



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - File

Ultimo aggiornamento: 23/11/2020

L'input/output

- Un importante argomento rimasto da affrontare è quello dell'*input/output*.
- Un programma C può comunicare con l'«esterno» in un solo modo, leggendo e scrivendo dati su **FILE**.
- Per quanto possa sembrare strano e scomodo è l'unico modo a disposizione, limitandosi allo standard.
- Con la parola «file» in questo caso non si intendono solamente quelli che normalmente vengono chiamati file, come i file di testo, i file jpeg, mp3, eccetera, ma si intende un'astrazione che rappresenta ogni possibile oggetto verso il quale un programma C può leggere o scrivere dati.
- Il file è una metafora per una sequenza di byte dalla quale è possibile leggere o scrivere (o entrambe le cose).
- Anche oggetti che all'apparenza non sono dei file, come ad esempio la tastiera e il monitor, vengono visti da un programma C come se lo fossero!
- Come questo venga realizzato in pratica lo vedremo più avanti.

Cos'è un file

- Quando si parla di «file» in C si parla in realtà di un flusso di byte o in inglese *stream*.
- Uno stream è un'astrazione che permette di accedere in modo *uniforme* a tutti i possibili tipi di file supportati dal sistema operativo.
- Avere un modo uniforme per accedere a qualsiasi tipo di file ci permette di usare le stesse funzioni sia per leggere da un file di testo che dalla tastiera o da un qualsiasi altro tipo di file.
- Il programma non dovrà preoccuparsi dei dettagli di ciascun dispositivo, di questo si occuperà il sistema operativo.
- Uno stream può essere immaginato come un flusso di dati ai quali si accede *sequenzialmente* byte per byte.
- Una volta *aperto un file* (vedremo tra poco cosa significa) normalmente l'accesso avviene a partire dal primo byte di questo stream e procede in avanti passando per tutti i byte successivi.

La libreria stdio.h

- In C non esistono keyword o operatori per gestire l'input/output.
- Per fortuna tutte le funzioni e i tipi necessari per questo sono disponibili però in una libreria, la libreria **stdio.h**.
- Questa libreria definisce un tipo di dato FILE i cui dettagli sono nascosti al programmatore.
- Non ci interessa sapere che cos'è esattamente il tipo di dato FILE (struct, int, o altro) perché sarà solo un mezzo per far riferimento dall'interno del programma ad un vero file che viene identificato tramite il suo nome.
- I nomi dei file sono specifici per ogni sistema operativo.
- La libreria stdio.h contiene inoltre tutte le funzioni per creare FILE, per utilizzarli per leggere o scrivere.
- Infine definirà alcune macro utili per la gestione dei diversi «casi» in cui ci si potrà imbattere.

Il tipo FILE

- Per rappresentare un file in C viene usato uno specifico tipo, il tipo `FILE`.
- Un `FILE` è un typedef di un tipo che contiene le informazioni necessarie per mantenere lo stato dello stream.
- Durante la programmazione non si accederà **mai** direttamente alla variabile di tipo `FILE`.
- Per effettuare una qualsiasi operazione su un file si utilizzano *sempre* le funzioni di `stdio.h`.
- Un `FILE` può essere usato solamente attraverso un puntatore, tutte le funzioni di `stdio.h` infatti hanno come parametro un puntatore a `FILE`.
- È ovviamente possibile dereferenziare un puntatore a `FILE` per ottenerne una copia locale ma non ci servirebbe a nulla e il suo utilizzo in una delle funzioni di `stdio.h` produrrebbe un undefined behavior!

Modalità di accesso ai file

- Quando si lavora sui file l'accesso agli stessi viene effettuato in maniera sequenziale.
- Questo avviene come se tutti i byte dello stream venissero analizzati da un «cursore» che si sposta in avanti byte per byte.
- Il controllo di questo cursore è affidato al programmatore. Ogni funzione che esegue qualche tipo di operazione su uno stream modifica la posizione del cursore di un certo numero di byte. Esistono inoltre alcune funzioni il cui scopo è solamente quello di spostare il cursore.
- L'accesso (in lettura) al file prosegue spostando in avanti il cursore fino a che una lettura non fallisce perché ha tentato di leggere oltre la fine dello stream. In quel caso si dice che si è raggiunto l'*End-of-file*.
- Alcune funzioni della libreria `stdio.h` sfruttano una macro per indicare quando si è raggiunto l'end-of-file. Questa macro si chiama `EOF`.
- `EOF` viene sostituita da un intero, solitamente -1. Vedremo che sfrutteremo spesso questa macro per capire quando verrà raggiunta la fine di un file o si verificheranno altri errori.

Modalità di accesso ai file

- È importante notare che alla fine di un file non sono fisicamente presenti uno o più byte con un valore particolare per indicarne la fine.
- Il sistema operativo conosce la dimensione di un file e quindi riesce a segnalare la fine di un file.
- Il valore EOF viene utilizzato dalle funzioni della libreria standard del C per indicare al programma che la lettura è fallita, ma non è un valore realmente presente alla fine del file.

Modalità di accesso ai file

- Nel momento in cui si apre un file in un programma C bisogna decidere quale sarà la modalità di accesso al file.
- Le modalità di accesso si differenziano in base a quali operazioni sono permesse sul file stesso e alla posizione in cui avvengono le scritture.
- Si può decidere di aprire il file in modalità testuale o binaria e si può decidere se aprire il file in lettura, scrittura o append.
- Vediamo ora come si differenziano le varie modalità di accesso:

	Operazioni permesse	Punto di accesso
Read	Solo letture	Inizio del file
Write	Solo scritture	Inizio del file
Append	Solo scritture	Fine del file

Modalità di accesso ai file

- Nel caso fosse necessario è possibile aprire un file in modo da poter effettuare sia operazioni di lettura che di scrittura.
- Se una volta aperto un file dovessimo per sbaglio effettuare delle operazioni non permesse dalla modalità con cui è stato aperto scopriremmo che tutte le funzioni che realizzano queste operazioni fallirebbero.

Modalità di accesso ai file

- Esistono altre due modalità di apertura di un file che possono essere combinate con quelle già viste in precedenza:
 - Modalità «**testuale**» o «**tradotta**».
 - Modalità «**binaria**» o «**non tradotta**».
- Sia chiaro, NON esistono file di testo e file binari. Esiste un solo tipo di file, quello che contiene una sequenza di byte.
- Cosa cambia quindi tra file aperti in modalità testuale e file aperti in modalità binaria?
- L'unica differenza riguarda il modo in cui viene trattata la rappresentazione del carattere «a capo». Essa infatti può variare a seconda del sistema operativo che si sta utilizzando, ad esempio:
 - Windows usa due caratteri: 13 10 («\r» «\n»).
 - I sistemi della famiglia Unix utilizzano solamente il carattere «\n».
 - I sistemi operativi MacOS fino alla versione 9 usavano solamente il carattere «\r», successivamente sono passati al «\n».

Modalità di accesso ai file

- La modalità tradotta o testuale che il C mette a disposizione vuole offrire al programmatore un modo uniforme di trattare i file di testo: è un'altra astrazione!
- Il C usa il modo più semplice possibile per rappresentare gli a capo, ossia quello dei sistemi Unix che utilizza un solo carattere (il 10 o '\n') per indicare la fine di una linea.
- In questo modo, un programma in C potrà essere ricompilato su più sistemi operativi senza preoccuparsi di dover gestire i caratteri di fine linea diversamente per ognuno di essi.

Modalità di accesso ai file

- Come avverrà quindi il processo di traduzione degli a capo?
- Dal punto di vista del programmatore non è importante sapere come questo avviene, l'importante è sapere che saranno la libreria standard del C e le sue funzioni ad occuparsene.
- È importante notare che **ogni volta** che si tenta di scrivere un byte che ha il valore 10 ('`\n`' o 0x0A in esadecimale), se il file è stato aperto in modalità tradotta, esso viene trasformato nella rappresentazione dell'«a capo» usata dal sistema operativo in uso.
- Non è importante ciò che si sta cercando di scrivere su file, ad esempio non c'è differenza tra array di interi e stringhe, si tratterà comunque di una sequenza di byte. Se uno di essi dovesse avere il valore 0x0A allora questo verrà tradotto.

Modalità di accesso ai file

- Cosa si intende quindi colloquialmente con il termine «file di testo»?
- Si parla di «file di testo» nel caso di un file che dovrebbe contenere solo caratteri stampabili, come lettere maiuscole e minuscole, numeri, punteggiatura, eccetera e il carattere di controllo «tab orizzontale» 9 o '\t' e caratteri di «a capo».
- La sola presenza di caratteri stampabili li rende facilmente leggibili dagli esseri umani attraverso l'uso di editor di testo.
- Nonostante queste caratteristiche rimangono comunque anch'essi sequenze di byte! È ovviamente possibile scrivere su un file di testo anche valori di byte che non rappresentano caratteri stampabili, ottenendo risultati inaspettati.

```

1  \x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
2  \x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
3  \x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
4  \x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
5  \x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
6  \xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
7  \x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
8  \x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
9  \x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
10 \x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
11 \xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
12 \x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
13 \x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
14 \x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
15 \x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
16 \xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
17 \x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
18 \x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
19 \x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
20 \x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
21 \xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
22 \x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
23 \x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
24 \x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f

```

Apertura di un file

- Vediamo ora come si può aprire un file in un programma C.
- La funzione da usare si chiama **fopen()**:

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

- La funzione ha due parametri, `filename` è una stringa che contiene il nome del file da aprire, `mode` è un'altra stringa che indica la modalità in cui aprirlo.
- Questa funzione ritorna un puntatore a un tipo `FILE`. Questo puntatore è quello che utilizzeremo ogni volta che avremo bisogno di eseguire una qualsiasi operazione sul file appena aperto.
- Se durante l'apertura del file si verifica un errore il puntatore ritornato è `NULL`.

Apertura di un file

- La stringa che passiamo alla `fopen()` come primo parametro contiene il nome del file da aprire.
- Il nome di un file su Windows può essere indicato specificando il suo *percorso*, in inglese *path*.
- Questo può essere un percorso *relativo* o *assoluto*.
- Quando utilizziamo un percorso assoluto specifichiamo sempre il disco in cui si trova (ad esempio «C:\») e poi tutto il percorso necessario per raggiungere il file.
- Un esempio di percorso assoluto è:
«C:\Users\77641\Desktop\file.txt»
- Quando invece specifichiamo un percorso relativo questo viene inteso rispetto alla posizione da cui eseguiamo il programma (la cartella che contiene il progetto in Visual Studio).
- Quando lavoravamo con ADE8 ad esempio eseguivamo l'assemblatore passandogli un percorso relativo a un file su cui lavorare. In quel caso specificavamo solamente il nome del file perché il programma `a8a.exe` e il file da passare all'assemblatore erano già nella stessa cartella.

