Sistemi Operativi

Unità 3: Programmazione in C

Operazioni sui file

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. File
2. Apertura e chiusura file
3. Lettura/scrittura su file
4. Gestione degli errori in C

File

File

In C, come in tutti i linguaggi, è possibile leggere e scrivere su file.

Esistono varie funzioni per farlo. Alcune sono strettamente legate alle **System Call** Linux, che vedremo più avanti.

I file possono essere:

- Binari: contenere sequenze di bit arbitrarie.
  - Immagini, file compressi
- Testuali: contengono solo caratteri stampabili
  - File `.txt`, `.html`, sorgenti di programmi `.c`
    Noi vedremo principalmente file testuali.

File

Le operazioni di base sono:

- Apertura di un file
- Lettura o scrittura nel file
- Chiusura del file

Vedremo le funzioni principali per queste operazioni.

- Esistono anche altre operazioni, che vedremo nel corso quando parleremo di file e file system

Apertura e chiusura file

Apertura e chiusura file

Un programma può accedere a file su disco, tramite il loro *path*.
Prima di leggere o scrivere un file, il programma deve *aprirlo*.

- Indicare al sistema operativo che accederà a tale file e in che **modalità**

La **modalità** può essere:

- Lettura
- Scrittura
- Aggiunta (o *append*)

Apertura e chiusura file

L'accesso (scrittura/lettura) ai file è **sequenziale**.
Si inizia a leggere o scrivere dall'inizio e si prosegue

- Simile all'idea di *cursore*
- Esistono funzioni per riposizionare il *cursore*, vedremo più avanti

Quando si apre un file, bisogna indicare se esso è binario o testuale.

- Se testuale, le funzioni si aspettano `\n` per delimitare le righe

Apertura e chiusura file

Per indicare un file aperto, su cui è possibile effettuare operazioni, in C si usa il tipo `FILE *`.

- Tecnicamente esso è un puntatore a una variabile di tipo `FILE`.
- Non ci interessa che tipo è `FILE`

`FILE *` è un cosidetto **handle opaco**: è un puntatore. Ma non ci interessa a cosa punta.
Solo le funzioni di libreria hanno interesse ad accedere al dato.
Pertanto il contenuto di `FILE` può cambiare o non seguire uno standard.

Apertura e chiusura file

Per aprire un file si usa la funzione `fopen(path, modo)`.
Essa ritorna un `FILE *`.

- `path` può essere assoluto o relativo.
- `modo` indica se apriamo in lettura (`r`), scrittura (`w`) o aggiunta (`a`). Di default la modalità è testuale. Per indicare modalità binaria, aggiungere `b` (es. `rb` o `wb`).

Esempio:

```
FILE * f;
f = fopen("file.txt", "r");
if (f==NULL){ /*Errore*/ }
```

In caso di errore, la `fopen` ritorna il puntatore nullo `NULL`.

Apertura e chiusura file

Note:

- In modalità scrittura e aggiunta, se il file non esiste, viene creato.
- In modalità lettura, se il file non esiste, la `fopen` ritorna `NULL`.
- In modalità scrittura, se il file esiste, il suo contenuto viene cancellato all'apertura

Apertura e chiusura file

Chiusura file: quando non si accede più a un file, bisogna *chiuderlo* e dismettere il corrispondente `FILE *`
Si chiama la funzione `fclose(file)` che accetta come argomento un `FILE *`.

- Mai chiamare la `fclose` con un `FILE *` invalido settato a `NULL`
- Si può chiudere un file aperto solo una volta

Se un file non viene chiuso, la chiusura è effettuata automaticamente dal SO quando il programma termina.

Lettura/scrittura su file

Lettura/scrittura su file

Le funzioni che si usano simili a quelle che si usano per leggere da tastiera e scrivere su console.

Lettura:

- `fgetc(file)`: legge un carattere e lo fornisce come valore di ritorno
  - Ritorna la costante `EOF` se il file è finito
- `fgets(buffer, N, file)`: legge una stringa di massimo `N` caratteri.
  - Legge fino a quando ha letto `N` caratteri o trova `\n`, che **è incluso** nella stringa ritornata e terminata da `\0`
  - Ritorna `NULL` se il file è finito

Lettura/scrittura su file

Scrittura:

- `fputc(carattere, file)`: scrive un carattere su file
- `fputs(stringa, file)`: scrive una stringa su file

Nota: queste funzioni possono leggere e scrivere anche su terminale. E' sufficiente dire loro di leggere dal file `stdin` e scrivere su file `stdout`.

- Abbiamo già visto questa funzione come valida alternativa alla insicura `gets`.

