

- 9° Modulo

- Files

- Un modo per estendere la memoria all'infinito
- I FILE sequenziali in C
- File Testuali

- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_

	<b>Politecnico di Milano</b> <b>Informatica C</b> <small>06/0025</small>
	Facoltà di Ingegneria IV- Milano Bovisio INGEGNERIA AEROSPAZIALE [P-Z] Anno Accademico 2003-04 - secondo semestre
	Prof. <b>P. Martucci</b> - M. Mussini
<small>Ultimo aggiornamento: sabato 28 febbraio 2004 11:18:59</small>	

<http://www.elet.polimi.it/upload/martucci/index.html>

# I File

- Generalità sui File
- I File Sequenziali
- I File in C
- Apertura e Chiusura
- Manipolazione
- R/W a carattere,

## I File

Il file e' l'unita' logica di memorizzazione dei dati su memoria di massa.

- Consente una **memorizzazione persistente** dei dati, **non limitata** dalle dimensioni della memoria centrale.
- Generalmente un file e' una sequenza di componenti omogenee (**record logici**).
- I file sono gestiti dal Sistema Operativo. Per ogni versione C esistono funzioni per il trattamento dei file (**Standard Library**) che tengono conto delle funzionalita' del S.O ospite.

### 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

## File di testo

Sono file di caratteri, organizzati in linee.

Ogni linea e' terminata da una marca di fine linea (**newline**, carattere '\n').

```
Egregio Sig. Rossi,
                    con la presente
le rendo noto di aver provveduto
...
```

- ➔ Il **record logico** puo' essere il singolo carattere oppure la singola linea.

# File Access Methods

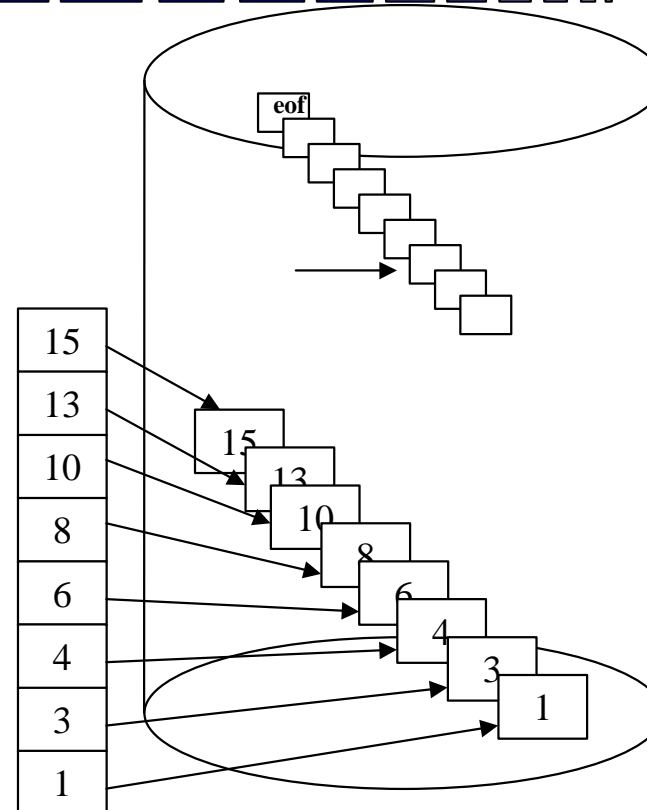
## □ Sequential

- elaborare dal 1° all'ultimo
- input ordinata (read only)
- 
- aggiornamento ordinato

## □ Random

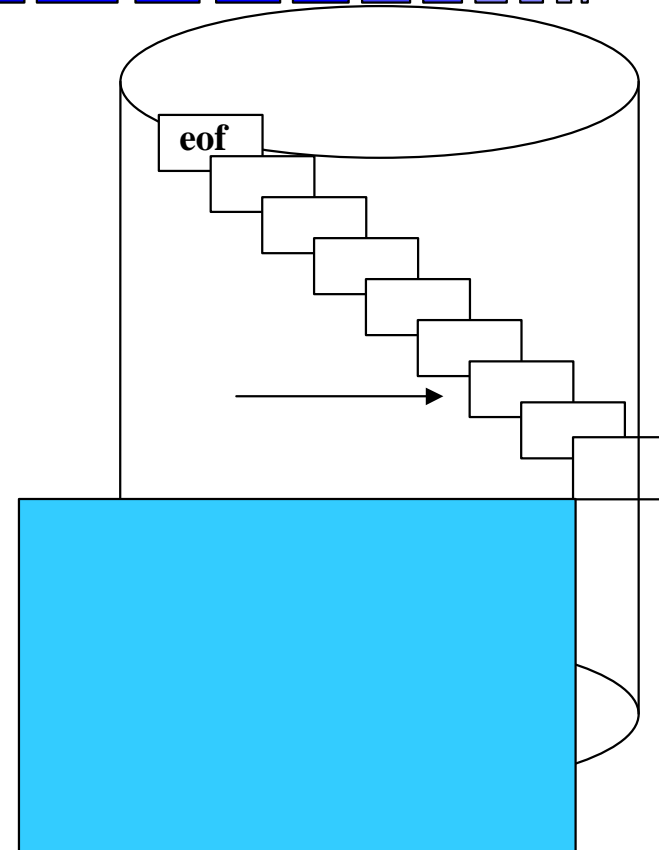
- con relativi record #
- Hashing
- via key field (index)
  - » primary (unique)
  - » secondary
  - » duplicate keys

Index  
array



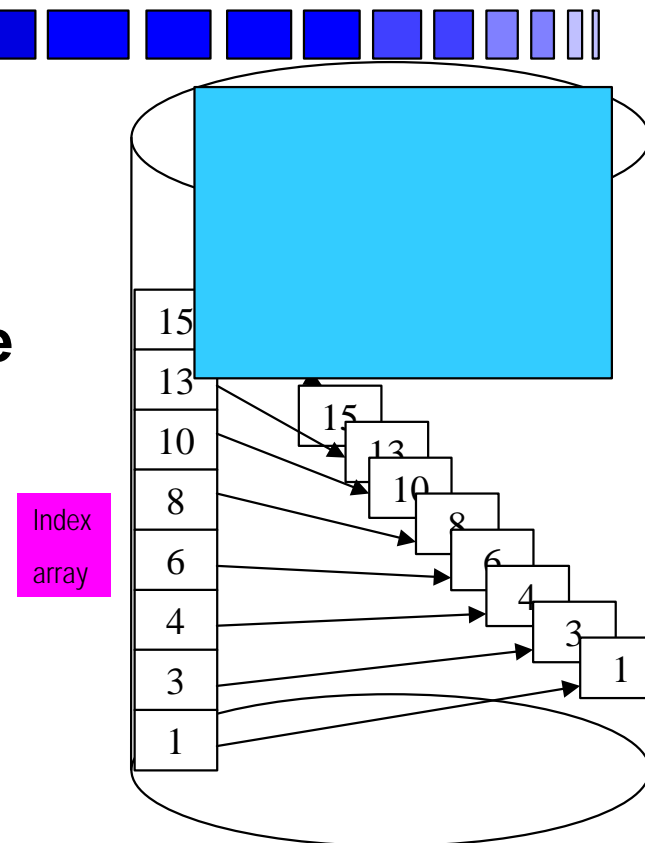
## File Sequenziali

- ❑ Tutti gli elementi (record) di ugual tipo
- ❑ Scrittura sempre in coda
- ❑ lettura dall'inizio sino all'EOF
- ❑ Termine determinato dall'End Of File (tappo)

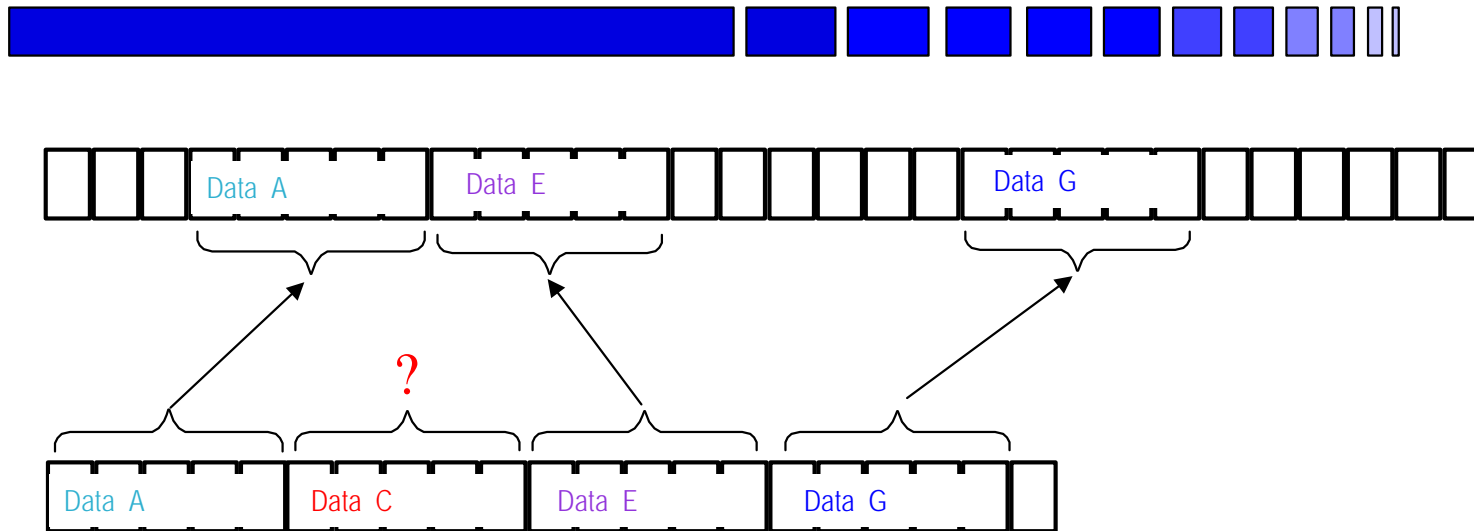


## File Random

- ❑ Elementi (record) anche diversi (dipende)
- ❑ Uso di uno o più INDICI in tabelle esterne
- ❑ Non necessario l'EOF
- ❑ Lettura e scrittura in qualunque posizione

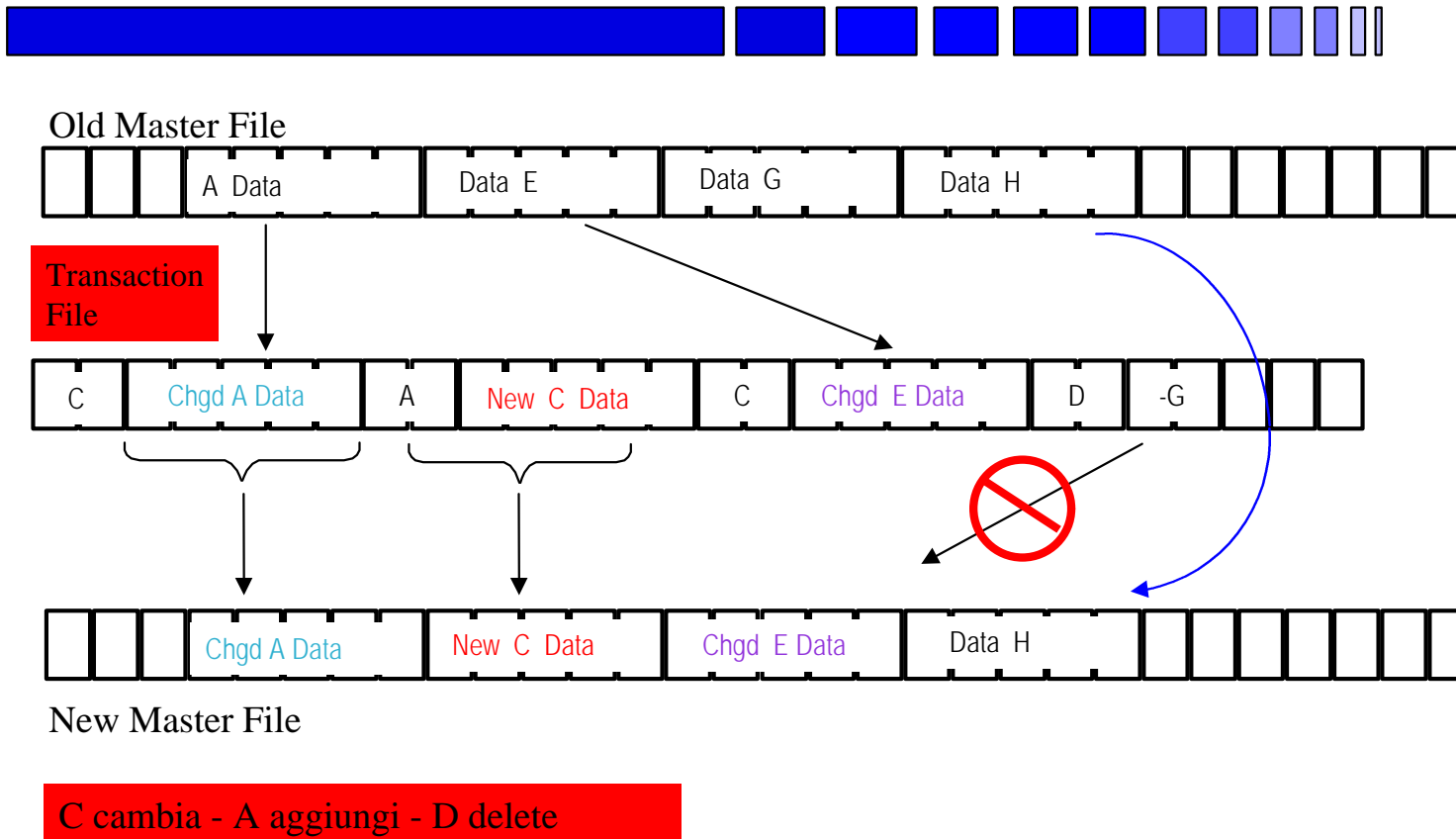


# Sequential File Updating



Updating in place is **possible**, but it presents problems!

# