

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

Quando si utilizzano le system call per le operazioni di I/O ci sono due aspetti **prestazionali** da tenere in considerazione:

- Ad ogni chiamata si accede al kernel del S.O., causando un passaggio da user mode a kernel mode → overhead
- Nel caso di read e write si accede direttamente a un dispositivo fisico (per esempio, il disco) → tempo di accesso non trascurabile

Problema di dimensionamento del buffer

- Usando le system call per effettuare operazione di I/O, la gestione del buffer è demandata al programmatore

```
const int BUFF_LEN = 4096;  
char buff[BUFF_LEN];  
int read_bytes = read(fd, buff, BUFF_LEN);
```



Qual è la dimensione ottimale del buffer?

Inoltre, il programmatore si deve fare carico di **gestire il buffer** di lettura/scrittura (allocazione, riempimento, svuotamento, ecc.), compreso scegliere la sua dimensione.

Quale valore assegno a BUFF_LEN?

Problema del dimensionamento del buffer

```
#include ...
int main() {
    const int BUFF_LEN = 512;
    char buff[BUFF_LEN];
    int n;

    while((n = read(STDIN_FILENO, buff, BUFF_LEN)) > 0)    // legge stdin fino alla fine
    {
        if (write(STDOUT_FILENO, buff, n) != n)            // scrive i byte letti su stdout
        {
            /* gestione dell'errore */
        }
    }
    return 0;
}
```

Quale dimensione scegliere?

```
$ time ./my_cp < file
real    2m15.759s
user    1m13.646s
sys     1m1.766s
```

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

63

Nell'esempio, il programma esegue un ciclo in cui abbiamo una lettura (read) e una scrittura (write): per ogni ciclo abbiamo due accessi al kernel/disco.

Il numero di cicli è dato da: $n_{cicli} = D/BUFFLEN$, dove D è la dimensione del file e BUFFLEN è la dimensione del buffer.

- Buffer grande → minor numero di cicli, quindi di system call
- Buffer piccolo → minor occupazione di memoria (principale)

Con il comando **time** possiamo misurare la durata dell'esecuzione del programma:

- **real**: tempo totale dall'avvio alla terminazione del programma (include overhead e eventuali attese per cambio di contesto – per esempio quando lo scheduler assegna la CPU a un altro processo)
- **user**: tempo per eseguire istruzioni in user mode
- **sys**: tempo per eseguire istruzioni in kernel mode

Problema del dimensionamento del buffer

```
time ./my_cp < inputfile > /dev/null
```

- inputfile → 614,198,784 byte
- Output su /dev/null

BUFF_LEN	User time (secondi)	System time (secondi)	Real time (secondi)	Numero di loop
2	148.105	129.443	281.872	307099392
4	75.759	61.766	135.759	153549696
8	37.221	32.009	69.551	76774848
16	18.688	15.954	34.887	38387424
32	9.443	8.316	22.533	19193712
64	4.713	4.023	8.817	9596856
128	2.503	1.900	4.467	4798428
256	1.175	1.039	2.242	2399214
512	0.583	0.571	1.190	1199607
1024	0.272	0.331	0.619	599804
2048	0.160	0.184	0.354	299902
4096	0.089	0.113	0.210	149951
8192	0.036	0.100	0.138	74975
16384	0.016	0.089	0.112	37488
32768	0.012	0.078	0.091	18744
65536	0.000	0.082	0.082	9372
131072	0.000	0.079	0.079	4686
262144	0.000	0.087	0.089	2343
524288	0.000	0.079	0.086	1171
1048576	0.000	0.080	0.082	586

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

64

Risultati con diverse dimensioni del buffer

Raddoppiando il buffer si dimezzano il numero di cicli. Dimezzando il numero di cicli:

- il tempo utente si dimezza
- il tempo di sistema si dimezza solo fino a un certo punto, che dipende da come è implementato il file system sul disco (file divisi in blocchi di dimensione prefissata). Per vedere la dimensione del blocco usiamo il comando: `stat -c "%o" nomefile`

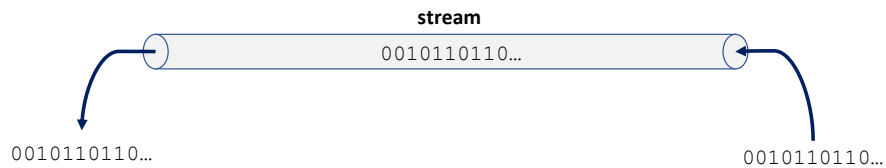
Funzioni di libreria per operazioni di I/O

- ISO C definisce una libreria per operazioni di input e output
 - standard I/O library
 - scritta da Dennis Ritchie nel 1975
- Non solo per sistemi Unix
- Gestisce in maniera ottimale i buffer
 - Minimizza il numero di system calls
- Non usa più i file descriptor, ma il concetto di **stream**
 - Quando si apre un file, gli viene associato uno stream invece che un file descriptor

La libreria mette a disposizione delle funzioni che gestiscono automaticamente il buffer di lettura/scrittura in modo da minimizzare il numero di system calls (quindi rendendo automaticamente le operazioni efficienti), e in modo che non sia più il programmatore a doversi occupare di questo compito.

Si usa uno stream come astrazione di un file descriptor.

Stream

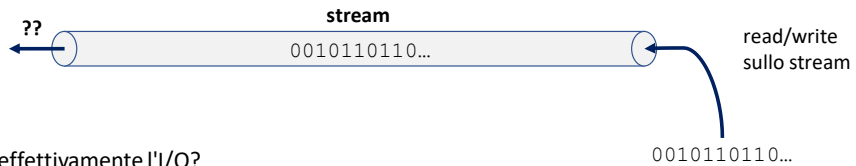


- Uno stream è un flusso di byte
- Associato a un oggetto di tipo **FILE***
 - Internamente al sistema, a un oggetto FILE corrisponde un file descriptor
- Tre stream predefiniti in `<stdio.h>`
 - `stdin`
 - `stdout`
 - `stderr`

Dal punto di vista del programmatore, lo stream è un punto di ingresso o uscita di byte verso o da il programma, e i dettagli implementativi dello stream (per esempio, il collegamento col file descriptor) sono nascosti al programmatore, sono gestiti dalle funzioni di libreria

Buffering

- Lo stream gestisce il buffering in modo da minimizzare il numero di `read()` / `write()`
- Un'operazione di lettura o scrittura inserisce dei byte nello stream



Quando viene fatto effettivamente I/O?

Dipende dal tipo di buffering!

- **Fully buffered**
- **Line buffered**
- **Unbuffered**

Quando si vuole scrivere su un file, inseriamo i dati nello stream, che li inserisce in un buffer. Solo in seguito, con i modi e i tempi che dipendono dal tipo di buffering, lo stream scrive i dati sul file invocando una o più system call. Dunque, le syscalls non spariscono, ma vengono eseguite dalle funzioni di libreria stdio.

Il buffer risiede in spazio utente, per cui l'operazione viene eseguita in user mode: solo quando lo stream decide di invocare la system call si passa in kernel mode.

Diversi stream possono avere tipo di buffering diverso.

Buffering

Stream **fully buffered**

- La scrittura viene fatta solo quando il buffer si riempie
- La lettura riempie sempre tutto il buffer
- Il buffer viene creato automaticamente con una `malloc()` quando si effettua la prima operazione sullo stream
- Tipicamente usato per I/O su file che risiedono sul disco
- Si può forzare il flushing invocando la funzione `fflush()`

Riduce al minimo il numero di syscalls, ma non sempre e' quello che vogliamo, es. shell interattiva

Buffering

Stream **line buffered**

- La scrittura viene fatta solo quando il buffer incontra il carattere ' \n ' (nuova linea)
- La lettura prende tutti i caratteri fino a incontrare il carattere ' \n ' (nuova linea)
- Permette di fare I/O di singoli caratteri, con un'unica syscall al termine della linea
- Usato per I/O su terminali
 - `stdin` e `stdout` sono tipicamente line buffered
- Attenzione: il buffer non ha dimensione infinita!
 - Se riempie prima di raggiungere la fine della linea, effettua il flush

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

69

Se eseguiamo una `printf` senza `'\n'` può succedere che l'output non venga mostrato subito sullo schermo

Buffering

Stream **unbuffered**

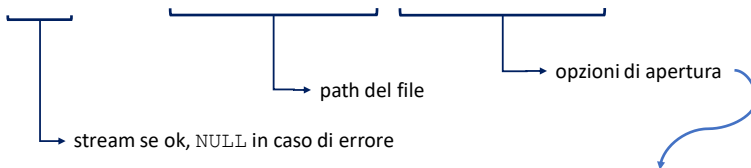
- L'I/O viene fatto per ogni operazione di input e output
 - Non usa buffer
 - Ogni lettura/scrittura corrisponde a una chiamata a una syscall
- Usato quando I/O deve essere effettuato il prima possibile
 - `stderr` è tipicamente unbuffered

