File

Corso di Programmazione di Sistema

Dr. Nicola Bicocchi

DIEF/UNIMORE

Aprile 2023

File binari e file di testo

File di testo:

- Contengono caratteri ASCII stampabili e di controllo (e.g., *newline*, *tabulazione*)
- Sono letti e scritti per caratteri o per righe (ogni riga delimitata da *newline*)
- Sono visualizzabili e manipolabili usando semplici editor di testo
- Esempi di file di testo: sorgenti C, file di configurazione Unix (/etc)

File binari:

- Non contengono solo caratteri ASCII stampabili, ma qualunque carattere
- Vengono letti e scritti in byte o blocchi di byte
- Se visualizzati con un editor di testo risultano incomprensibili
- Esempi di file binari: file eseguibili, file compressi, immagini, audio, video

File binari e file di testo

- Un file di testo è un caso particolare di file binario che utilizza un sottoinsieme dei caratteri ASCII (i caratteri stampabili). Può essere manipolato con le funzioni dedicate ai file binari; è infatti possibile leggerli a blocchi di byte come per i file binari
- Operare su file binari con le funzioni usate per i file di testo non è invece agevole

```c
/* Prints the full ASCII table */
int main(void) {
    int i;
    for (i = 0; i < 255; i++) {
        printf("[%3d] -> %c\n", i, i);
    }
}
```

File binari e file di testo

- Uno stesso dato può sempre essere memorizzato sia in file di testo che in file binari utilizzando un'opportuna rappresentazione. La scelta del tipo di rappresentazione dipende da considerazioni legate al tipo di utilizzo che viene fatto dei dati
- In un file di testo il contenuto si presenta come una stringa di caratteri mentre in un file binario si usa la codifica binaria
- Ad esempio, per memorizzare un intero a 32bit in un file binario sono sempre necessari 4 byte indipendentemente dal valore memorizzato. Al contrario, la memorizzazione in un file di testo richiede un numero di byte che dipende dal valore da memorizzare

```
1 Esempio: memorizzazione del numero decimale 214439
2 File di testo: '2' '1' '4' '4' '3' '9'                          (6 byte)
3 File di binario (in base alla endianness): 0x00 0x03 0x45 0xa7  (4 byte)
```

```
1 Esempio: memorizzazione del numero decimale 6
2 File di testo: '6'                                              (1 byte)
3 File di binario (in base alla endianness): 0x00 0x00 0x00 0x06  (4 byte)
```

Il tipo FILE

- Le funzioni per l'accesso a file sono dichiarate nel file di intestazione *stdio.h*
- Il riferimento al file desiderato viene mantenuto per mezzo di un puntatore (*FILE* *)
- FILE è una struttura che contiene tutte le informazioni utili a permettere l'interazione di un programma con i file gestiti dal sistema operativo. Tipicamente il programmatore non ha alcuna necessità di conoscere il contenuto dei campi delle variabili di tipo FILE
- Il programma opera su un file chiamando le funzioni messe a disposizione dalla libreria standard che usano le variabili di tipo FILE per identificare e manipolare il file fisico
- Un programma può aprire e usare più file contemporaneamente

struct file

```
1  struct file {
2      ...
3      struct path             f_path;
4      struct inode            *f_inode;   /* cached value */
5      const struct file_operations    *f_op;
6      ...
7      unsigned int            f_flags;
8      fmode_t                 f_mode;
9      struct mutex            f_pos_lock;
10     loff_t                  f_pos;
11     struct fown_struct      f_owner;
12     const struct cred       *f_cred;
13     struct file_ra_state    f_ra;
14     ...
15 };
```

Apertura e chiusura di file

- Per utilizzare un file è necessario sia *aprirlo* che *chiuderlo*
- Aprire un file significa comunicare al sistema operativo che si intende accedere al contenuto del file attravero l'utilizzo della funzione *fopen* (File OPEN)
- Quando il file non è più utilizzato il file deve essere chiuso utilizzando la funzione *fclose* (File CLOSE)
- Le funzioni di apertura e chiusura di file operano su variabili di tipo puntatore a FILE

```
1 FILE *fopen(char *path, char *mode);
2 int fclose(FILE *fp);
```

fopen()

```
1  FILE *fopen(char *path, char *mode);
```

- Accetta due parametri di tipo puntatore a carattere: *path* contiene il percorso del file da aprire, *mode* specifica il modo con il quale aprire il file
- La funzione *fopen()* restituisce:
  - un puntatore di tipo FILE correttamente inizializzato se l'operazione ha avuto successo
  - NULL in caso di errore
- Tipiche fonti di errore:
  - aprire in lettura un file che non esiste
  - aprire un file per il quale non si dispongono dei necessari permessi