Apertura di un file

- Il secondo parametro della `fopen()` è un'altra stringa che serve ad indicare la modalità in cui aprire il file.
- La stringa deve essere formattata in modo particolare per permettere alla `fopen()` di capire la modalità.
- Le possibili stringhe che possono essere utilizzate sono le seguenti:

"r"	Apri il file in lettura in modalità tradotta (testo). Il file deve esistere.
"w"	Apri il file in scrittura in modalità tradotta (testo). Se il file non esiste, viene creato un file vuoto con il nome specificato, altrimenti esso viene sovrascritto da quello nuovo vuoto.
"a"	Apri il file in append in modalità tradotta (testo), cioè in scrittura posizionandosi alla fine del file. Se il file non esiste allora viene creato.
"rb"	Apri il file in lettura in modalità non tradotta (binaria).
"wb"	Apri il file in scrittura in modalità non tradotta (binaria).
"ab"	Apri il file in append in modalità non tradotta (binaria).

Apertura di un file

- I possibili errori che potrebbero portare ad avere un puntatore `NULL` come risultato di una `fopen()` potrebbero essere i seguenti:
 - Aprire in lettura un file che non esiste.
 - Aprire in scrittura un file su un DVD-ROM.
 - Aprire un file per cui non si dispone dei diritti necessari.
- Esistono anche le modalità di lettura/scrittura seguenti:

"r+"/"r+b"	Apri il file per lettura/scrittura. Il file deve esistere.
"w+"/"w+b"	Apri il file per lettura/scrittura. Se il file non esiste, viene creato un file vuoto con il nome specificato, altrimenti esso viene sovrascritto da quello nuovo vuoto.
"a+"/"a+b"	Apri il file per lettura/scrittura posizionandosi alla fine del file. Se il file non esiste allora viene creato.

Chiusura di un file

- Quando un file non serve più al programma questo va chiuso sfruttando la funzione **fclose()**:

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

- La funzione prende come unico parametro il puntatore al file da chiudere e ritorna un intero.
- Il valore di ritorno indica se l'operazione ha avuto successo o meno. In caso di successo ritorna uno 0, altrimenti ritorna `EOF` se si è verificato qualche problema.
- È importante che ogni funzione che gestisce un puntatore a `FILE` lo chiuda, per evitare di trovarsi con un numero di file aperti superiore al limite imposto dal sistema operativo.
- Ad esempio il limite di file aperti contemporaneamente da un processo per Linux è di default 1024, mentre per Windows è 2048.

Apertura e chiusura

- Ecco un primo esempio di uso di `fopen()` e `fclose()`.
- Per ora diamo per scontato che nella stessa cartella in cui eseguiamo il programma esista un file chiamato «test.txt» che vogliamo aprire.

```
#include <stdio.h>

int main(void)
{
    FILE *f = fopen("test.txt", "r");
    if (f == NULL)
        return -1;

    // operazioni su f...

    fclose(f);

    return 0;
}
```

Apro il file chiamando la `fopen()`. Passo il nome del file e la stringa `"r"` dato che voglio aprire un file in modalità lettura tradotta.

Controllo che l'apertura del file sia riuscita. Se `f` è `NULL` allora si è verificato un errore.

Quando il file non mi serve più lo chiudo chiamando la `fclose()`.

Lettura da file

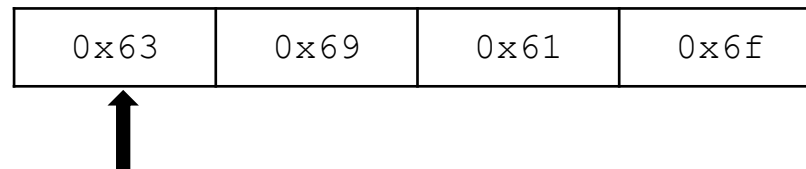
- Iniziamo quindi a vedere alcune delle funzioni che `stdio.h` mette a disposizione per leggere e scrivere su file.
- La funzione di lettura più semplice è la **`fgetc()`**, essa legge un solo byte dallo stream (il file deve essere stato aperto in lettura):

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

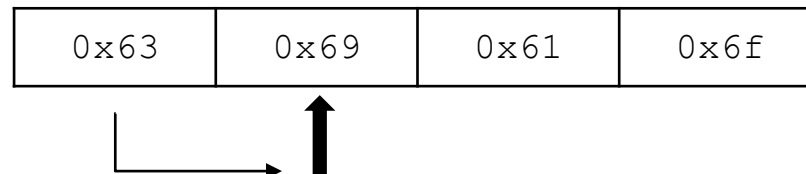
- La funzione ha come unico parametro il file da cui vogliamo leggere e restituisce il byte letto oppure `EOF` se è stata raggiunta la fine del file o se si è verificato qualche altro tipo di errore.
- Ma se stiamo leggendo un solo byte perché il tipo di ritorno è un `int`?
- Ritornare un `int` ci permette di distinguere i casi in cui è stato ritornato un `EOF` da tutti gli altri.
- Se usassimo `char` come tipo di ritorno non saremmo in grado di distinguere la lettura di un byte dal valore `0xff` (255) dall'`EOF` (-1)!

fgetc()

- Quando leggiamo un byte con la funzione `fgetc()` spostiamo in avanti il cursore di un byte.
- Immaginiamo di aprire un file che contiene 4 byte. Supponiamo che la situazione iniziale dello stream dopo l'apertura del file sia la seguente:



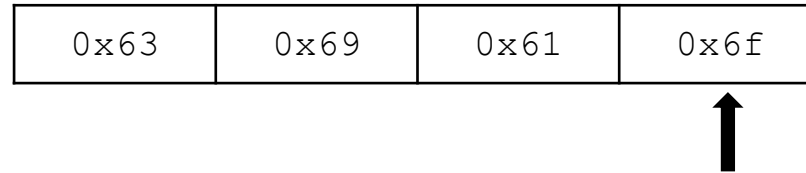
- La freccia nera indica la posizione del cursore.
- Dopo una chiamata a `fgetc()` la funzione legge il primo byte dallo stream e lo avrà ritornato e il cursore si sarà spostato in avanti di una posizione (un byte) :



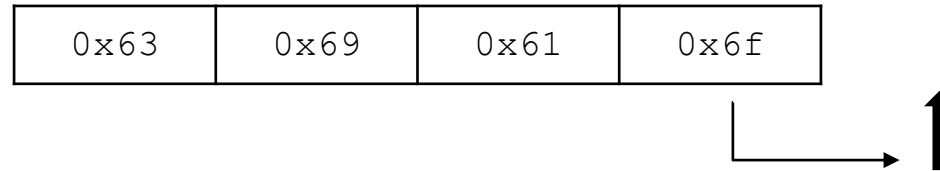
- In questo caso la funzione avrà ritornato il valore `0x63` in un `int`.

fgetc()

- Se continuassimo a chiamare la funzione fgetc() sullo stream leggeremmo tutti i byte che il file contiene fino all'ultimo:



- In questo caso possiamo supporre che i byte 0x63, 0x69 e 0x61 siano già stati letti. Una chiamata a fgetc() ora ci restituirà il valore 0x6f e sposterà ulteriormente in avanti il cursore:



- Ora tutto il contenuto del file è stato letto, nello stream non ci sono più byte da leggere.
- A questo punto se chiamassimo fgetc() un'altra volta ancora otterremmo dalla funzione un EOF come valore di ritorno e capiremmo di aver raggiunto la fine dello stream.

Esempio di fgetc()

- Vediamo quindi come possiamo sfruttare la fgetc() in un esempio più completo:

```
#include <stdio.h>

int main(void)
{
    FILE *f = fopen("test.txt", "r");
    if (f == NULL)
        return -1;

    while (1) {
        int c = fgetc(f);
        if (c == EOF )
            break;
        // altre operazioni su c...
    }

    fclose(f);
    return 0;
}
```

Apro il file chiamando la fopen() e controllo che l'apertura sia andata a buon fine.

Uso un ciclo while per iterare su ogni byte letto dallo stream. Il ciclo prosegue finché non viene incontrato l'EOF. All'interno del ciclo posso poi fare operazioni sul byte letto.

Quando ho terminato di usare il file lo chiudo con la fclose().

Come leggere da file

- Analizziamo il ciclo precedente:

```
while (1) {  
    int c = fgetc(f);  
    if (c == EOF )  
        break;  
    // altre operazioni su c...  
}
```

- La logica con cui si deve lavorare in lettura sui file è sempre composta di tre passi: Leggo, Controllo, Uso
 1. Leggo: eseguo una operazione di lettura dal file più o meno complessa
 2. Controllo: verifico che l'operazione sia andata a buon fine
 3. Uso: il dato letto è valido, quindi posso utilizzarlo per quello che intendo fare.
- Se non piace la versione del ciclo "infinito", interrotto con un break, è poi possibile utilizzare altri modi di scriverlo.
- Vediamo nella prossima slide come.

Esempio di fgetc()

- Vediamo quindi come possiamo sfruttare la fgetc() in un esempio più completo:

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

```
int main(void)  
{
```

```
    FILE *f = fopen("test.txt", "r");  
    if (f == NULL)  
        return -1;
```

```
    int c;  
    while ((c = fgetc(f)) != EOF) {  
        // altre operazioni su c...  
    }
```

```
    fclose(f);
```

```
    return 0;
```

```
}
```

Apro il file chiamando la fopen() e controllo che l'apertura sia andata a buon fine.

Uso un ciclo while per iterare su ogni byte letto dallo stream. Il ciclo prosegue finché non viene incontrato l'EOF. All'interno del ciclo posso poi fare operazioni sul byte letto.

Quando ho terminato di usare il file lo chiudo con la fclose().

Esempio di fgetc()

- Esaminiamo meglio il modo in cui è stata scritta la condizione del ciclo `while`:

```
(c = fgetc(f)) != EOF
```

- Questa espressione fa due cose in una riga sola, legge un byte dallo stream e lo assegna a `c` (*Leggo*), in più controlla anche se il carattere letto è diverso dall'EOF (*Controllo*).
- La prima parte: `(c = fgetc(f))`
legge con `fgetc()` un carattere e lo assegna a `c`. Questa espressione ha un valore, che non è altro che il valore che è stato usato per l'assegnamento, quindi il valore letto dal file!
- La seconda parte: `(...) != EOF`
confronta il valore dell'espressione tra parentesi (quindi quella descritta al punto precedente) con `EOF`, se è diverso allora continuo ad eseguire il codice del ciclo `while`.