Stesso numero di system call delle operazioni unbuffered, ma con la semplificazione che non serve gestire «manualmente» il buffer

Apertura di uno stream

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

```
FILE* fopen(const char* path, const char* type);
```



Type	Significato	Equivalenza con oflags
r , rb	Lettura	O_RDONLY
w , wb	Scrittura, con creazione del file e troncamento a 0	O_WRONLY O_CREAT O_TRUNC
a , ab	Scrittura in append	O_WRONLY O_CREAT O_APPEND
r+ , rb+ , r+b	Lettura e scrittura	O_RDWR
w+ , wb+ , w+b	Lettura e scrittura, con creazione del file e troncamento a 0	O_RDWR O_CREAT O_TRUNC
a+ , ab+ , a+b	Lettura e scrittura in append, con creazione del file	O_RDWR O_CREAT O_APPEND

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

71

Le modalità di apertura si specificano con una stringa, invece che con le costanti (come avveniva per la `open()`)

A differenza della `open`, non dobbiamo specificare esplicitamente un'opzione per creare un file. Se un file non esiste, viene creato automaticamente ogni volta che usiamo le opzioni `w`, `a`, `w+` o `a+`.

Non possiamo, inoltre, specificare i permessi esplicitamente – come avveniva con la `open()`.

In caso di errore, viene settata la variabile `errno`

Apertura di uno stream

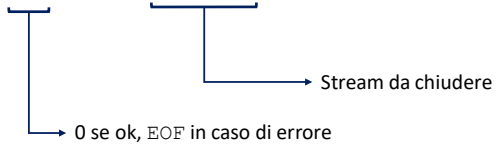
- Con la syscall `open()`, potevamo specificare i permessi nell'argomento `mode`, quando il flag `O_CREAT` era specificato
 - `int open(const char* path, int oflags, mode_t mode);`
 - Esempio:
 - `int fd = open("new_file", O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);`
- Con la libreria standard, un nuovo stream viene creato da `fopen()` specificando tipo `w` oppure `a`
- Permessi? → non si possono specificare!
- Default POSIX
 - `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH`

Anche in questo caso, dobbiamo tenere conto di `umask`, che definisce quali permessi non si possono assegnare.

Chiusura di uno stream

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

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



- Se ci sono dati di input nel buffer, vengono scartati
- Se ci sono dati di output nel buffer, vengono mandati in output (flush)
- Libera la memoria del buffer eventualmente allocata

Come per i file descriptor, anche gli stream vengono chiusi automaticamente al termine del processo.

Flush di uno stream

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

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

→ 0 se ok, EOF in caso di errore

→ Stream su cui effettuare il flushing

- Qualunque dato di output nel buffer viene mandato in output
 - Viene invocata la syscall `write()`
- Caso particolare
 - `fflush(NULL);`
 - Effettua il flush di tutti gli stream di output aperti

Si usa solo con gli stream di output

I/O non formattato

Tre modi di leggere da uno stream/scrivere su uno stream dati *non formattati*:

I/O di un carattere alla volta

- Funzioni che leggono dallo stream/scrivono sullo stream un singolo carattere (1 byte)

I/O di una linea alla volta

- Funzioni che leggono dallo stream/scrivono sullo stream una linea
- Ogni linea è una sequenza di caratteri terminata da ' \n '

I/O binario

- I/O diretto
- Funzioni che leggono dallo stream/scrivono sullo stream una serie di oggetti
 - Array, strutture, ecc.
- Tipicamente usato per file binari (che non devono essere human-readable)

Nota bene: ci si riferisce a lettura/scrittura da/verso uno stream, **non** da/verso un file. La lettura/scrittura vera e propria dipende dal tipo di buffering.

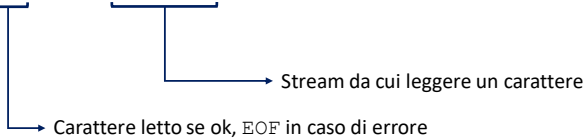
Ad esempio, il programma scrive un carattere all'interno dello stream, mentre lo stream è fully-buffered, perciò scrive sul file tutto il contenuto del buffer (tutti i caratteri insieme) quando quest'ultimo è pieno.

Lettura per carattere

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

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

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



- Non c'è differenza tra le due funzioni
- Restituiscono un `unsigned char`, convertito in `int`
 - `int` necessario per poter restituire il codice di errore
- Che succede se la funzione restituisce EOF (tipicamente -1) e assegno tale valore a una variabile `char`?

Le due funzioni sono equivalenti dal punto di vista funzionale, anche se la loro implementazione all'interno del sistema operativo può essere diversa.

Le funzioni restituiscono un intero per permettere di rappresentare EOF, che è implementato come un numero negativo (non esisterebbe un `char` «negativo»)

Le funzioni prelevano un carattere **dallo stream**. Se nel buffer dello stream non è presente alcun carattere, allora la funzione esegue una `system call read()` per prelevare dei caratteri dal file e metterli nello stream. Il numero di byte prelevati dal file verso lo stream dipende dal tipo di buffering.

Lettura per carattere

- `getc()` e `fgetc()` restituiscono EOF sia se si è verificato un errore, sia se si è raggiunta la fine del file
- Come si distinguono i casi?
- L'oggetto `FILE*` contiene due flag
 - *error flag*
 - *end-of-file flag*

Funzioni per testare i flag:

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

Restituisce un valore > 0 (true) se c'è stato un errore, 0 (false) altrimenti

Restituisce un valore > 0 (true) se si è raggiunta la fine del file, 0 (false) altrimenti

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

← Azzera i flag

Se la funzione restituisce EOF, non posso sapere a priori se c'è stato un errore o se ho semplicemente raggiunto la fine del file. Per capirlo, devo testare i due flag.

Lettura per carattere

- Gestione dell'errore:

```
...  
FILE* fp;  
if (fp = fopen("path/to/file", "r") < 0) {  
    fprintf(stderr, "Error while opening the file\n");  
    exit(EXIT_FAILURE);  
}  
  
int read_char;  
if ((read_char = fgetc(fp)) == EOF) {  
    if (feof(fp) > 0)  
        fprintf(stderr, "End of file reached\n");  
    else  
        fprintf(stderr, "Error while reading a character\n");  
    exit(EXIT_FAILURE);  
}  
char new_char = read_char;  
...
```

Lettura per carattere

- Caso particolare: lettura da standard input

```
int new_char = fgetc(stdin);
```

- Equivalente a:

```
int new_char = getchar();
```

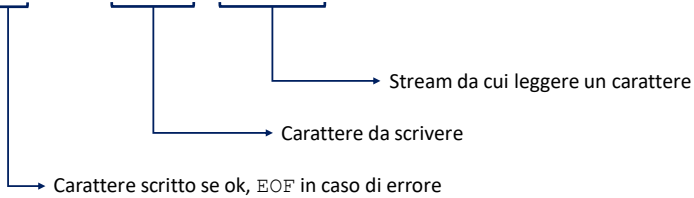
- Per il valore di ritorno, valgono le stesse considerazioni fatte per `getc()` e `fgetc()`

Scrittura per carattere

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

```
int putc (int ch, FILE* fp);
```

```
int fputc(int ch, FILE* fp);
```



- Stesse considerazioni per le analoghe funzioni di lettura per carattere

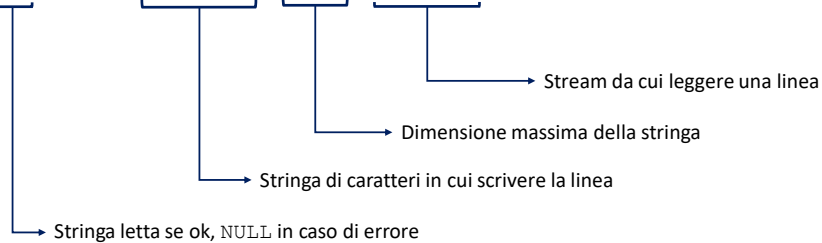
```
int written_ch = fputc(ch, stdout);    equivale a    int written_ch = putchar(ch);
```

Scrittura di un carattere **sullo stream**. L'output sul file vero e proprio dipende dal tipo di buffering.

Lettura per linea

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

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



- Legge il contenuto dello stream fino a incontrare '`\n`' e lo inserisce in `str`
 - Dopo '`\n`', viene sempre inserito un carattere nullo '`\0`'
- Se la linea è più lunga di `n-1`, viene spezzata
 - L'ultimo carattere di `str` è comunque '`\0`'

La funzione può restituire una stringa più corta di n caratteri, nel caso in cui si incontri prima il carattere `\n`.

La lunghezza effettiva della stringa si può ottenere usando la funzione `strlen()`

Lettura per linea

```
#include <stdio.h>

char* gets(char* str);
```

- Legge il contenuto dello stream di ingresso `stdin` fino a incontrare '`\n`' e lo inserisce in `str`
- Attenzione! Questa funzione **non** dovrebbe essere usata!
- Due buoni motivi:
 - Non richiede la dimensione massima della stringa → *buffer overflow*
 - Non inserisce '`\0`' alla fine della riga

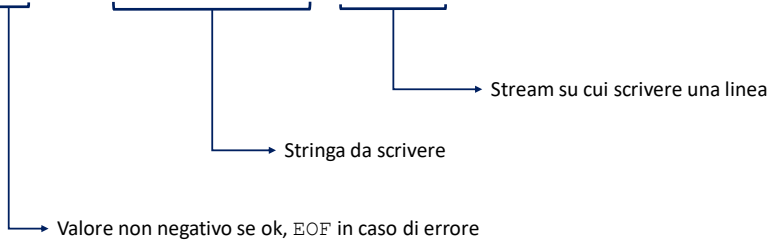
Si ha buffer overflow se `str` è un array di `N` caratteri, ma la linea che si legge dallo stream è lunga `M` caratteri, con $M > N$. In questo caso si vanno a occupare le locazioni di memoria contigue all'array `str`, eventualmente sovrascrivendo degli altri dati (o scrivendo in zone vietate della memoria, generando un «segmentation fault»)

Il carattere `\0` deve essere aggiunto manualmente. Se non viene fatto, non si possono usare le funzioni sulle stringhe, come `strlen()`.

Scrittura per linea

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

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



- La stringa `str` deve essere terminata dal carattere nullo `'\0'`
 - Il carattere `'\0'` non viene scritto sullo stream
 - Non necessariamente contiene `'\n'`

Notare che non serve specificare la lunghezza della stringa, perché la funzione si interrompe da sola quando incontra `\0`

Scrittura per linea

```
#include <stdio.h>

int puts(char* str);
```

- Scrive il contenuto della stringa `str` sullo stream di uscita `stdout`
- Non è «pericolosa» come `gets()`, ma...
 - A differenza di `fputs()`, aggiunge un `'\n'`

Efficienza dello standard I/O

- Leggere tutto il contenuto di un file e stamparlo a video

I/O di un carattere alla volta (*fgetc/fputc*)

```
int main() {
    char ch;
    while((n = fgetc(stdin)) != EOF) {
        ch = n;
        if (fputc(ch, stdout) == EOF) {
            /* error */
        }
    }
    return 0;
}
```

I/O di una linea alla volta (*fgets/fputs*)

```
int main() {
    const int MAX_LINE = 1024;
    char str[MAX_LINE];
    while (fgets(str, MAX_LINE, stdin) != NULL) {
        if (fputs(str, stdout) == EOF) {
            /* error */
        }
    }
    return 0;
}
```

I due programmi sono funzionalmente equivalenti, ma possono presentare prestazioni (tempi di esecuzione) diverse.

Efficienza dello standard I/O

```
time ./my_cp_fgetc < inputfile > /dev/null
time ./my_cp_fgets < inputfile > /dev/null
```

- inputfile → 614,198,784 byte
- Output su /dev/null

Metodo	User time (secondi)	System time (secondi)	Real time (secondi)
fgetc, fputc	3.360	0.219	4.168
fgets, fputs	0.590	0.152	1.465
Unbuffered (BUFF_LEN = 2 byte)	148.105	129.443	281.872
Unbuffered (BUFF_LEN = 32768 byte)	0.012	0.078	0.091

Standard I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

86

Lo user time è nettamente superiore nel caso di lettura/scrittura per carattere rispetto alla lettura/scrittura per linea, perché eseguiamo più iterazioni del ciclo while.

Nonostante questo, il system time non presenta differenze significative. Ciò accade perché il numero di read/write è indipendente dal tipo di funzione usata, ma dipende solo dal tipo di buffering (lo stesso nei due casi)

Benché il caso unbuffered con buffer grande sia il migliore dal punto di vista dei tempi di esecuzione (seppure di poco), ciò avviene al prezzo di maggiore complessità nell'implementazione (dobbiamo gestire manualmente i buffer).

Quando si sceglie una implementazione, bisogna tenere presente sia le prestazioni che la facilità di implementazione.

