fp);
```

```
    fclose(fp);

    printf("letto:   ");
    stampa_vett(v2, n2);



    return 0;
}
```

```
scritto: ( 64, 77, 48, 92)
letto:   ( 64, 77, 48, 92)
```

File di testo

Scrittura e lettura in un file di testo

Diverse funzioni possibili:

- `int fputc( int c, FILE *fp );`

Scrive il carattere con codifica ASCII `c` sul file gestito da `fp`. In caso di successo, restituisce il carattere scritto.

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

The function fputs() writes the string s to the output stream referenced by fp. It returns a non-negative value on success.

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

come `printf(...)`, ma su file di testo.

- `int fgetc( FILE * fp );`

legge un caratter dal file gestito da `fp` e ne restituisce la codifica ASCII.

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

legge al più $n-1$ caratteri dal file gestito da `fp`. Copia la stringa letta nell'area di memoria puntata da `buf`, aggiungendo un terminatore di stringa (ossia, dopo l'ultimo carattere letto, viene inserito nel buffer il carattere `'\0'`). Se nella stringa letta c'è un `\n` oppure il file termina prima di leggere $n-1$ caratteri, allora `buf` conterrà solo i caratteri letti fino a quel puno, incluso il `\n`. Se funzione viene completata con successo, allora il valore di ritorno sarà lo stesso di `buf` (ossia l'indirizzo dell'area di memoria in cui è stata scritta la stringa).

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

Come `scanf(...)`, ma legge dal file di testo gestito da `stream`. NB: ricordate però come la `scanf(...)` si comporta in presenza di spazi...

Supponiamo di avere un file di testo *stringa.txt* contenente:

```
ascii
ciao Pluto come stai?
Bene grazie
```

In [22]:
```
// esempio di utilizzo di fputs(...)

#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    int rit = fputs("ciao pluto", fp);
    printf("fputs: valore di ritorno: %d\n", rit);

    fclose(fp);

    // apertura in lettura
    // supponiamo che non sappia quanto sia lunga la stringa da leggere
    // ipotizzo che sia composta di al più 29 caratteri
    char str[30]; // 29 caratteri + terminatore
    fp = fopen("stringa.txt","r");

    // lettura
    char* str_rit  = fgets(str,30, fp);

    printf("fgets:  *str_rit:%s, *str:%s\n", str_rit, str);
    printf("fgets:  str_rit: %p,  str:%p\n", str_rit, str);
    printf("fgets:  valore di ritorno: %d\n", rit);

    fclose(fp);


    return 0;
}
```

```
fputs: valore di ritorno: 1
fgets:  *str_rit:ciao pluto, *str:ciao pluto
fgets:  str_rit: 0x7ffec8771370,  str:0x7ffec8771370
fgets:  valore di ritorno: 1
```

Lettura di un file già "terminato"

Cosa succede nel caso di tentativo di lettura di un file già completamente letto terminato (ossia in cui già è stato letto tutto e non c'è più nulla da leggere)?

Supponiamo di avere un file di testo *stringa.txt* contenente:

```
ascii
ciao Pluto come stai?
Bene grazie
```

Iniziamo con la funzione `fgets(...)`. Tale funzione, in caso tentativo di lettura da file terminato, restituisce `NULL`. Ma attenzione...

In [1]:
```
// attenzione...

#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    int rit = fprintf(fp, "ciao Pluto come stai?\nBene grazie");
    printf("fprintf: valore di ritorno: %d\n", rit);

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt","r");

    // lettura
    char str[30];
    // acquisisco con scanf
    rit  = fscanf(fp, "%s", str);

    printf("fscanf:  *str:%s; ", str); // come si vede,
                                       // la scanf avrà acquisito
                                       // da file fino al primo spazio
    printf(" valore di ritorno: %d\n", rit);

    // prima invocazione di fgets
    char* rit_str = fgets(str,30, fp); //fgets invece acqusirà fino al pr
    printf("(1) fgets:  *str:%s;", str);
    printf(" valore di ritorno: %p\n", rit_str);
    // seconda invocazione di fgets
    rit_str = fgets(str,30, fp);
    printf("(2) fgets:  *str:%s;", str);
    printf(" valore di ritorno: %p\n", rit_str);
    // terza invocazione di fgets
    rit_str = fgets(str,30, fp);
    printf("(3) fgets:  *str:%s;", str);
    printf(" valore di ritorno: %p\n", rit_str);
    fclose(fp);


    return 0;
}
```

```
fprintf: valore di ritorno: 33
fscanf:  *str:ciao;  valore di ritorno: 1
(1) fgets:  *str: Pluto come stai?
; valore di ritorno: 0x7ffebb4eaf70
(2) fgets:  *str:Bene grazie; valore di ritorno: 0x7ffebb4eaf70
(3) fgets:  *str:Bene grazie; valore di ritorno: (nil)
```

Ricapitolando: 0. la funzione `scanf` legge il file fino al primo spazio, acquisendo `ciao`.

1. prima invocazione di `fgets`: stringa contiene `Pluto come stai?\n`; valore di ritorno: indirizzo stringa
2. seconda invocazione di `fgets`: stringa contiene `bene grazie`; valore di ritorno: indirizzo stringa
3. terza invocazione di `fgets`: stringa contiene `bene grazie`; valore di ritorno: `NULL`.

- Intuitivamente, dato che il file `stringa.txt` contiene due righe (delimitate da un `\n`) e dato che ogni invocazione di `fgets(...)` prende una riga, mi sarei aspettato che il valore di ritorno della **seconda** invocazione di `fgets(...)` fosse `NULL`. Invece, ciò avviene soltanto dopo la ***terza*** invocazione.

- Questo perchè `fgets(...)` (e le funzioni di lettura in generale) impostano lo stato di fine-file soltanto quando materialmente viene fatta una lettura "a vuoto". Dato che le prime due letture vanno entrambe a buon fine (la prima per la prima frase, i.e. `Pluto come stai?\n`, e la seconda per la seconda frase, i.e. `Bene grazie`) non viene rilevato lo stato di fine-file.

- E' quindi necessaria una lettura ulteriore affinché `fgets(...)` fallisca la lettura dal file (in quanto il file è finito) e restituisca materialmente `NULL`. Da notare che il contenuto di `str` resta inalterato *rispetto alla lettura precedente*.

Non tenere conto di ciò potrebbe portare ad una implementazione ingenua come la seguente:

In [45]:
```
// Leggere un file di testo e stampare a video tutte le righe presenti (i

#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);
```

```
    // apertura in lettura

    fp = fopen("stringa.txt","r");
    char str[30];
    char* rit_str = str;
    while(rit_str != NULL)
    {

        rit_str = fgets(str,30, fp);
        printf("fgets: %s\n", str);
    }
    fclose(fp);


    return 0;
}
```

```
fgets: ciao Pluto come stai?

fgets: Bene grazie
fgets: Bene grazie
```

Il codice di sopra stampa due volte l'ultima riga. Questo perchè `rit_str` diventerà `NULL` solo quando `fgets(...)` non riuscirà a leggere materialmente nulla dal file in quanto terminato (lasciando quindi `str` inalterata), e ciò avverrà solo alla terza iterazione (e non alla seconda).

Possibile soluzione:

In [4]:
```
// Leggere un file di testo e stampare a video tutte le righe presenti (s
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt","r");
    char str[30];
    char* rit_str = str;
    while(rit_str != NULL)
    {

        rit_str    = fgets(str,30, fp);
        if(rit_str != NULL)
            printf("fgets: %s\n", str);
    }
    fclose(fp);


    return 0;
}
```

```
fgets: ciao Pluto come stai?

fgets: Bene grazie
```

Oppure...

In [5]:
```
// Leggere un file di testo e stampare a video tutte le righe presenti (s
#include <stdio.h>
int main()
{
    // apertura in scrittura
    FILE* fp = fopen("stringa.txt","w");
    if (fp == NULL)
    {
        printf("qualcosa è andato storto!\n");
        return 1;
    }

    // scrittura
    fprintf(fp, "ciao Pluto come stai?\nBene grazie");

    fclose(fp);

    // apertura in lettura

    fp = fopen("stringa.txt","r");
    char str[30];

    while(fgets(str,30, fp) != NULL) // controllo il valore di ritorno
                                     //all'interno della condizione ciclo
    {
        printf("fgets: %s\n", str);
    }
    fclose(fp);


    return 0;
}
```

```
fgets: ciao Pluto come stai?

fgets: Bene grazie
```

The curse of *End Of File*

Quando una funzione prova a leggere un file *terminato*, viene segnalata **internamente** (ad esempio attraverso un apposito flag nel file handler) una condizione di *end-of-file*. *A prescindere da questo*, ogni funzione adotta una politica differente nel caso in cui si provi ad accedere ad un file terminato. Ad esempio:

function	In caso di end-of-file (o errore) restituisce
fgets(...)	NULL

fscanf(...)	numero di conversioni effettuate minore di quanto atteso; In the case of an input failure before any data could be successfully read, EOF is returned.
fgetc(...)	EOF
fread(...)	numero di elementi letti minore di quanto atteso

`EOF` è una macro definita in `stdio.h` (o in qualche file `.h` da questultimo a sua volta incluso).

NB: `EOF` $\not\equiv$ `end-of-file` ;

- `EOF` è una macro definita in un header file forniti con la libreria standard (e.g., `#define EOF -1` in `stdio.h` o in file da questo inclusi). E' un valore che è restituito da *alcune* (e non tutte!) le funzioni della liberia standard quando si prova a leggere da un file terminato (o se accade qualche errore durante la lettura). Generalmente il suo valore è `-1`, o comunque un valore < 0.

- *end-of-file* è un possibile *stato* del file, generalmente identificato da uno (o più) flag nel file handler (ossia nella struct di tipo `FILE`).

  - Tale flag nasce per dire se il file è terminato oppure no (quindi, all'apertura del file, sarà impostato *false*).
  - Il valore di tale flag viene generalmente impostato come *true* quando le funzioni di accesso al file (e.g., `fread(...)`, `fgets(...)`, `fscanf(...)`, ecc.) provano a leggere qualcosa da un file *già terminato* in precedenza.
  - L'accesso in lettura a tale flag (ossia sapere quale valore contiene in un dato istante) è possibile in maniera semplice tramite la funzione `feof(...)`.

`int feof(FILE *stream)` :

tale funzione controlla se, *a seguito dell'ultimo accesso al file effettuato*, è stato impostato lo *stato end-of-file*.

In altri termini, `feof(fp)` restituisce un valore $\neq 0$ se l'indicatore *end-of-file* associato al file-handler `*fp` è impostato, altrimenti restituisce 0.

L'utilizzo di tale funzione è scongliato.

Esempio:

supponiamo di avere un file `testo.txt` contenente:

```
topolino
paperino
```

In [70]:
```
// lettura di un file di testo e stampa a video di ogni riga (versione er

#include <stdio.h>
int main()
{
    // apertura in lettura

    FILE* fp = fopen("testo.txt","r");
    char str[30];
    while(feof(fp) == 0)
    {

        fgets(str,sizeof(str), fp);
        printf("fgets: %s\n", str);
    }
    fclose(fp);


    return 0;
}
```

```
fgets: topolino

fgets: paperino

fgets: paperino
```

- 1° invocazione di `fgets(...)` : legge `topolino` (ma non imposta lo stato *end-of-file*)
- 2° invocazione di `fgets(...)` : legge `paperino` (ma non imposta lo stato *end-of-file*)
- 3° invocazione di `fgets(...)` : il file è terminato quindi non legge nulla, lascia `str` inalterato ed imposta lo stato di *end-of-file* all'interno del file handler

In [72]:
```
// lettura di un file di testo e stampa a video di ogni riga (versione co

#include <stdio.h>
int main()
{
    // apertura in lettura

    FILE* fp = fopen("testo.txt","r");
    char str[30];
    while(fgets(str,sizeof(str), fp)!= NULL)
    {

        printf("fgets: %s\n", str);
    }
    fclose(fp);


    return 0;
}
```

```
fgets: topolino

fgets: paperino
```

In [6]:
```
// lettura di un file di testo e stampa a video di ogni parola attraverso
// (versione corretta)

#include <stdio.h>
int main()
{
```

```
    // apertura in lettura

    FILE* fp = fopen("testo.txt","r");
    char str[30];
    while(fscanf(fp, "%s", str) > 0)
    {

        printf("fscanf: %s\n", str);
    }
    fclose(fp);

    return 0;
}
```

```
fscanf: topolino
fscanf: paperino
```

Consiglio: non utilizzare la funzione `feof(...)`, utilizzare invece in maniera corretta i valori di ritorno delle diverse funzioni di accesso ai file.

STDIN/STDOUT/STDERR

definite in `<stdio.h>`

- `FILE* stdin;`

ogni lettura da tastiera viene vista come una lettura dal file definito dalla struct (puntata da) stdin

- `FILE* stdout;`

ogni scrittura sul file definito dalla struct (puntata da) stdout viene vista come una scrittura sul monitor

- `FILE *stderr;`

associato tipicamente con il monitor ed è utilizzato per visualizzare messaggi di errore.

Alcune funzioni, ad esempio `getchar(...)` e `putchar(...)`, usano automaticamente `stdin` e `stdout`

`getchar(...)` $\equiv$ `getc(stdin,...)`

`printf(...)` $\equiv$ `fprintf(stdout,...)`

`scanf(...)` $\equiv$ `fscanf(stdin,...)`

per stampare un messaggio su `stderr`, si può usare la funzione

`void perror(const char *s);`

In [3]:
```
// Esempio di modifica dell' stdout (sconsigliato)...

#include <stdio.h>
int main()
{

    FILE* original_stdout = stdout;
    stdout = fopen("new_stdout.txt", "w");
    printf("ciao\n");
    fclose(stdout);
    stdout = original_stdout;
    printf("a tutti\n");
    return 0;
}
```

```
a tutti
```

Avvendo cambiato lo `stdout`, la parte iniziale del messaggio si troverà nel file `new_stdout.txt`.

Altre funzioni per l'accesso ai file

- `void rewind ( FILE * stream );` "riavvolge" il file all'inizio (lavora quindi sui cursori all'interno del file handler).

Altre funzioni per "muoversi" lungo il file:

- `long int ftell ( FILE * stream );`
- `int fseek ( FILE * stream, long int offset, int origin );`
- `int fsetpos ( FILE * stream, const fpos_t * pos );`
- `int fgetpos ( FILE * stream, fpos_t * pos );`