Esempio di fgetc()

- Il metodo appena visto è il modo più comune per leggere da file fino alla fine del file stesso.
- Permette inoltre di evitare errori come leggere un carattere in più, leggere un carattere in meno, saltare un carattere, eccetera.

- Attenzione alle parentesi! Se avessimo scritto:

```
while(c = fgetc(f) != EOF) { // ... }
```

- Sarebbe stato un errore, così il confronto avrebbe avuto la precedenza sull'assegnamento:

```
fgetc(f) != EOF
```

e alla variabile c avremmo assegnato di conseguenza il risultato del confronto, quindi sempre 0 o 1!

- Il ciclo sarebbe terminato nel momento giusto, cioè alla fine del file, ma non avremmo comunque potuto usare il valore letto da esso.

Scrittura su file

- Il corrispondente in scrittura di `fgetc()` è la funzione **`fputc()`**:

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

- La funzione ha due parametri, il primo è il byte che si vuole scrivere, il secondo è il file su cui si vuole scrivere il byte.
- La funzione ritorna un intero che, in caso di scrittura andata a buon fine, ha lo stesso valore del byte che si voleva scrivere, in caso di fallimento ritorna `EOF`.
- Esattamente come la `fgetc()`, la `fputc()` sposta il cursore in avanti di un byte sullo stream.
- Ovviamente il file su cui vogliamo scrivere il carattere deve essere stato aperto in scrittura (o append).

fputc()

- Vediamo ora un esempio di utilizzo di fputc(), proviamo a scrivere su un file il contenuto di una stringa carattere per carattere:

```
#include <stdio.h>

int main(void)
{
    FILE *f = fopen("test2.txt", "w");
    if (f == NULL)
        return -1;

    const char *s = "stringa di prova";

    for (size_t i = 0; s[i] != 0; ++i) {
        fputc(s[i], f);
    }

    fclose(f);

    return 0;
}
```

Apro il file «test2.txt» in modalità scrittura e controllo che l'apertura sia riuscita. Il file «test2.txt» non esiste ma viene creato appena provo ad aprirlo.

Definisco un puntatore const alla stringa da scrivere sul file.

Con un ciclo for scorro la stringa e scrivo un carattere alla volta. Non c'è bisogno di copiare anche il terminatore, su file il concetto di array zero terminato non si applica più, bastano solo i caratteri.

Chiudo il file.

fgets()

- Passiamo ora a un'altra funzione che ci permette di leggere direttamente una stringa da file, la **fgets()**:

```
char *fgets(char *str, int count, FILE *stream);
```

- la funzione prova a leggere dal file puntato da `stream` un certo numero di caratteri pari a `count - 1` e li scrive nell'array puntato da `str`.
- Se durante la lettura viene letto un a capo o si incontra la fine del file (prima di riuscire a leggere `count - 1` caratteri) allora la lettura si ferma. Nel byte di `str` successivo all'ultimo letto dal file viene scritto il terminatore. L'eventuale carattere a capo viene lasciato nella stringa.
- La funzione, in caso di successo ritorna una copia del puntatore `str`, altrimenti in caso di fallimento (ad esempio se proviamo a leggere ma lo `stream` è già arrivato a `EOF`) ritorna `NULL`.
- Se `count` è minore di 1 allora si genera un undefined behavior.
- Dopo lettura il cursore si sposta in avanti del numero di byte letti.

fgets()

- Supponiamo di aver aperto un file in lettura i cui byte sono:

'C'	'i'	'a'	'o'	10	'M'	'a'	'r'	'c'	'o'
-----	-----	-----	-----	----	-----	-----	-----	-----	-----

- E di provare a leggere tre volte una stringa di 8 byte, ad esempio con queste chiamate:

```
char str[8];
```

```
char *ret;
```

```
ret = fgets(str, sizeof str, f);
```

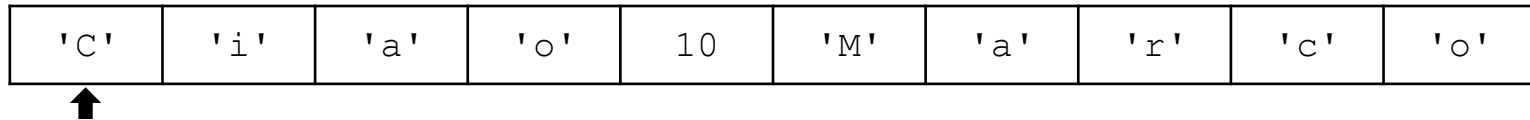
```
ret = fgets(str, sizeof str, f);
```

```
ret = fgets(str, sizeof str, f);
```

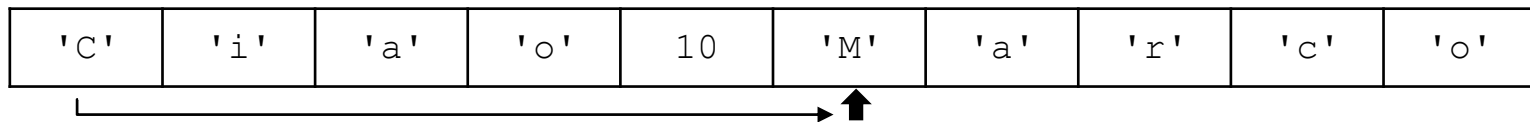
- `str` è la stringa in cui proviamo a scrivere i byte letti dal file e `ret` è il puntatore che usiamo per memorizzare il valore di ritorno.
- In questo caso riutilizziamo sempre la stessa stringa per memorizzare i caratteri letti dal file.

fgets()

- La situazione iniziale dopo l'apertura del file è la seguente:



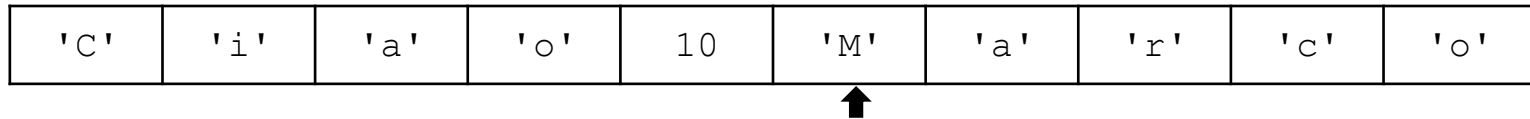
- Il cursore punta al primo byte dello stream.
- Eseguiamo la prima chiamata a fgets().
- Il parametro `count` è `sizeof str`, cioè 8. Stiamo quindi provando a leggere 7 caratteri e l'ottavo verrà riempito dal terminatore.
- Dopo la chiamata la situazione sarà questa:



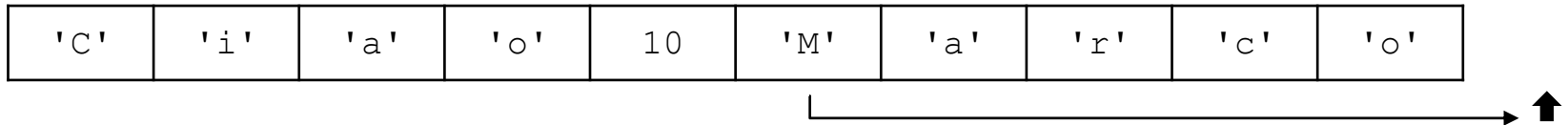
- La funzione ha smesso di leggere quando ha incontrato un a capo e quindi il cursore si è spostato in avanti di 5 byte.
- In `str` avremo: **"Ciao\n"** e `ret` sarà un puntatore all'inizio di `str`.

fgets()

- Dopo la prima lettura il cursore punta alla «M»:



- La seconda chiamata a `fgets()` è uguale alla prima, vogliamo sempre provare a leggere 7 caratteri.
- Ma nello stream, dopo il cursore, non ci sono 7 caratteri ma solamente 5. la funzione ovviamente smetterà di leggere quando incontrerà la fine dello stream.
- Dopo la chiamata la situazione sarà questa:



- In `str` avremo: **"Marco"** e `ret` sarà un puntatore all'inizio di `str`.

fgets()

- Dopo la seconda lettura il cursore è già arrivato alla fine dello stream:

'C'	'i'	'a'	'o'	10	'M'	'a'	'r'	'c'	'o'
-----	-----	-----	-----	----	-----	-----	-----	-----	-----



- Anche la terza chiamata a fgets() è uguale alle prime due.
- La fgets() però non riuscirà a leggere niente dallo stream e quindi fallirà.
- In `str` avremo ancora **"Marco"** dato che in caso di fallimento la funzione non modifica `str`.
- `ret` questa volta sarà `NULL` per indicare il fallimento.

fputs()

- Così come per `fgetc()` per la lettura esiste la corrispondente `fputc()` per la scrittura, anche per `fgets()` esiste la corrispondente **`fputs()`** per scrivere una stringa su file:

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

- La funzione ha due parametri, `str` è un puntatore a una stringa C che deve essere scritta sul file, `stream` è il puntatore al file su cui deve essere scritta `str`.
- In questo caso non è necessario specificare il numero di caratteri da scrivere dato che `str` deve puntare ad un array zero terminato.
- Il valore di ritorno è un intero. In caso di scrittura avvenuta con successo ritorna un valore non negativo, in caso di fallimento ritorna `EOF`.
- Il terminatore della stringa NON viene scritto sul file.
- Se non si verificano errori la scrittura della stringa sposta in avanti il cursore di un valore pari al numero di caratteri scritti, ricordando che il terminatore non viene scritto.

fputs()

- Esempio:

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

```
int main(void) {
```

```
    FILE *f = fopen("test4.txt", "w");
```

```
    if (f == NULL)
```

```
        return -1;
```

```
    char s1[] = "Fondamenti di";
```

```
    char s2[] = "informatica";
```

```
    char s3[] = "2016";
```

```
    fputs(s1, f);
```

```
    fputs("\n", f);
```

```
    fputs(s2, f);
```

```
    fputs(" ", f);
```

```
    fputs(s3, f);
```

```
    fclose(f);
```

```
    return 0;
```

```
}
```

Apro il file «test4.txt» in modalità scrittura e controllo che l'apertura sia riuscita.

Definisco le tre stringhe che scriverò sul file.

Chiamo fputs() per scrivere tutte le tre stringhe, in più la uso anche per scrivere un a capo dopo «Fondamenti di» e uno spazio tra «informatica» e «2016».

Chiudo il file.

fputs()

- Continuando con l'esempio precedente, supponiamo che il file «test4.txt» non esistesse prima dell'esecuzione del programma.
- Cosa troveremmo nel file se lo aprissimo dopo aver eseguito il programma?

test4.txt

```
Fondamenti di  
informatica 2016
```

- Nel file ritroviamo tutte le stringhe che abbiamo scritto in precedenza.
- Se usassimo la rappresentazione di uno stream che abbiamo già usato in precedenza per visualizzare il contenuto del file avremmo:

F	o	n	d	a	m	e	n	t	i		d	i	\n	i	n	f	o	r	m	a	t	i	c	a		2	0	1	6
---	---	---	---	---	---	---	---	---	---	--	---	---	----	---	---	---	---	---	---	---	---	---	---	---	--	---	---	---	---

- Nel file non ci sono i terminatori e c'è un a capo tra «di» e «informatica».
- La dimensione totale del file sarà quindi 30 byte (su Windows saranno 31 byte perché il ' \n ' viene sostituito dalla sequenza " \r \n ").

fprintf()

- Finora abbiamo visto come scrivere su file una singola stringa, ma non abbiamo visto neanche un modo per «creare stringhe» in base a certi parametri, se non facendolo a mano con delle funzioni scritte apposta.
- Ad esempio, solo per creare una stringa che contenesse la rappresentazione testuale di un numero intero abbiamo dovuto scrivere una funzione!
- Per fortuna nella libreria standard del C sono state inserite anche funzioni che facilitano questi compiti e che ci permettono di scrivere su file stringhe in modo più elaborato e comodo.
- La prima funzione che analizziamo è la **fprintf()**, essa è la diretta evoluzione della fputs() e ci permette di scrivere su file stringhe, facendosi carico della conversione delle variabili, dal loro tipo fondamentale alla loro rappresentazione testuale.

fprintf()

- La `fprintf()` è una funzione con una dichiarazione particolare, essa può infatti *accettare un numero variabile di parametri*:

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

- I puntini all'interno delle parentesi non sono un modo per abbreviare e risparmiare spazio, stanno ad indicare che dopo i primi due parametri possono comparirne altri in numero variabile (anche 0).
- Il parametro `stream` è il puntatore al file su cui vogliamo scrivere la stringa.
- Il parametro `format` è una stringa C che rappresenta il modo in cui vogliamo «comporre» la stringa da scrivere sul file. Vedremo in dettaglio tra poco come usarlo.
- Di seguito a `format`, vengono passate tutte le variabili che dovranno essere usate per comporre la stringa risultante.
- La funzione ritorna un intero che contiene il numero di caratteri scritti sul file oppure un valore negativo se si è verificato un errore.

Il parametro `format` della `fprintf()`

- Il parametro `format` è una stringa C che indica alla `fprintf()` come dovrà essere composta la stringa che dovrà essere scritta sul file.
- È una normalissima stringa ma all'interno di essa possiamo usare delle sequenze speciali di caratteri per indicare dove la `fprintf()` dovrà andare a inserire le rappresentazioni testuali dei parametri che seguono `format`.
- Le sequenze speciali iniziano tutte con il carattere '%', dopo il quale vengono inseriti dei modificatori per indicare come devono essere rappresentati testualmente i parametri.
- I modificatori ci permettono di stabilire varie opzioni per le conversioni, ad esempio:
 - Numero di caratteri utilizzati per la rappresentazione.
 - Numero di caratteri usati per la parte decimale di una variabile floating point.
 - Specificare il tipo di conversione da applicare.
- Il modo di scrivere queste sequenze va ovviamente rispettato con precisione.
- Vediamo in dettaglio come scrivere queste sequenze.

Il parametro `format` della `fprintf()`

- Il formato è il seguente:

`%[flags][width][.precision][length]specifier`

- All'apparenza può sembrare abbastanza complicato da utilizzare. Le parti scritte tra parentesi quadre sono opzionali, quindi le uniche cose da utilizzare obbligatoriamente sono il `%` iniziale e uno `specifier` per indicare che tipo di conversione si vuole effettuare.
- Partiamo ad analizzare questo formato dagli `specifier`.
- Tutti i possibili `specifier` sono singoli caratteri:

Specifier per tipi interi	
c	Rappresentazione come singolo carattere.
d i	Rappresentazione come intero con segno in decimale.
u	Rappresentazione come intero senza segno in decimale.
o	Rappresentazione come intero senza segno in ottale.
x X	Rappresentazione come intero senza segno in esadecimale (rispettivamente con lettere maiuscole o minuscole).

Il parametro `format` della `fprintf()`

Specifier per tipi con la virgola	
f F	Conversione in un numero double con la virgola senza usare notazione esponenziale.
e E	Conversione in un numero con la virgola usando la notazione esponenziale. La «e» della notazione esponenziale è scritta maiuscola o minuscola rispettivamente.
a A	Conversione in un numero con la virgola usando la notazione esponenziale esadecimale.
g G	Usa la rappresentazione più corta tra quelle di f ed e .

Altri specifier	
s	Inserisce una stringa.
p	Converte un puntatore in una rappresentazione testuale.
n	Non produce output. Riempie il puntatore a int passato come parametro con il numero di caratteri scritti in output fino a quel momento.
%	Scrivere il carattere «%». È una sequenza di escape per scrivere il carattere % che altrimenti verrebbe sempre interpretato come l'inizio di una sequenza.

- Questo è solo un riassunto e per ognuno di questi `specifier` ci sono varie sottigliezze che potrebbero trarre in inganno, gli stessi modificatori potrebbero avere comportamenti diversi in combinazione con certi `specifier`.
- La reference viene sempre in aiuto in questi casi!

Il parametro `format` della `fprintf()`

- Il campo `length` serve per specificare se la variabile deve essere interpretata come `long`, in tal caso bisognerà inserire una «`l`» (ad esempio `%lld` per un `long long int`).
- Il campo `precision` deve contenere un «`.`» e un numero che indica il numero massimo di caratteri per una stringa (`%s`) o il numero di cifre decimali per variabili floating point (`%f` ad esempio).
- Il campo `width` indica il numero minimo di caratteri da utilizzare durante la conversione, se la rappresentazione ne richiede meno i restanti caratteri vengono riempiti con degli spazi.

Flags	
-	Allinea a sinistra (di default tutto viene allineato a destra).
+	Quando vengono rappresentati numeri con segno forza la rappresentazione del segno (normalmente il + verrebbe omissso).
(space)	Quando vengono rappresentati numeri con segno inserisce uno spazio al posto del +.
#	Per alcune conversioni esiste una forma alternativa che è possibile usare utilizzando il # nei flags.
0	Quando il campo <code>width</code> è settato a un numero maggiore dei caratteri effettivamente necessari riempie i restanti con degli zeri (di default sarebbero stati usati degli spazi).

Gli altri parametri della fprintf()

- Dopo il parametro format vengono inseriti come parametri tutti quelli che devono essere usati nelle conversioni.
- I parametri vengono usati nell'ordine in cui vengono passati alla funzione per essere convertiti.
- Se dopo il parametro format vengono passati più parametri di quelli che vengono effettivamente usati in format per delle conversioni, quelli in più non vengono considerati.
- Se invece passiamo meno parametri di quelli che servirebbero si genera un undefined behavior.
- I parametri possono essere di qualsiasi tipo, bisogna quindi stare attenti a come si scrive la stringa format per fare in modo che le conversioni effettuate dalla fprintf() abbiano effettivamente senso!
- Ovviamente nella stringa format possiamo ancora usare le sequenze di escape per tab e a capo!
- Per completezza, vale la pena sottolineare che quando si passa un float ad una funzione ad argomenti variabili (tipo printf) questo viene promosso a double.

Esempi

- Vediamo qualche esempio di utilizzo di fprintf():

```
#include <stdio.h>
int main(void) {
    int x = 35, y = -7;
    long long ll = LLONG_MAX;
    char c1 = 13, c2 = 'v';
    float f1 = 12.65, f2 = 5.6;
    double d = 5.3e-15;
    char *s = "una stringa";
    double *ptr = &d;

    fprintf(stdout, "x vale %d, y vale %d\n", x, y);
    fprintf(stdout, "x vale %+4d\n", x);
    fprintf(stdout, "ll vale %lld\n", ll);
    fprintf(stdout, "x e y in esadecimale: %#x e %X\n", x, y);
    fprintf(stdout, "x e y in ottale: %o e %o\n", x, y);
    fprintf(stdout, "f1 e f2 valgono: %f e %f\n", f1, f2);
    fprintf(stdout, "f1 e f2 valgono: %e e %E\n", f1, f2);
    fprintf(stdout, "f1 e f2 valgono: %+1f e %.7f\n", f1, f2);
    fprintf(stdout, "d vale: %f e %.1f\n", d, d);
    fprintf(stdout, "d vale: %le\n", d);
    fprintf(stdout, "s contiene: %s\n", s);
    fprintf(stdout, "ptr contiene l'indirizzo %p\n", ptr);

    return 0;
}
```

Il seguente programma scrive su stdout, usando la fprintf(), varie rappresentazioni delle variabili inizializzate in precedenza. Cosa potremo leggere sulla console dopo l'esecuzione?

Esempi

- Il risultato sarà:

```
x vale 35, y vale -7
x vale +0035
ll vale 9223372036854775807
x e y in esadecimale: 0x23 e FFFFFFFF9
x e y in ottale: 43 e 37777777771
f1 e f2 valgono: 12.650000 e 5.600000
f1 e f2 valgono: 1.265000e+001 e 5.600000E+000
f1 e f2 valgono: +12.6 e 5.5999999
d vale: 0.000000 e 0.0
d vale: 5.300000e-015
s contiene: una stringa
ptr contiene l'indirizzo 0038FD40
```

- A ognuna delle `fprintf()` corrisponde una riga stampata sulla console, infatti alla fine di tutte le stringhe format passate alla `fprintf()` era presente un a capo.
- Analizziamo quindi ognuna di esse.

Esempi

- 1) `fprintf(stdout, "x vale %d, y vale %d\n", x, y);`
- 2) `fprintf(stdout, "x vale %+.4d\n", x);`
- 3) `fprintf(stdout, "ll vale %lld\n", ll);`
- 4) `fprintf(stdout, "x e y in esadecimale: %#x e %X\n", x, y);`
- 5) `fprintf(stdout, "x e y in ottale: %o e %o\n", x, y);`

- In 1 vogliamo stampare come interi con segno `x` e `y`, quindi usiamo `%d`.
- In 2 stampiamo solo `x` includendo però il segno (+), indipendentemente da positivo o negativo, e specificando che vogliamo usare 4 cifre per rappresentarlo (`.4`).
- In 3 stampiamo `ll`, che è un long long, quindi usiamo il modificatore «`ll`» prima della `d`.
- In 4 stampo `x` e `y` in esadecimale, per `x` uso il `#` per usare la rappresentazione alternativa che aggiunge automaticamente `0x` davanti al numero, per `y` uso la `X` maiuscola che stampa usando lettere maiuscole per le cifre esadecimali.
- In 5 stampo `x` e `y` nella loro rappresentazione ottale.

Esempi

- 6) `fprintf(stdout, "f1 e f2 valgono: %f e %f\n", f1, f2);`
- 7) `fprintf(stdout, "f1 e f2 valgono: %#e e %E\n", f1, f2);`
- 8) `fprintf(stdout, "f1 e f2 valgono: %+.1f e %.7f\n", f1, f2);`

- In 6 stampo `f1` e `f2` come double usando la rappresentazione di default, che usa una precisione di 6 cifre decimali.
- In 7 stampo sempre `f1` e `f2` usando la notazione esponenziale, per `f1` uso quella alternativa, che stampa il punto decimale anche se non ci sono cifre decimali (in questo caso non ha effetto perché `f1` ha effettivamente delle cifre decimali). Usare la `E` maiuscola per `f2` stampa la «e» della notazione esponenziale in maiuscolo.
- In 8 specifico che `f1` deve essere stampato includendo il segno e un massimo di una cifra decimale, mentre per `f2` specifico che deve essere stampato usando 7 cifre decimali.

Esempi

```
9) fprintf(stdout, "d vale: %f e %.1f\n", d, d);
```

```
10) fprintf(stdout, "d vale: %e\n", d);
```

```
11) fprintf(stdout, "s contiene: %s\n", s);
```

```
12) fprintf(stdout, "ptr contiene l'indirizzo %p\n", ptr);
```

- In 9 stampo `d` in due modi: prima usando la rappresentazione di default per i double (cioè `%f` che usa 6 cifre decimali), poi usando solamente una cifra decimale, specificando il `.1`.
- In questo caso viene stampato sempre 0, una o sei cifre decimali infatti non bastano per rappresentare il valore di `d` (5.3×10^{-15}).
- in 10 stampo `d` usando la notazione scientifica (e questa volta viene stampato correttamente).
- In 11 stampo al posto di `%s` il contenuto della stringa `s`.
- In 12 stampo il valore del puntatore `ptr`.

fscanf()

- Così come per `fputc()` e `fputs()` esistono le corrispondenti funzioni `fgetc()` e `fgets()` per la lettura, anche per la `fprintf()` esiste il corrispondente in lettura, la **`fscanf()`**:

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

- La dichiarazione è molto simile a quella della `fprintf()`.
- Il parametro `stream` è il file da cui leggere.
- Il parametro `format` è una stringa in cui definiamo il formato dei dati da leggere, ma **questo è diverso da quello della `fprintf()`**.
- I parametri variabili sono costituiti dalle variabili che verranno usate per memorizzare i dati (quindi saranno *puntatori* alle variabili da usare!).
- Il valore di ritorno è un intero che rappresenta il numero di parametri letti correttamente dal file oppure EOF se la lettura fallisce prima di leggere correttamente il primo parametro.
- La `fscanf()` effettua la conversione nel senso opposto rispetto alla `fprintf()`, da una rappresentazione testuale a quella usata in memoria.

fscanf()

- La fscanf() è molto utile perché ci permette di leggere valori memorizzati in file (di testo) senza dover convertire manualmente i caratteri letti!
- L'unica cosa da fare è quella di specificare il formato dei dati da leggere.
- Il modo in cui scrivere la stringa format assomiglia molto a quello che abbiamo già visto nel caso della fprintf(), ci sono però **alcune differenze molto importanti di cui tenere conto**.
- Durante la lettura dal file i caratteri che non fanno parte di uno specifier vengono trattati diversamente:
 - Quando in format viene specificata una sequenza di caratteri allora questa **deve comparire** uguale anche in lettura dal file.
 - Quando in format viene specificato un whitespace (non c'è differenza tra spazio, tab, a capo o combinazioni di essi) allora tutti i whitespace eventualmente incontrati consecutivamente in lettura dal file vengono letti e scartati.
- Per la fscanf() il formato degli specifier è il seguente:
`%[*][width][length]specifier`
- I campi tra parentesi quadre sono opzionali.

fscanf()

- L'asterisco, se specificato, indica alla funzione di leggere un elemento ma di NON memorizzarlo in uno dei parametri.
- Il campo `width` è costituito da un numero (maggiore di zero) che specifica il numero massimo di caratteri che la funzione deve usare per leggere un certo valore.
- Il campo `length`, in maniera simile alla `fprintf()`, specifica la dimensione della variabile in cui andrà memorizzato il dato letto dal file.
- Gli `specifier` ricordano vagamente quelli visti per la `fprintf()`, **ma sono diversi**.
- Analizziamo nel dettaglio gli specificatori della `scanf()`.

Specificatori per numeri interi

- Lo specificatore %i della fscanf() sono identici e permettono di leggere numeri in base 10, 8 o 16 con la sintassi del C. In alternativa %d, %u, %o e %x servono per supportare solo la base specificata.
- Ognuno di questi specificatori, come prima cosa, salta eventuali whitespace. È quindi equivalente scriverli con uno o più spazi davanti.

Specificatore	Tipo di dato	Descrizione
i	int *	Salta i whitespace, estrae un eventuale segno e poi un numero intero. Di default assume che il numero sia in base 10, ma un prefisso 0 indica un numero in base 8, e un prefisso 0x o 0X indica un numero in base 16.
d	int *	Salta i whitespace, estrae un eventuale segno e poi un numero intero in base 10.
u	unsigned int *	Salta i whitespace, estrae un eventuale segno e poi un numero intero senza segno in base 10.
o	unsigned int *	Salta i whitespace, estrae un eventuale segno e poi un numero intero senza segno in base 8.
x	unsigned int *	Salta i whitespace, estrae un eventuale segno e poi un numero intero senza segno in base 16, che può essere eventualmente preceduto da un prefisso 0x o 0X.

Specificatori per numeri in virgola mobile

- Gli specificatori %f, %e, %g e %a della fscanf() sono identici tra loro e permettono di leggere numeri in virgola mobile espressi con la sintassi del C.
- Ognuno di questi specificatori, come prima cosa, salta eventuali whitespace. È quindi equivalente scriverli con uno o più spazi davanti.
- Per specificare che si vuole leggere il dato in un double e non in un float bisogna aggiungere una «l» prima della lettera dello specificatore.

Specificatore	Tipo di dato	Descrizione
f, e, g, a	float*	Salta i whitespace, estrae un eventuale segno poi un numero con la virgola, se il numero non ha parte decimale il punto è opzionale. Il numero può essere seguito da una e o E e da un numero intero.
lf, le, lg, la	double*	Le implementazioni che supportano il C99 permettono di leggere numeri floating point anche se espressi in esadecimale, cioè preceduti da 0x o 0X.

Specificatori per stringhe C

- Gli specificatori %s e %[] sono quelli utilizzati per leggere stringhe C da file.
- Il più importante è sicuramente %[], %s è solo un caso particolare di %[] e non è troppo flessibile.
- Lo spazio necessario per contenere i caratteri letti più il terminatore deve essere già stato allocato prima di leggere dal file.
- Con un trattino «-» all'interno delle parentesi quadre si indicano sequenze di caratteri, ad esempio le cifre si indicano con [0-9].

<i>Specificatore</i>	<i>Tipo di dato</i>	<i>Descrizione</i>
[<i>caratteri</i>]	char*	Non salta i whitespace , estrae qualsiasi numero di caratteri compresi tra quelli specificati tra parentesi. Si ferma quando ne incontra uno diverso.
[^ <i>caratteri</i>]	char*	Non salta i whitespace , estrae qualsiasi numero di caratteri NON compresi tra quelli specificati tra parentesi dopo il "^". Si ferma quando ne incontra uno di quelli specificati.
s	char*	Modo alternativo per scrivere " %[^ \n\t] ". Salta eventuali whitespace, poi legge tutti i caratteri che trova e si ferma se incontra un whitespace.

Specificatore per singoli caratteri

- Lo specificatore per singoli caratteri si comporta diversamente dagli altri visti finora.
- Il %c legge sempre un solo carattere dallo stream senza fare differenza tra whitespace e non-whitespace.
- Se si vogliono saltare i whitespace prima del carattere bisogna specificarlo con uno spazio prima del %c (" %c").

Specificatore	Tipo di dato	Descrizione
c	char*	Non salta i whitespace ed estrae un singolo carattere, qualsiasi esso sia, whitespace compresi.
nc (n è un numero)	char*	Non salta i whitespace ed estrae <i>n</i> caratteri. I caratteri vengono inseriti in memoria a partire dal puntatore e nelle posizioni successive.

Altri specificatori

- Esistono anche altri specificatori per usi più particolari:
 - Quello per leggere un indirizzo di memoria rappresentato in forma testuale.
 - Quello per inserire in una variabile il numero di caratteri letti.
 - Quello per leggere il carattere «%» dallo stream.

<i>Specificatore</i>	<i>Tipo di dato</i>	<i>Descrizione</i>
p	void**	Salta i whitespace ed estrae un numero in esadecimale valido come indirizzo di memoria. Il formato per la rappresentazione di un indirizzo può variare a seconda del sistema ma è la stessa usata dalla fprintf().
n	int*	Non estrae dallo stream nessun carattere ma memorizza nel parametro passato il numero di caratteri letti dallo stream fino a quel momento.
%	-	Serve per leggere dallo stream il carattere «%». Il carattere letto non viene memorizzato in nessun parametro.

fscanf()

- Vediamo un esempio di fscanf(). Supponiamo di voler leggere i dati contenuti nel file «prova.txt», il cui contenuto è:

prova.txt

```
21 5.6 fdi2016 0x3d5f 2.5e-18  
55  
  
87
```

- Stiamo quindi cercando di leggere un certo numero di variabili, per la precisione, 4 int (di cui uno scritto in esadecimale), una stringa, e due variabili floating point (una float e una double).
- Lo spazio in memoria per tutte queste variabili dovrà già essere allocato prima della chiamata a fscanf()!

Esempio

- Il codice per leggere queste variabili dal file potrebbe essere il seguente:

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

```
int main(void)
```

```
{
```

```
    FILE *in_file = fopen("prova.txt", "r");
```

```
    if (in_file == NULL)
```

```
        return -1;
```

```
    int i1, i2, i3, hex;
```

```
    float f;
```

```
    double d;
```

```
    char str[20];
```

```
    int ret = fscanf(in_file, "%d%f%19s%x%lf%d%d",
```

```
        &i1, &f, str, &hex, &d, &i2, &i3);
```

```
    fclose(in_file);
```

```
    return 0;
```

```
}
```

Apro il file in modalità lettura tradotta.

Controllo che l'apertura del file sia andata a buon fine,

Dichiaro le variabili che userò per memorizzare i dati letti dal file

Chiamo la fscanf().

Chiudo il file.