fopen()

```
1  FILE *fopen(char *path, char *mode);
```

- Un file può essere aperto per diversi scopi: lettura, scrittura e append (scrittura alla fine del file). La stringa *mode* può quindi valere:
  - **r** (lettura)
  - **r+** (lettura e scrittura)
  - **w** (scrittura) : se il file esiste, esso viene sovrascritto, altrimenti un nuovo file viene creato
  - **w+** (lettura e scrittura) : se il file esiste, esso viene sovrascritto, altrimenti un nuovo file viene creato
  - **a** (append) : la scrittura avviene a partire dalla fine del file
  - **a+** (lettura e append) : la lettura avviene dall'inizio, mentre la scrittura è sempre effettuata alla fine del file
- E' anche possibile aggiungere il carattere *b* alla fine o in mezzo alla combinazione di caratteri che rappresentano *mode* (es., ottenendo wb o ab+) per indicare che il file va trattato come binario
- Molti sistemi trattano i file binari e i file di testo allo stesso modo, quindi il fatto di specificare la b risulta superfluo. Altri sistemi operativi potrebbero operare delle distinzioni, e quindi potrebbe essere necessario specificare esplicitamente il tipo di file trattato

fclose()

```c
int fclose(FILE *f);
```

- I file devono essere chiusi utilizzando la funzione *fclose()*
- La chiusura del file serve per liberare delle aree di memoria allocate dalle funzioni della libreria per memorizzare le informazioni lette e scritte sul file
- *fclose()* accetta come parametro il puntatore a FILE che identifica il file da chiudere

```c
int fflush(FILE *f);
```

- La funzione *fflush()* forza la scrittura di tutti i dati non ancora scritti sul dispositivo fisico
- In caso riceva come parametro un puntatore null, forza la scrittura in tutti i file aperti
- *fclose()* chiama *fflush()* prima di chiudere definitivamente i file

Esempio: apertura di un file

```c
FILE *fin, *fout;
if (!(fin = fopen("matrice.dat", "r"))) {
    perror("matrice.dat");
    exit(1);
}

if (!(fout = fopen("documenti/info.txt", "w"))) {
    perror("documenti/info.txt");
    exit(1);
}

/* ... corpo del programma ... */

fclose(fin);
fclose(fout);
```

Esempio: apertura di un file

- Il file *matrice.dat* viene aperto in lettura
- Il file *documenti/info.txt* viene aperto in scrittura
- I test verificano che i puntatori restituiti siano non nulli, cioè che i file siano stati aperti correttamente
- La funzione *perror()* (print error) viene usata per visualizzare un eventuale messaggio
  - Il messaggio descrive l'ultimo errore che si è verificato durante la chiamata ad una funzione di libreria, memorizzato all'interno della variabile *errno*
  - Se il parametro della perror è diverso da NULL, la funzione prima visualizza la stringa passatale come parametro, seguita dai due punti e dal messaggio di errore
- I file vengono chiusi con la chiamata a *fclose()*

Accesso a file binari

```c
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nelem, FILE *f);
size_t fwrite(void *ptr, size_t size, size_t nelem, FILE *f);
```

- Le funzioni *fread()* ed *fwrite()* vengono utilizzate per leggere e scrivere dati in formato binario e richiedono i seguenti parametri:
  - *ptr* puntatore all'area di memoria che contiene di dati da scrivere o in cui memorizzare i dati letti
  - *size* la dimensione del singolo elemento da leggere o scrivere
  - *nelem* il numero degli elementi da leggere o scrivere
  - *f* puntatore a FILE da cui leggere o scrivere
- Ritornano una variabile di tipo *size_t* che rappresenta il numero degli elementi effettivamente letti o scritti (utile per verificare errori o il raggiungimento della fine del file)

Esempio: copia di un file binario

```c
void copy_data(FILE *source, FILE *target) {
    unsigned long nread, nwrite;
    char buffer[4096];

    do {
        nread = nwrite = 0;
        nread = fread(buffer, 1, sizeof(buffer), source);
        if (nread > 0) {
            nwrite = fwrite(buffer, 1, nread, target);
        }
    } while ((nread > 0) && (nread == nwrite));

    if (nread != nwrite) {
        perror("copy_data()");
        exit(EXIT_FAILURE);
    }
}
```

Accesso a file di testo

```c
#include <stdio.h>
int fgetc(FILE *stream)
int fputc(int char, FILE *stream)
char *fgets(char *str, int n, FILE *stream)
int fputs(const char *str, FILE *stream)
```

