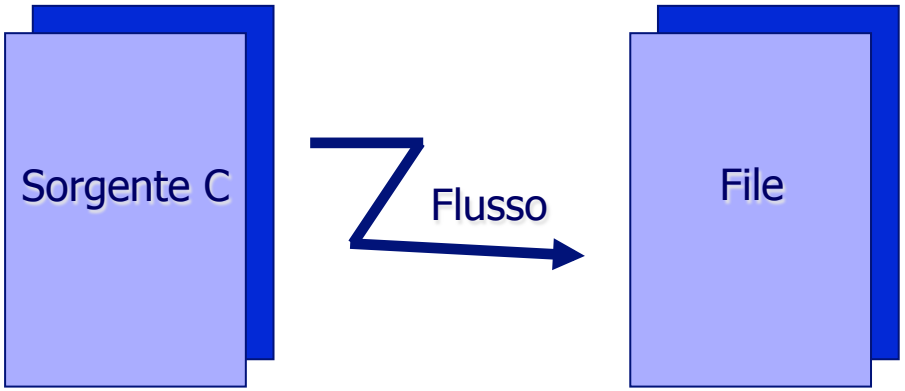
I file in C

Input e Output in C

- Input e output
  - I flussi di I/O
  - L'uso di buffer
  - Apertura e chiusura di file
  - Lettura e scrittura di dati
  - La selezione di un metodo di I/O
  - I/O non bufferizzato
  - L'accesso diretto a file

I flussi di I/O - 1

- Il linguaggio C non distingue tra periferiche e file su disco: in entrambi i casi le operazioni di I/O sono realizzate attraverso flussi o stream di I/O associati a file o periferiche
- Un flusso di I/O è una sequenza ordinata di byte: la lettura e la scrittura di un file o di una periferica implicano la lettura/ scrittura di dati da/su un flusso



I flussi di I/O: i programmi C accedono ai dati memorizzati in file attraverso array monodimensionali di caratteri, detti flussi di I/O

In stdio.h sono contenute macro, definizioni di tipo, prototipi di funzioni inerenti il mondo dei file.
E' presente la struttura FILE:

```
typedef struct {
 int cnt; /* dimensione della parte di */
          /* buffer inutilizzata */
 unsigned char *b_ptr ; /* prossima locazione */
                  /* del buffer */
 unsigned char *base;   /* inizio buffer */
 int bufsize;          /* dimensione del buffer */
 short flag;     /* informazioni varie (bit a bit) */
 char fd;              /* descrittore del file */
} FILE;
```

e il vettore _iob

```
extern FILE _iob[];
```

I flussi di I/O - 3

- Le strutture **`FILE`** forniscono al sistema operativo le informazioni necessarie per la gestione dei file: l'unico meccanismo di accesso ad un flusso è il puntatore alla struttura **`FILE`**, detto puntatore a file
- Il puntatore a file deve essere dichiarato nel programma, contiene l'identificatore del flusso restituito da una chiamata alla **`fopen()`**, e viene utilizzato per leggere, scrivere e chiudere il flusso
- Ogni programma può aprire più flussi simultaneamente, nel rispetto dei limiti imposti dalla particolare implementazione

I flussi di I/O - 4

- Uno dei campi della struttura **`FILE`** è un indicatore di posizione nel file, che punta al successivo byte da leggere o scrivere: a fronte di operazioni di lettura/scrittura, il sistema operativo modifica conseguentemente l'indicatore di posizione
- L'indicatore di posizione del file non può essere manipolato direttamente (in maniera portabile), ma può essere letto e modificato tramite funzioni di libreria, permettendo accessi al flusso non sequenziali

I flussi standard

- Esistono tre flussi standard, che vengono aperti automaticamente per ogni programma: `stdin`, `stdout`, `stderr`
- I flussi standard sono connessi al terminale, per default, ma molti sistemi operativi ne permettono la redirezione (ad es., è possibile inviare messaggi di errore ad un file per effettuare diagnostica)