Esempio

- Analizziamo meglio la chiamata a `fscanf()` dell'esempio precedente.
- Con la stringa format dell'esempio la `fscanf()` si aspetta di leggere 7 valori.
- Ognuno di questi come prima cosa "mangia" eventuali whitespace che precedono il dato, poi inizia a convertire tutto quello che riesce.

"%d%f%19s%x%lf%d%d"

Elemento da leggere	Specifier	Tipo parametro corrispondente	Variabile da utilizzare
21	%d	int*	i1
5.6	%f	float*	f
fdi2016	%19s	char*	str
0x3d5f	%x	unsigned int*	hex
2.5e-18	%lf	double*	d
55	%d	int*	i2
87	%d	int*	i3

printf() e scanf()

- La fprintf() e la fscanf() sono le funzioni per lettura e scrittura da file.
- Nella libreria stdio.h esistono però anche altre due varianti di queste due, la **printf()** e la **scanf()**.
- Queste due si comportano esattamente come fprintf() e fscanf(), l'unica differenza è che non hanno un parametro di tipo FILE * per indicare su quale file effettuare l'operazione.
- Queste utilizzano direttamente **stdout** e **stdin** rispettivamente. Servono solo a scrivere meno caratteri.
- Esistono anche due utilissime versioni di queste funzioni per leggere e scrivere su stringhe: sprintf() e sscanf().
- Il funzionamento è analogo a quanto già visto, ma il primo parametro è un puntatore a char, ovvero una stringa C da cui leggere o scrivere.
- L'utilizzo però di un file (dotato di posizione corrente) e di una stringa (in cui non c'è una posizione corrente) ne rende l'utilizzo un po' diverso e serve un po' di pratica per utilizzarle.
- **In ogni caso è sempre importante fare riferimento ad una reference completa con tutti i dettagli del loro funzionamento.**

File binari

- Per ora abbiamo visto solamente le funzioni per operare su file di testo, vediamo anche quelle per operare su file binari.
- Per i file binari esistono principalmente due funzioni, una per leggere e una per scrivere, chiamate rispettivamente **fread()** e **fwrite()**.
- Le dichiarazioni delle due funzioni sono praticamente identiche e anche il loro principio di funzionamento è lo stesso.
- Le dichiarazioni sono:

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
```

- Entrambe hanno lo stesso numero di parametri:
 - `buffer` è un puntatore allo spazio di memoria usato per contenere i dati che verranno letti o scritti.
 - `size` indica la dimensione di un singolo elemento contenuto in `buffer`.
 - `count` il numero di elementi di dimensione `size` da leggere o da scrivere in `buffer`.
 - `stream` è il file da cui leggere o scrivere.

fread()

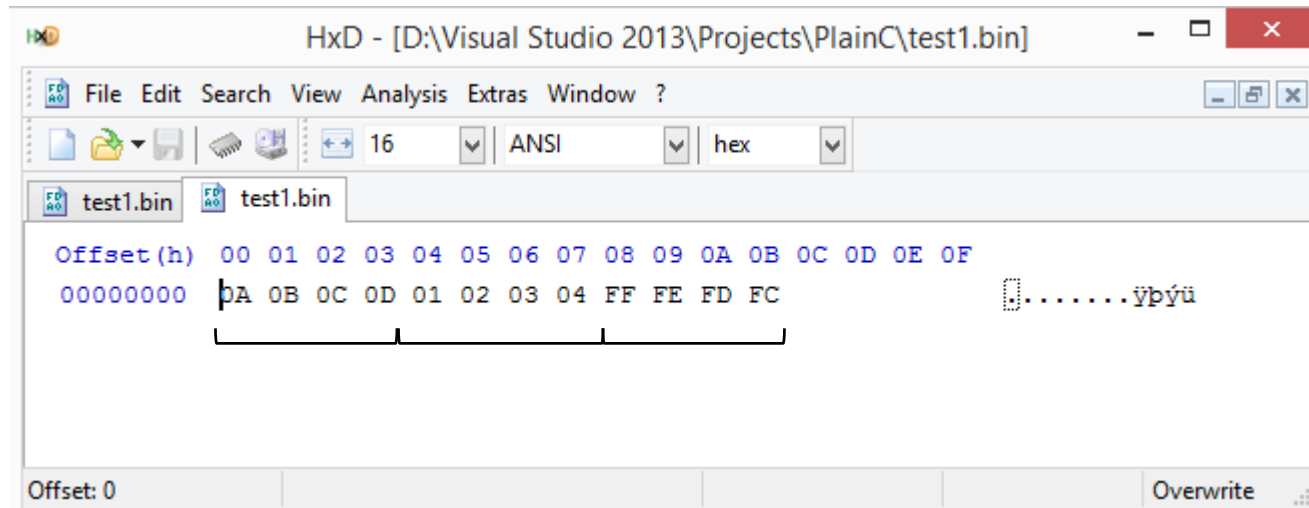
- La `fread()` prende come parametro un puntatore al primo elemento di un'area di memoria che verrà usata per contenere i dati letti.
- Lo spazio necessario deve già essere stato allocato!
- Se ad esempio vogliamo leggere da file 5 double dovremo allocare spazio per `5 * sizeof(double)`.
- Del resto 5 e `sizeof(double)` saranno anche rispettivamente il terzo e il secondo parametro della `fread()`, che ci chiede la dimensione di un elemento e il numero di elementi da leggere.
- Come quarto e ultimo parametro passeremo il file da cui bisogna leggere i dati.
- Il valore di ritorno di `fread()` è un `size_t` che contiene il numero di elementi letti correttamente dal file.
- Se questo valore è diverso dal parametro `count` allora si è verificato un errore o è stata raggiunta la fine del file.

fread()

- Bisogna ovviamente prestare attenzione alla dimensione di memoria allocata per `buffer`.
- Se dovessimo leggere per sbaglio più variabili di quelle che possono effettivamente essere contenute in buffer allora si genererebbe un undefined behavior, del resto è una scrittura a caso in memoria!
- Attenzione anche al fatto che la `fread()` non distingue tra il raggiungimento della fine del file e il verificarsi di un errore.
- Quando una lettura ritorna un valore diverso dal numero di elementi che si volevano leggere bisogna controllare lo stato del file attraverso le funzioni **`feof()`** e **`ferror()`**.
- `feof()` ritorna un valore diverso da 0 se l'ultima operazione di lettura è fallita perché si è tentato di leggere oltre la fine del file, 0 altrimenti.
- `ferror()` ritorna un valore diverso da 0 se l'ultima operazione di lettura o scrittura è fallita per altri tipi di errore, 0 altrimenti. Esempi di errore sono: la scrittura su file protetti, il supporto fisico non esiste più (hai estratto la chiavetta mentre la stavo usando!)

Esempio

- Vediamo un esempio di lettura da file binario.
- Supponiamo di voler leggere il contenuto del seguente file chiamato «test1.bin»:



- Il file contiene 3 interi scritti in little endian che valgono rispettivamente 0x0d0c0b0a, 0x04030201 e 0xfcfdfeff.
- Vediamo i vari casi che potremmo incontrare nell'utilizzo di fread().

Esempio

- Il codice per leggere test1.bin potrebbe essere il seguente:

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

```
int main(void) {
```

```
    int valori[3];
```

```
    FILE *f = fopen("test1.bin", "rb");
```

```
    if (f == NULL)
```

```
        return -1;
```

```
    size_t letti = fread(valori, sizeof(int), 3, f);
```

```
    if (letti != 3) {
```

```
        if (feof(f)) {
```

```
            fclose(f);
```

```
            return -2;
```

```
        }
```

```
        if (ferror(f)) {
```

```
            fclose(f);
```

```
            return -3;
```

```
        }
```

```
    }
```

```
    fclose(f);
```

```
    return 0;
```

```
}
```

Apro il file in modalità lettura binaria usando la «b» e controllo il risultato dell'apertura.

Chiamo la fread(), dicendole di scrivere gli elementi che leggerà dal file `f` nell'array `valori`, e di leggere 3 elementi di dimensione `sizeof(int)`.

Controllo il risultato della lettura verificando il valore `letti` ritornato da fread() e usando le funzioni feof() e ferror().

Chiudo il file visto che non serve più.

Esempio

- Cosa sarebbe successo se avessimo chiesto alla `fread()` di leggere 4 elementi di dimensione `sizeof(int)` invece che 3?

```
int valori[4];

size_t letti = fread(valori, sizeof(int), 4, f);

if (letti != 4) {
    if (feof(f)) {
        fclose(f); ←
        return -2;
    }
    if (ferror(f)) {
        fclose(f);
        return -3;
    }
}

fclose(f);
```

- Il valore di `letti` sarebbe stato comunque 3 (diverso da 4) e la funzione sarebbe terminata ritornando -2 dato che tentando di eseguire la quarta lettura avrebbe raggiunto l'EOF.

`fwrite()`

- L'analogo in scrittura della `fread()` è la **`fwrite()`**.
- Esattamente come la `fread()` ha quattro parametri:
 - `buffer`: un puntatore alla zona di memoria da cui leggere i dati.
 - `size`: un parametro che indica la dimensione di un elemento da scrivere.
 - `count`: un altro parametro che indica quanti elementi scrivere.
 - `stream`: il file su cui scrivere i dati.
- Esattamente come per la `fread()` il valore di ritorno indica il numero di elementi che sono stati effettivamente scritti con successo sul file.
- Se il valore di ritorno è diverso dal parametro `count` allora significa che si è verificato un errore durante la scrittura.
- La `fwrite()` si occupa di scrittura di dati binari, perciò tutte le considerazioni fatte sui file di testo e gli a capo non valgono in questo caso.
- La `fwrite()` legge una serie di byte da un'area di memoria e li scrive su un file esattamente così come sono!

fwrite()

- Vediamo come esempio il contrario di quello che abbiamo visto per la `fread()`, questa volta scriviamo su file 3 int:

```
#include <stdio.h>
int main(void) {
    int valori[] = {
        0x11223344,
        0xcacbcdce,
        0x01020304,
    };

    size_t count = 3;