- Le funzioni *fgetc()* ed *fputc()* vengono utilizzate per leggere e scrivere singoli caratteri (impacchettati all'interno di una variabile di tipo intero)
- Le funzioni *fgets()* ed *fputs()* vengono utilizzate per leggere e scrivere array di caratteri. La funzione *fgets()* è particolarmente utile in quanto consente la lettura di un file di testo orientata alla singola linea (la lettura viene interrotta in corrispondenza del carattere *newline* oppure in caso dell'esaurimento del buffer)
- Nota bene: la lettura orientata alla linea ha significato solo nel caso di file di testo. Nei file binari, il carattere a capo ('\n') non ha significato. Rappresenta solo un byte di valore 10.

Esempio: copia di un file di testo

```c
void copy_by_char(FILE *source, FILE *target) {
    int ch;
    while ((ch = fgetc(source)) != EOF)
        fputc(ch, target);
}

void copy_by_line(FILE *source, FILE *target) {
    char buffer[LINE_MAX];
    while ((fgets(buffer, LINE_MAX, source)) != NULL)
        fputs(buffer, target);
}

int main() {
    char *source_file = "example.c";
    char *target_file = "example.c.bak";

    FILE *source, *target;
    ...
}
```

Posizione corrente all'interno del file

- Dal punto di vista logico, un file è una sequenza di byte [0, size - 1]
- Quando un programma accede ad un file, in lettura o in scrittura, il sistema ricorda la sua posizione corrente (all'interno della struttura FILE)
- La posizione corrente è relativa ad un singola apertura. Un file può essere aperto contemporaneamente più volte ed avere di conseguenza molteplici posizioni correnti (ognuna annotata all'interno della variabile FILE dedicata)

```c
#include <stdio.h>
/* whence [SEEK_SET, SEEK_CUR, SEEK_END] */
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);

/* alternatives to fseek, ftell */
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

Posizione corrente all'interno del file

```
echo -n "0123456789" > test.txt
```

```
int main(void) {
    FILE *src;
    long i, length;

    if (!(src = fopen("test.txt", "r"))) {
        perror("fopen()");
        exit(EXIT_FAILURE);
    }

    fseek(src, 0L, SEEK_END);
    length = ftell(src);
    for (i = -1; i >= -length; i--) {
        fseek(src, i, SEEK_END);
        printf("%c\n", fgetc(src));
    }
}
```

Flussi standard

- In C, le azioni di scrivere su video o leggere un dato da tastiera sono assimilate rispettivamente alla scrittura del dato su file e della sua lettura da file. I file standard sono utilizzati a questo scopo
- In stdio.h sono definite tre variabili di tipo puntatore a FILE:
  - *stdin*: standard input (normalmente la tastiera)
  - *stdout*: standard output (normalmente il video)
  - *stderr*: standard error (normalmente il video)
- Vengono aperti automaticamente dal programma ed ereditati dalla shell:
  - *scrivere su standard output* equivale alla visualizzazione a video
  - *leggere da standard input* equivale alla lettura da tastiera
- Invocando l'esecuzione da linea di comando, è possibile utilizzare gli operatori di ridirezione

Lettura e scrittura formattata

```c
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

```c
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

- *printf()* e *scanf()* offrono funzionità di input (lettura) ed output (scrittura) formattate. In particolare è possibile specificare un formato all'interno del quale posizionare delle variabili.
- Ne esistono diverse versioni, caratterizzate dalla lettera iniziale, che utilizzano canali di input o di output diversi. Ad esempio, *fprintf()* stampa stringhe formattate su un file, mentre *sprintf()* su una stringa di caratteri.

argc, argv

```c
int main(void) { /* ... */ }
int main (int argc, char *argv[]) { /* ... */ }
```

- La funzione *main()* può essere invocata utilizzando due interfacce distinte in base alla necessità, o meno, di intercettare parametri passati dalla shell
- int argc: contiene il numero di stringhe inserite dall'utente a linea di comando (cardinalità del 2° argomento)
- char *argv[]: l'array che contiene le stringhe inserite dall'utente a linea di comando (ogni elemento dell'array è un puntatore a carattere)
- argv[argc] contiene un puntatore NULL (per terminare la lista di stringhe)

argc, argv

```c
int main(int argc, char **argv) {
    int i;
    printf("argc=%d\n", argc);
    for (i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    /* argv[argc] is printed as an int (NULL pointer) */
    printf("argv[%d] = %d\n", i, argv[i]);
}
```

```
./example a b c
argc=4
argv[0] = ./example
argv[1] = a
argv[2] = b
argv[3] = c
argv[4] = 0
```