Output formattato

```
#include <stdio.h>

int printf (const char* format, ...);
int fprintf (FILE* fp, const char* format, ...);
int dprintf (int fd, const char* format, ...);
int sprintf (char* buf, const char* format, ...);
int snprintf(char* buf, size_t n, const char* format, ...);
```


- Differiscono per la destinazione dell'operazione di scrittura
 - `printf()` scrive su `stdout`,
 - vedi primo argomento per le altre funzioni
- `snprintf()` differisce da `sprintf()` in quanto specifica la dimensione del buffer
 - Evita buffer overflow (ricordate?)
- L'argomento `format` è una stringa contenente degli specificatori di conversione
 - Indicano come i rimanenti argomenti (e.g. variabili) saranno interpretati
- Restituiscono il numero di byte scritti

Le funzioni per l'I/O formattato servono a fare input e output tenendo conto del tipo dei dati che vogliamo leggere/scrivere, grazie agli specificatori di conversione ("%d" serve a specificare un numero intero, che verrà convertito in stringa).

Le varie funzioni si differiscono per la destinazione della stringa da scrivere. Per esempio, la `sprintf()` può essere sfruttata per convertire un numero intero in una stringa

Output formattato

```
int written_bytes = fprintf(fp, "String: %s, number: %d \n", str, n);
```



Conversion type	Description
d, i	signed decimal
o	unsigned octal
u	unsigned decimal
x, X	unsigned hexadecimal
f, F	double floating-point number
e, E	double floating-point number in exponential format
g, G	interpreted as f, F, e, or E, depending on value converted
a, A	double floating-point number in hexadecimal exponential format
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters written so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

Input formattato

```
#include <stdio.h>

int scanf(const char* format, ...);
int fscanf(FILE* fp, const char* format, ...);
int sscanf(char* buf, const char* format, ...);
```

- Duali delle operazioni di scrittura
- Restituiscono il numero di oggetti letti
 - Differente dalle operazioni di scrittura

Anche qui le diverse versioni si differenziano per il tipo della sorgente dei dati. La `sscanf` può essere sfruttata per separare le parole di una stringa, o per convertire una stringa in un numero intero.

Input formattato

```
int read_obj = fscanf(fp, "%c %f", &ch, &fn);
```



Conversion type	Description
d	signed decimal, base 10
i	signed decimal, base determined by format of input
o	unsigned octal (input optionally signed)
u	unsigned decimal, base 10 (input optionally signed)
x, X	unsigned hexadecimal (input optionally signed)
a, A, e, E, f, F, g, G	floating-point number
c	character (with 1 length modifier, wide character)
s	string (with 1 length modifier, wide character string)
[matches a sequence of listed characters, ending with]
[^	matches all characters except the ones listed, ending with]
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters read so far
%	a % character
C	wide character (XSI option, equivalent to 1c)
S	wide character string (XSI option, equivalent to 1s)

Posizionamento nello stream

```
#include <stdio.h>

int fseek (FILE* fp, long offset, int whence);
int fseeko (FILE* fp, off_t offset, int whence);
int fsetpos(FILE* fp, const fpos_t* pos);

long ftell (FILE* fp);
off_t ftello(FILE* fp);
int fgetpos(FILE* fp, fpos_t* pos);
```

Diagram annotations:

- `long offset` in `fseek` points to **nuova posizione (relativa)**.
- `fp` in `fseek` points to **stream**.
- `const fpos_t* pos` in `fsetpos` points to **nuova posizione (assoluta)**.
- `fp` in `ftell` points to **posizione corrente**.
- `fp` in `fgetpos` points to **posizione corrente**.
- `pos` in `fgetpos` points to **posizione corrente**.
- `0 se ok, !=0 se errore` is annotated for the return value of `fseek`, `fseeko`, `fsetpos`, `ftell`, and `fgetpos`.

- `fseek()`, `fseeko()` e `fsetpos()` non restituiscono la nuova posizione
 - A differenza della system call `lseek()`

Posizionamento nello stream

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

```
int main ()
```

```
{
```

```
    FILE *fp;
```

```
    fpos_t position;
```

```
    fp = fopen("newfile", "w+");
```

```
    fgetpos(fp, &position);
```

```
    fputs("Hello, World!", fp);
```

```
    fsetpos(fp, &position);
```

```
    fputs("Questa stringa sovrascivera' il contenuto del file", fp);
```

```
    fclose(fp);
```

```
    return EXIT_SUCCESS;
```

```
}
```

Ottiene la posizione corrente e la
salva dentro la variabile `position`

Riposiziona lo stream nella posizione salvata
precedente nella variabile `position`