    FILE *f = fopen("test2.bin", "wb");
    if (f == NULL)
        return -1;

    size_t ret = fwrite(valori, sizeof(int), count, f);
    fclose(f);

    if (ret != count)
        return -2;

    return 0;
}
```

Dichiaro un array di 3 int. In `count` determino la dimensione (in questo caso lo posso fare perché `valori` è un array).

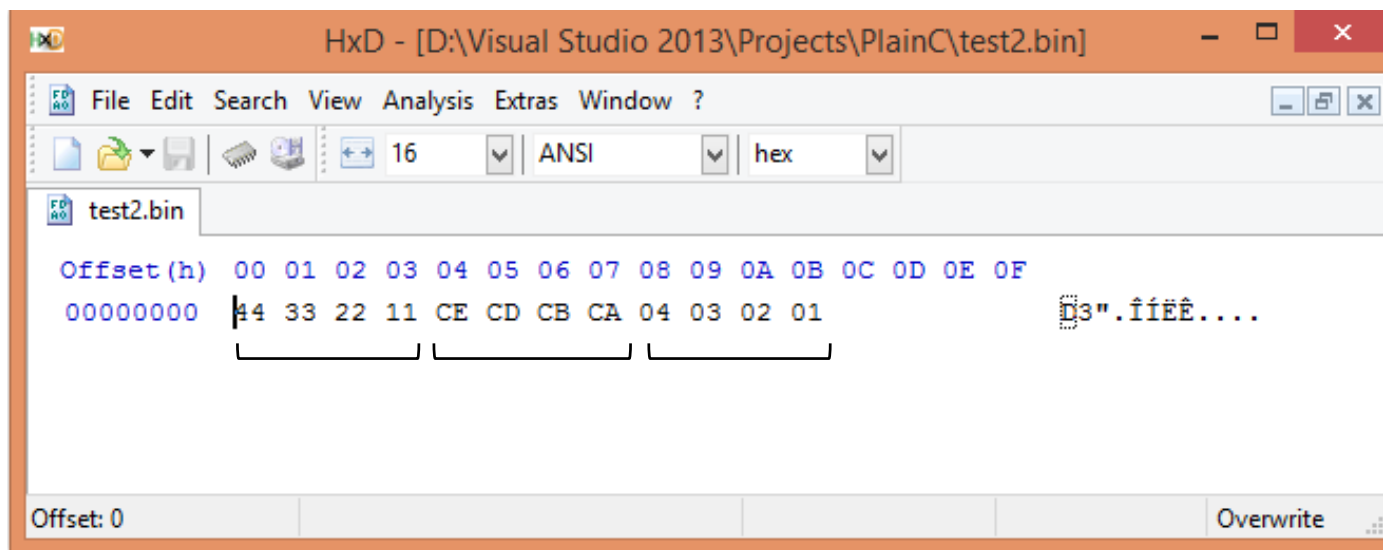
Apro il file «test2.bin» in scrittura in modalità binaria. Successivamente controllo che l'apertura sia andata a buon fine.

Scrivo i 3 int sul file poi chiudo il file.

Controllo il valore di ritorno per determinare se si sono verificati errori.

fwrite()

- Per controllare che la scrittura sia andata veramente a buon fine controllo il contenuto del file con un editor esadecimale, in questo caso un editor di testo non servirebbe a niente:



- In questo caso la scrittura è andata a buon fine e sul file ritrovo esattamente i valori che erano stati inseriti nell'array `valori`.
- Ovviamente nel file sono scritti nello stesso modo in cui erano scritti in memoria, cioè in little-endian.

Spostarsi all'interno di un file

- Finora abbiamo solo visto funzioni che spostavano avanti il cursore.
- Come possiamo fare per spostare il cursore all'interno dello stream senza dover per forza effettuare una lettura?
- Oppure, come possiamo riportare indietro il cursore se ne abbiamo necessità?
- Nella libreria `stdio.h` sono presenti due funzioni che vengono utilizzate per questi scopi, sono **`rewind()`**, **`ftell()`** e **`fseek()`**.
- La più semplice delle tre è la `rewind()`:

```
void rewind(FILE *stream);
```

- Questa sposta il cursore del file passato nel parametro `stream` all'inizio del file.

ftell()

- La funzione `ftell()` ci dice qual è la posizione del cursore all'interno del file:

```
long ftell(FILE *stream);
```

- L'unico parametro `stream` è il file di cui vogliamo conoscere la posizione del cursore.
- Il valore di ritorno è un `long`:
 - In caso di file aperti in modalità binaria corrisponde anche al numero di byte a partire dall'inizio del file.
 - In caso di file aperti in modalità testuale è un valore che ha senso solamente per la funzione `fseek()` che vedremo tra poco.
- Di per sé la funzione `ftell()` non è sempre utile, lo diventa quando viene combinata con la `fseek()`.

fseek()

- La funzione `fseek()` è quella che ci permette di spostare veramente in modo flessibile il cursore all'interno del file:

```
int fseek(FILE *stream, long offset, int origin);
```

- La `fseek()` ha tre parametri:
 - `stream`: è il file di cui vogliamo spostare il cursore.
 - `offset`: indica di quanto vogliamo spostare il cursore a partire da `origin`.
 - `origin`: indica il punto di partenza per lo spostamento.
- La `fseek()` si comporta diversamente a seconda che il file sia stato aperto in modalità binaria o testuale.
- Quando il file è aperto in modalità binaria lo spostamento è di esattamente `offset` byte misurati a partire da `origin`.
- Quando il file viene aperto in modalità testuale gli unici valori che si possono usare come `offset` sono 0 e un valore precedentemente ritornato da una chiamata a `ftell()` sullo stesso file.

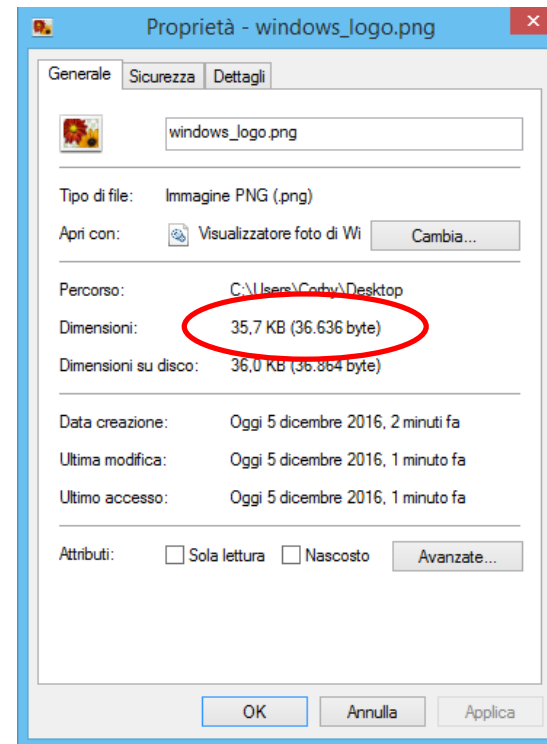
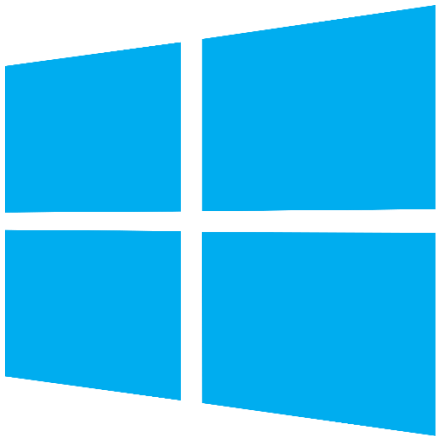
fseek()

- Per il parametro origin si possono usare tre macro definite in stdio.h:
 - SEEK_SET: indica l'inizio del file.
 - SEEK_CUR: indica la posizione corrente del cursore.
 - SEEK_END: indica la fine del file.
- Al posto di SEEK_SET è possibile utilizzare anche 0.
- Ecco alcuni esempi di chiamate a fseek():

```
FILE *f = fopen("test1.bin", "rb");  
// mi sposto alla fine del file  
fseek(f, 0, SEEK_END);  
// mi sposto all'inizio del file  
fseek(f, 0, SEEK_SET);  
// mi sposto indietro di 4 byte rispetto alla posizione corrente  
fseek(f, -4, SEEK_CUR);  
// mi sposto in avanti di 16 byte rispetto all'inizio del file  
fseek(f, 16, SEEK_SET);  
fclose(f);
```

Esempio

- La più semplice operazione che si può realizzare tramite l'uso delle funzioni appena viste è calcolare la dimensione di un file in byte.
- Proviamo a determinare la dimensione della seguente immagine:



- Il valore che vogliamo ottenere è quindi 36636.

Esempio

- Il codice potrebbe essere il seguente:

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

```
    char *filename = "windows_logo.png";
    FILE *img = fopen(filename, "rb");
```

```
    fseek(img, 0, SEEK_END);
    long size = ftell(img);
    rewind(img);
```

```
    printf("La dimensione di %s e': %ld byte", filename, size);
```

```
    // altre operazioni su img...
```

```
    fclose(img);
```

```
    return 0;
```

```
}
```

Apro il file in modalità lettura binaria.

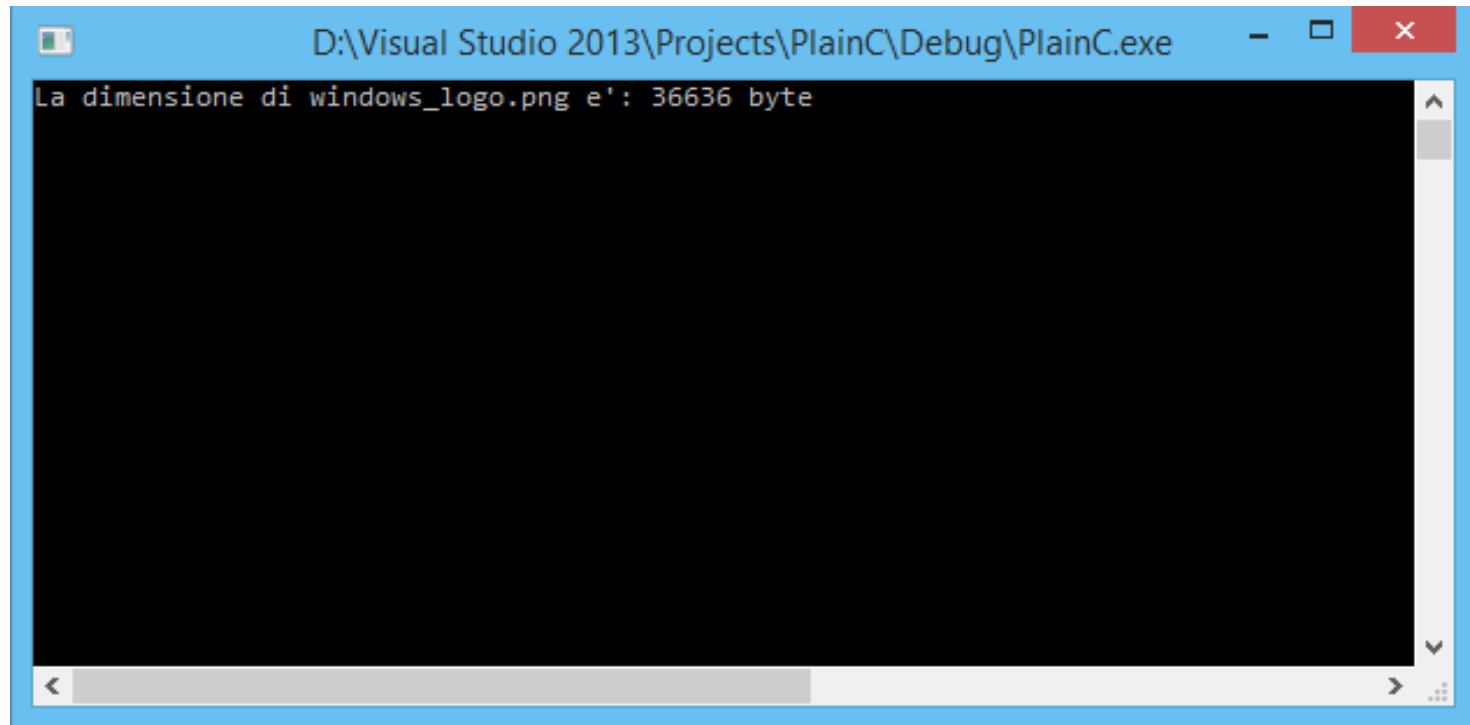
Sposto il cursore alla fine del file con una `fseek()` e con `ftell()` ottengo il numero di byte dall'inizio dello stream (quindi la dimensione del file). Infine riporto il cursore all'inizio del file con una `rewind()`.

Stampo su `stdout` il valore della dimensione.

Chiudo il file.

Esempio

- Ecco cosa è stato stampato su stdout dopo l'esecuzione:



A screenshot of a Windows command prompt window. The title bar at the top reads "D:\Visual Studio 2013\Projects\PlainC\Debug\PlainC.exe" and includes standard window controls (minimize, maximize, close). The main area of the window is black with white text. The first line of text is "La dimensione di windows_logo.png e': 36636 byte". The window has a scroll bar on the right side.

```
La dimensione di windows_logo.png e': 36636 byte
```

- Il valore è esattamente lo stesso mostrato dalla finestra delle proprietà di Windows.

fgetpos() e fsetpos()

- Esistono altre due funzioni che svolgono lo stesso compito di ftell() e fseek(), ma superano il problema di usare un long per rappresentare la posizione dello stream (qual è la massima dimensione trattabile di un file se usiamo un long?)
- Queste due funzioni sono:

```
int fgetpos (FILE * stream, fpos_t * pos);  
int fsetpos (FILE * stream, const fpos_t * pos);
```

- Queste usano un tipo apposta per rappresentare qualsiasi posizione possibile all'interno di un file, il tipo `fpos_t`.
- Il valore di ritorno in queste funzioni serve solo a indicare il verificarsi di un errore. In caso di successo ritornano 0, altrimenti 1.
- Con queste funzioni però non ci si può più spostare in posizioni arbitrarie di uno stream come con la `fseek()`, ma solamente in posizioni precedentemente ottenute da una chiamata a `fgetpos()`.

Redirezione di stdin e stdout

- È possibile fare in modo che tutto quello che viene inviato sul file stdout venga rediretto verso un altro file? Magari un file di testo sul disco.
- È possibile far sì che stdin venga rediretto verso un altro file?
- La risposta è sì.
- Questa redirezione può essere effettuata direttamente dalla linea di comando prima di lanciare il programma.

Redirezione da linea di comando

- Per effettuare questa redirezione da linea di comando si usano i caratteri maggiore e minore (> e <).
- Il maggiore ci permette di ridirigere stdout verso un altro file.
- Il minore ci permette di redirigere stdin verso un altro file.
- La sintassi è:

```
programma.exe > output.txt
```

```
programma.exe < input.txt
```

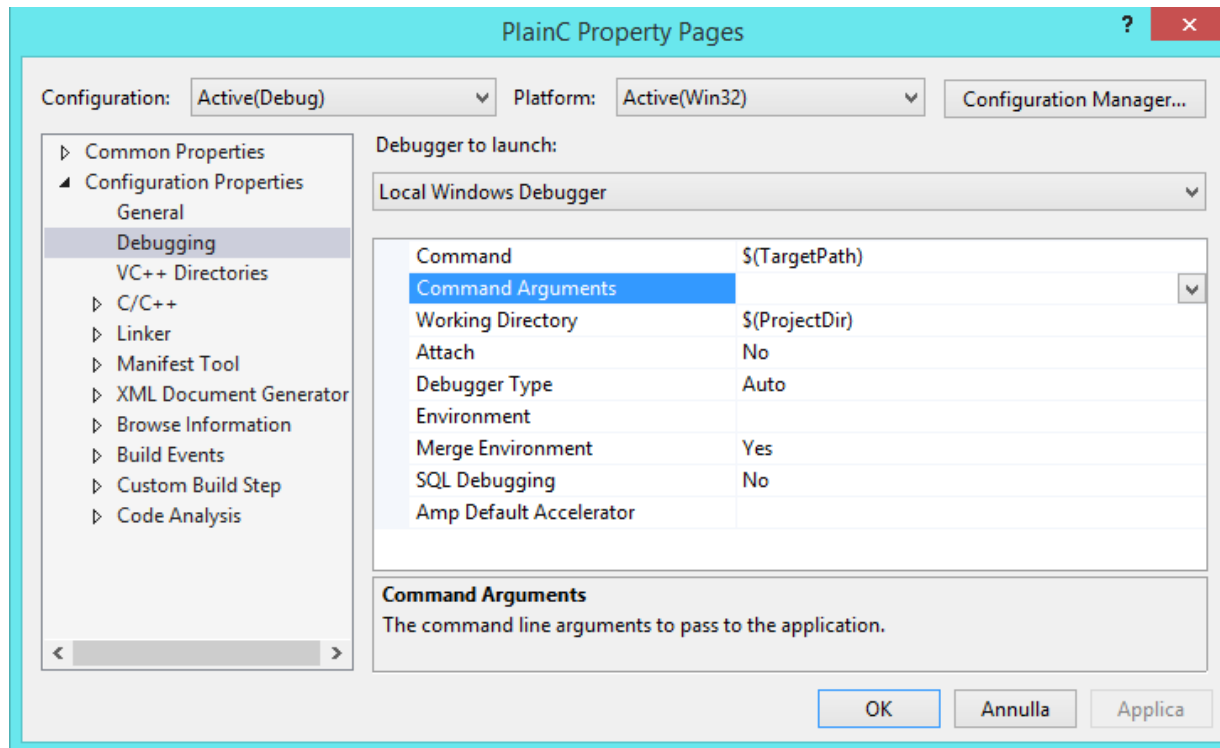
- Le due si possono anche combinare:

```
programma.exe < input.txt > output.txt
```

- In questo modo il programma verrà avviato con `input.txt` come stdin e con `output.txt` come stdout.

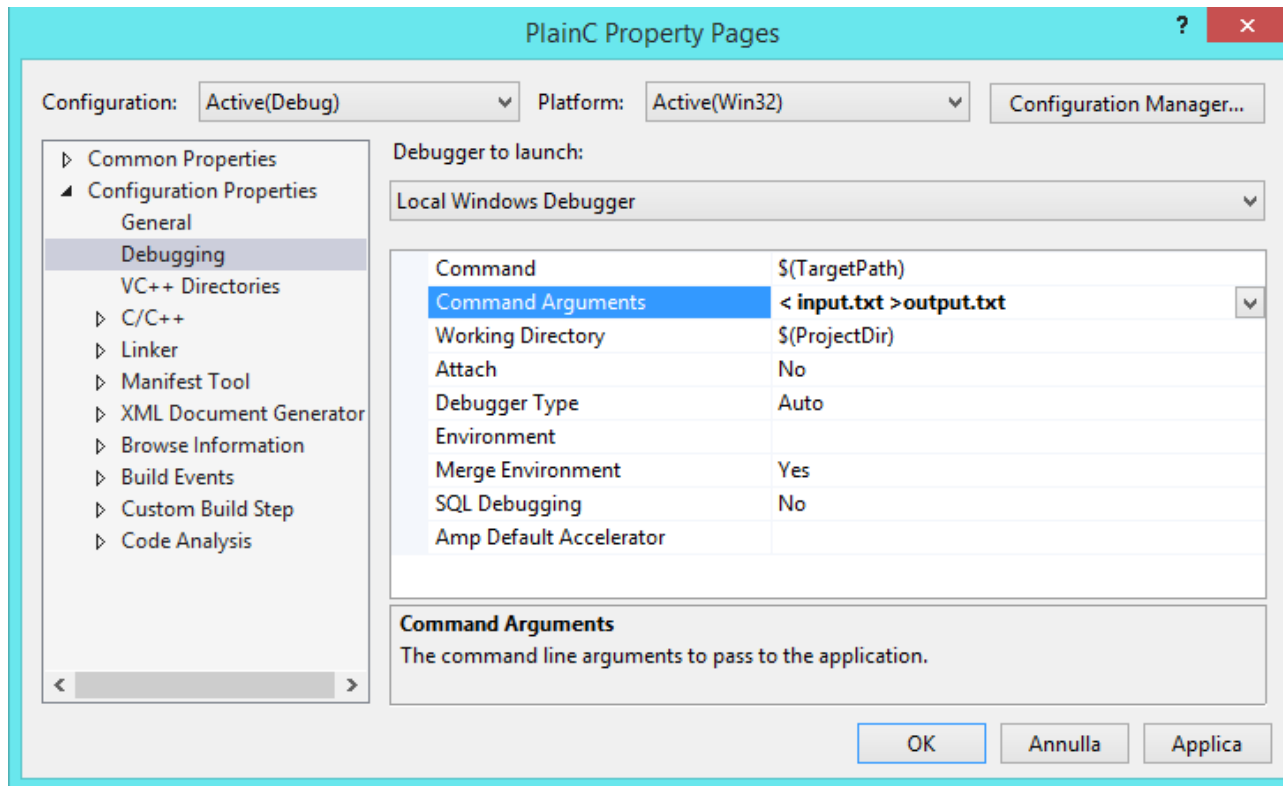
Redirezione attraverso Visual Studio

- Durante il debug con Visual Studio però non vediamo nessuna linea di comando quando premiamo F5 o F10.
- Questo accade perché Visual Studio si occupa per noi della linea di comando da usare per avviare il programma che vogliamo debuggare.
- Possiamo però aggiungere dei parametri a questa linea di comando dalla finestra della configurazione del progetto:



Redirezione attraverso Visual Studio

- Nella configurazione che si vuole testare (quindi Debug), nel menu «Configuration Properties», alla voce «Debugging» bisogna modificare l'opzione «Command Arguments»:



- Quello che viene scritto in questo parametro del progetto viene aggiunto sulla linea di comando quando viene lanciato il programma per il debug.