La gestione degli errori

- Ogni funzione di I/O restituisce un valore speciale in caso di errore: alcune restituiscono zero, altre un valore diverso da zero o **EOF**
- Per ogni flusso aperto, esistono due campi booleani nella struttura **FILE** che registrano condizioni di errore o di fine file
- Si può accedere ai campi di fine file e di errore utilizzando le funzioni **feof()** e **ferror()**
- La funzione **clearerr()** pone entrambi i campi a zero

```
#include <stdio.h>
#define ERR_FLAG 1
#define EOF_FLAG 2

char stream_stat(fp)
FILE *fp;
{
/* se nessun campo booleano è alterato,
 * stat vale 0; se il solo campo di errore
 * è alterato, stat vale 1; se il solo
 * campo di fine file è alterato, stat
 * vale 2; se sono alterati entrambi, stat
 * vale 3
 */
        char stat = 0;

        if (ferror(fp))
           stat |= ERR_FLAG;
        if (feof(fp))
           stat |= EOF_FLAG;
        clearerr(fp);
        return stat;
}
```

I formati testo e binario

- In C i file vengono distinti in due categorie:
  - file di testo, trattati come sequenze di caratteri. organizzati in linee (ciascuna terminata da '\n')
  - file binari, visti come sequenze di bit

I formati testo e binario

- Accesso ai dati in modalità testo o binaria:
  - Un flusso testuale è composto da una sequenza di linee, concluse da newline (sistemi operativi diversi possono memorizzare linee con formati diversi, utilizzando un carattere differente di terminazione linea)
  - I flussi standard sono testuali
  - In formato binario il compilatore non effettua alcuna interpretazione dei byte: i bit sono letti e scritti come un flusso continuo
  - I flussi binari sono utilizzati quando è fondamentale preservare l'esatto contenuto del file

L'uso del buffer - 1

- Un buffer è un'area di memoria in cui i dati sono memorizzati temporaneamente, prima di essere inviati a destinazione
- Mediante l'uso di buffer, il sistema operativo può limitare il numero di accessi effettivi alla memoria di massa
- Tutti i sistemi operativi utilizzano buffer per leggere/scrivere su unità di I/O: l'accesso a disco, per esempio, avviene con "granularità di blocco", con blocchi di dimensione 512/4096 byte

L’uso del buffer - 2

- Le librerie di run-time del C contengono un ulteriore livello di bufferizzazione che può assumere due forme distinte: bufferizzazione a linee e bufferizzazione a blocchi
  - Nella bufferizzazione a linee, il sistema immagazzina i caratteri fino a quando si incontra un newline (o il buffer è pieno), inviando l’intera linea al sistema operativo (ciò che accade per l’inserimento da tastiera)
  - Nella bufferizzazione a blocchi, il sistema immagazzina i caratteri fino a riempire un blocco, trasferendolo quindi al sistema operativo

L'uso del buffer - 3

- Le librerie standard di I/O del C comprendono un gestore di buffer che mantiene il buffer in memoria il più a lungo possibile: se si accede alla stessa porzione di un flusso più volte, si ha alta probabilità che il flusso sia stato mantenuto nella memoria centrale (si possono verificare problemi di accesso concorrente, gestibili via sistema operativo, se il file è condiviso da più processi)
- Sia nel caso di bufferizzazione a righe che a blocchi, è possibile richiedere esplicitamente al sistema operativo di forzare l'invio del buffer a destinazione in un momento qualsiasi, per mezzo della funzione `fflush()`
- Il C consente di personalizzare il meccanismo di bufferizzazione (modificando le dimensioni del buffer) fino ad eliminarla, ponendo la dimensione del buffer a zero

Apertura e chiusura di file - 1

- Prima di poter accedere al contenuto di un file, è necessario aprirlo tramite la funzione `fopen()`, che prevede due parametri: il nome (assoluto o relativo) del file nel file system e la modalità di accesso

```
FILE *fopen(char *name, char *mode);
```

Modalita' di accesso

"r"	Apre un file testuale esistente in lettura, posizionandosi all'inizio del file; se il file non esiste, la funzione ritornerà il codice di errore NULL
"w"	Crea un nuovo file testuale e lo apre in scrittura, posizionandosi all'inizio del file; se il file esiste, i dati precedenti vengono eliminati
"a"	Apre un file testuale esistente in modalità append ; la scrittura può avvenire solo alla fine del file; se il file non esiste verrà creato automaticamente, in caso contrario il contenuto del file preesistente verrà mantenuto
"r+"	Apre un file testuale esistente in lettura e scrittura, posizionandosi all'inizio del file; se il file non esiste, la funzione ritornerà il codice di errore NULL
"w+"	Crea un nuovo file testuale e lo apre in lettura e scrittura
"a+"	Apre un file testuale esistente o ne apre uno nuovo in modalità append; la lettura può avvenire in una posizione qualsiasi del file, la scrittura solo alla fine

Apertura e chiusura di file - 2

- Le modalità binarie differiscono per l'aggiunta di una **`b`** (es., **`rb`**)
- La funzione **`fopen()`** restituisce un puntatore a file, utilizzabile per accedere successivamente al file aperto

Proprietà di file e flussi rispetto alle modalità di apertura della **`fopen()`**

	r	w	a	r+	w+	a+
Il file deve esistere prima dell'apertura	*			*		
Il file preesistente viene reinizializzato		*			*	
Possibilità di lettura del flusso	*			*	*	*
Possibilità di scrittura sul flusso		*	*	*	*	*
Possibilità di scrittura solo alla fine			*			*

Apertura e chiusura di file - 5

- La chiusura del file forza anche la scrittura del contenuto del buffer associato al flusso
- Dato che tutti i sistemi operativi stabiliscono un numero massimo di flussi aperti contemporaneamente, è buona norma chiudere i file quando se ne è conclusa l'elaborazione
- Tutti i flussi aperti vengono comunque chiusi dal sistema operativo quando il programma termina correttamente (nel caso di terminazione anomala, il comportamento non è standardizzato)

Lettura e scrittura per caratteri - 1

- Esistono due modalità per la lettura/scrittura di caratteri da un flusso
  - `getc()` : macro che legge un carattere da un flusso
  - `fgetc()` : analoga a `getc()`, ma realizzata come una funzione
  - `putc()` : macro che scrive un carattere su un flusso
  - `fputc()` : analoga a `putc()`, ma realizzata come una funzione

GESTIONE DEGLI ERRORI
SERVONO PER VERIFICARE E GESTIRE SITUAZIONI DI ERRORE

Funzioni

int ferror(FILE *FP)

restituisce il valore 0 se non è stato commesso nessun errore nella precedente operazione di lettura o scrittura sul file FP altrimenti restituisce un valore maggiore di 0

int feof(FILE *FP)

restituisce il valore 0 se non è stata raggiunta la fine del file nella precedente operazione di lettura o scrittura sul file FP altrimenti restituisce un valore maggiore di 0

void clearerr(FILE *FP)

azzera gli errori e la condizione di fine file

SUMMARY:Lettura da un file ad accesso sequenziale

- Lettura
  - Crea un puntatore `FILE`, collegarlo al file da leggere
    ```
    cfPtr = fopen( “clients.dat", "r" );
    ```
  - Utilizzare la `fscanf` per leggere dal file
    - Come la `scanf`, eccetto che il primo argomento è un puntatore `FILE`

    ```
    fscanf( cfPtr, "%d%s%f", &accounnt, name, &balance );
    ```
  - Dati letti dall’inizio alla fine
  - Puntatore di posizione del file
    - Indica il numero del prossimo byte da leggere/scivere
    - Non è veramente un puntatore, ma un valore intero (specifica la locazione in byte)
    - Detto anche byte offset
  - `rewind( cfPtr )`
    - Riposizionamento del punatatore posizione all’inizio del file (byte 0)

Lettura e scrittura di dati

- Su un file aperto è possibile utilizzare il puntatore a file per svolgere operazioni di lettura e scrittura
- Le operazioni di accesso al file possono essere effettuate su oggetti di granularità diversa, in particolare a livello di…
  - …carattere
  - …linea
  - …blocco
- Qualsiasi sia la granularità, è impossibile leggere da un flusso e quindi scrivere sullo stesso flusso senza che fra le due operazioni venga effettuata una chiamata a `fseek()`, `rewind()` o `fflush()`
- Le funzioni `fseek()`, `rewind()` e `fflush()` sono le uniche funzioni di I/O che forzano la scrittura del buffer sul flusso

Lettura e scrittura per caratteri - 2

- Esempio: Funzione che copia il contenuto di un file in un altro
- Note:
  - Accesso ad entrambi i file vengono in modalità binaria
  - `getc()` legge il prossimo carattere dal flusso specificato e sposta l'indicatore di posizione del file avanti di un elemento ad ogni chiamata
  - In modalità binaria, non è possibile interpretare il valore di ritorno della `getc()` per decretare la fine del file: `feof()` invece non presenta ambiguità

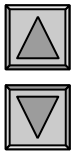
```
#include <stdio.h>
#include <stddef.h>
#define FAIL 0
#define SUCCESS 1

int copy_file(infile, outfile)
char *infile, *outfile;

{
   FILE *fp1, *fp2;
   if ((fp1 = fopen(infile, "rb")) == NULL)
        return FAIL;
   if ((fp2 = fopen(outfile, "wb")) == NULL)
   {
        fclose(fp1);
        return FAIL;
   }
   while (!feof(fp1))
        putc(getc(fp1), fp2);
   fclose(fp1);
   fclose(fp2);
   return SUCCESS;
}
```

- Inoltre `fgets()` permette la specifica del numero massimo dei caratteri da leggere, mentre `gets()` procede sempre fino ad un terminatore (newline o `EOF`)

Lettura/scrittura di stringhe - Funzioni simili a gets e puts:



char *fgets (char *s, int n, FILE *fp);

- Trasferisce nella stringa s i caratteri letti dal file puntato da fp, fino a quando ha letto n-1 caratteri, oppure ha incontrato un newline, oppure la fine del file. La fgets mantiene il newline nella stringa s.
- Restituisce la stringa letta in caso di corretta terminazione; '\0' in caso di errore o fine del file.

int *fputs (char *s, FILE *fp);

- Trasferisce la stringa s (terminata da '\0') nel file puntato da fp. Non copia il carattere terminatore '\0' ne` aggiunge un newline finale.
- Restituisce l'ultimo carattere scritto in caso di terminazione corretta; EOF altrimenti.

Un file binario è una sequenza di byte

- Può essere usato per archiviare su memoria di massa qualunque tipo di informazione
- Input e output avvengono sotto forma di una sequenza di byte
- La fine del file è SEMPRE rilevata in base all'esito delle operazioni di lettura
  - non c'è EOF, perché un file binario non è una sequenza di caratteri
  - qualsiasi byte si scegliesse come marcatore, potrebbe sempre capitare nella sequenza

Lettura di file binari

int **fread** (void *vet, int size, int n, FILE *fp);

• legge (al piu`) n oggetti dal file puntato da fp, collocandoli nel vettore vet, ciascuno di dimensione size.

Restituisce un intero che rappresenta il numero di oggetti effettivamente letti.

ESEMPIO 4: OUTPUT DI NUMERI

L'uso di file binari consente di rendere evidente la differenza fra la ***rappresentazione interna*** di un numero e la sua ***rappresentazione esterna*** come *stringa di caratteri in una certa base*

- Supponiamo che sia `int x = 31466;`
- Che differenza c'è fra:

```
fprintf(file,"%d", x);
fwrite(&x, sizeof(int), 1, file);
```

ESEMPIO 4: OUTPUT DI NUMERI

Se **x** è un intero che vale **31466**, internamente la sua rappresentazione è (su 16 bit): `01111010 11101010`

- **`fwrite()` emette direttamente tale sequenza,** scrivendo quindi i ***due byte*** sopra indicati
- **`fprintf()`** invece **emette la sequenza di caratteri ASCII** corrispondenti alla rappresentazione esterna del numero 31466, ossia i ***cinque byte***

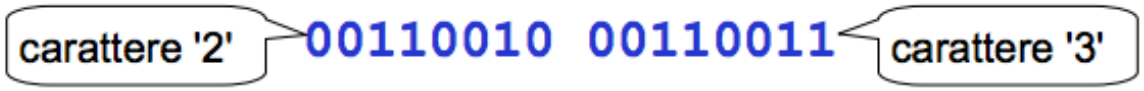
`00110011 00110001 00110100 00110110 00110110`

Se **per sbaglio** si emettessero **su un file di testo** (o su video) direttamente i due byte: `01111010 11101010`

si otterrebbero *i caratteri corrispondenti al codice ASCII di quei byte:* **êz**

ESEMPIO 5: INPUT DI NUMERI

`fscanf()` preleva la **stringa di caratteri ASCII**

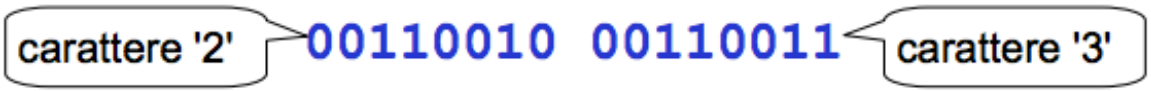


che costituisce la rappresentazione esterna del numero, e la **converte** nella corrispondente rappresentazione interna, ottenendo i due byte:

`00000000 00010111`

che rappresentano in binario il valore ***ventitre***

`fread()` invece **preleverebbe i due byte**



credendoli già la rappresentazione interna di un numero, senza fare alcuna conversione

Tale modo di agire porterebbe a inserire nella variabile x *esattamente la sequenza di byte sopra indicata*, che verrebbe quindi interpretata come il numero ***tredicimilacentosei***

La selezione di un metodo di I/O - 1

- Le macro **`putc()`** e **`getc()`** sono le più veloci, ma la maggior parte dei sistemi operativi è in grado di realizzare operazioni di I/O su blocchi ancora più efficienti (ad es., **`read()`** e **`write()`** in UNIX)
- Talvolta occorre privilegiare la semplicità all'efficienza: **`fgets()`** e **`fputs()`**, ad esempio, sono lente, ma particolarmente adatte nei casi in cui sia necessario analizzare linee

Funzione che conta il numero di linee di un file

```
#include <stdio.h>
#include <stddef.h>
#define MAX_LINE_SIZE 120

int lines_in_file(fp)
FILE *fp;
{
   char buf[MAX_LINE_SIZE];
   int line_num = 0;

   rewind(fp); /* sposta l'indicatore di
                  posizione all'inizio
                  del file */

   while (fgets(buf,MAX_LINE_SIZE,fp)!= NULL)
          line_num++;
   return line_num;
}
```

La selezione di un metodo di I/O - 2

- Ultimo, ma non meno importante, fattore da considerare nella scelta di un metodo di I/O è la portabilità, fondamentale non tanto nella scelta del tipo di I/O (granularità a caratteri, linee o blocchi), ma nella scelta della modalità testo o binaria
  - Se il file contiene dati testuali (codice sorgente o documenti), la modalità testo e l'accesso per linee sono da privilegiare
  - Se i dati sono numerici e non sono strutturati per linee, è preferibile la modalità binaria, con accesso per caratteri o per blocchi (codice eseguibile)

L'accesso diretto a file - 1

- In C, le funzioni per l'accesso diretto a file sono **`fseek()`** e **`ftell()`**
- La funzione **`fseek()`** sposta l'indicatore di posizione del file a un carattere specificato nel flusso
- Il prototipo della **`fseek()`** è

```
int fseek(FILE *stream,long int offset,int whence)
```

dove:

  - **`stream`** : puntatore a file
  - **`offset`** : numero di caratteri di spostamento
  - **`whence`** : posizione di partenza da cui calcolare lo spostamento
- L'argomento **`whence`** può assumere uno dei tre seguenti valori, definiti in **`stdio.h`**
  - **`SEEK_SET`** : inizio del file
  - **`SEEK_CUR`** : posizione corrente dell'indicatore
  - **`SEEK_END`** : fine del file

- Per flussi binari, lo spostamento può essere un qualsiasi numero intero che non sposti l'indicatore al di fuori del file; per flussi testuali, deve essere zero o un valore restituito dalla `ftell()`

L'accesso diretto a file - 3

- La funzione **`ftell()`** richiede, come unico argomento, un puntatore a file e restituisce la posizione corrente dell'indicatore di posizione nel file
- La posizione restituita da **`ftell()`** si intende relativa all'inizio del file…
  - …per flussi binari rappresenta il numero di caratteri dall'inizio del file alla posizione corrente
  - …per flussi testuali rappresenta un valore dipendente dall'implementazione, significativo solo se utilizzato come parametro per la **`fseek()`**

```
cur_pos = ftell(fp);
if (search(string) == FAIL)
   fseek(fp,cur_pos,SEEK_set);
```

Se la ricerca di una certa stringa nel file fallisce, l'indicatore di posizione nel file viene riportato al valore originale