Lettura/scrittura su file

Esempio: si legga un path da tastiera e se ne stampi il contenuto come file di testo

```c
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Inserisci un path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossibile aprire %s\n", s);
        return 1; /* Errore */
    }

    /* Non è importante la lunghezza di buffer */
    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout); // Equivale a printf("%s", s);

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

Lettura/scrittura su file

Si possono usare le funzioni `fprintf` e `fscanf` che sono equivalenti a `printf` e `scanf` con la differenza che scrivono su file e non da console.

Sintassi: `fprintf(file, formato, argomenti)` e `scanf(file, formato, &argomenti)`

Queste funzioni sono particolarmente utili per leggere e scrivere numeri interi e reali

Esempio:

```
fprintf(f, "%d\n", 123); // scrive il numero 123 e un ritorno a capo
```

Lettura/scrittura su file

Esempio: si legga un numero N da tastiera e si stampi sul file `numeri.txt` i numeri da 1 a N

```c
#include <stdio.h>

int main ()
{
    int n, i;
    FILE * f;

    printf("Inserisci un numero: ");
    scanf("%d", &n);

    f = fopen("numeri.txt", "w");
    if (f==NULL){
        printf("Impossibile aprire numeri.txt\n");
        return 1;
    }

    for (i=1; i<=n; i++)
        fprintf(f, "%d\n", i);

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

Lettura/scrittura su file

In caso di lettura, abbiamo visto che `fgetc` e `fgets` hanno comportamenti diversi. In pratica, in caso di file completamente letto:

- `fgets` ritorna `EOF`
- `fgets` ritorna `NULL`
- `fscanf` ritorna `EOF`

E' possibile usare la funzione `eof(FILE *)` per verificare se il file è stato letto completamente.

- Ritorna `TRUE` / `FALSE`

Lettura/scrittura su file

Abbiamo visto che `(f/s)printf` e `(f/s)scanf` permettono di specificare il formato come `%d` `%f` `%s` `%c`.
Riassumiamo:

- Carattere `char`: `%c`
- Stringa `char []`: `%s`
- Intero `int`: `%d`
- Reale `float`: `%f`

Lista completa: https://man7.org/linux/man-pages/man3/printf.3.html

Esistono modificatori per definire numero di cifre decimali, padding per interi, ecc..

Lettura/scrittura su file

Esercizio: si legga `persone.txt` che contiene su ogni riga nome ed età di una persona, separati da `' '`. Si calcoli l'età media. Esempio di file:

```
martino 31
andrea 37
```

```c
#include <stdio.h>

int main ()
{
    int n=0, s, e;
    FILE * f;
    char nome[100];

    f = fopen("persone.txt", "r");
    if (f==NULL){
        printf("Impossibile aprire persone.txt\n");
        return 1;
    }

    /* La fscanf si aspetta di trovare su ogni riga una parola e un intero */
    while (fscanf(f, "%s %d\n", nome, &e) != EOF){
        n++; s+=e; /* Accumula i contatori */
    }

    printf("La media è %f\n", (float)s/n ); /* Notare il casting */
    return 0;
}
```

Gestione degli errori in C

Gestione degli errori in C

Problematica

Abbiamo visto che le funzioni di libreria segnalano un eventuale errore tramite il valore di ritorno

Esempio:

```
f = fopen("file.txt", "r");
if (f==NULL){
    /*Errore*/
}
```

Con questo meccanismo, non è possibile sapere niente su **quale** sia stato l'errore.

- File non esistente?
- No permessi di lettura?

Gestione degli errori in C

Funzionamento

La libreria standard del C utilizza il seguente meccanismo per specificare la causa di errore

- Ogni programma in C ha la variabile globale `int errno`
- Una funzione di libreria che fallisce, setta `errno` con un codice di errore esplicativo
- Se il chiamante, tramite il valore di ritorno, rileva se c'è stato un errore
- Se c'è stato un errore, il chiamante legge in `errno` il codice di errore

Necessario:

```
#include <errno.h>
```

Gestione degli errori in C

Gestione dell'errore

La variabile globale `errno` è intera e contiene un codice di errore.

- La pagina di manuale di ogni funzione specifica quali codice di errore può ritornare
- `fopen` può fallire con `ENOENT` (file inesistente), `EACCES` (permessi insufficienti) e molti altri

Tutti i codici di errore sono costanti definite nella libreria standard

- Il programmatore può confrontare `errno` con le costanti per identificare l'errore

Gestione degli errori in C

Gestione dell'errore

Esempio:

```
FILE * f = fopen("file.txt", "r");
if (f==NULL){
    if (errno == ENOENT)
        printf("File inesistente\n");
    else if (errno == EACCES)
        printf("Permessi insufficienti\n");
    else
        printf("Errore generico\n");
    return 1;
}
```

Gestione degli errori in C

Stampa dell'errore

Esistono delle funzioni di libreria per semplificare la gestione dell'errore.

```c
#include <stdio.h>
void perror(const char *s);
```

Stampa il messaggio di errore relativo al valore corrente di `errno`, premettendo la stringa `s`

Esempio:

```c
FILE * f = fopen("file.txt", "r");
if (f==NULL){
    perror("Error");
    return 1;
}
```

Stampa: `Error: No such file or directory`

Gestione degli errori in C

Stampa dell'errore

```c
#include <string.h>
char *strerror(int errnum);
```

Ritorna una stringa che spiega l'errore dal codice `errnum`

Esempio:

```c
FILE * f = fopen("file.txt", "r");
if (f==NULL){
    printf("Impossibile aprire il file. Errore: %s\n", strerror(errno));
    return 1;
}
```

Stampa: `Impossibile aprire il file. Errore: No such file or directory`

Gestione degli errori in C

Limiti

La gestion degli errori tramite la variabile globale `errno` è una tecnica problematica, in caso di:

- In caso di segnali (**vedremo**)
- Fortunatamente `errno` è thread safe (**vedremo**)

La gestione degli errori tramite `errno` è considerata obsoleta.

- I linguaggi più moderni usano i costrutti `try catch`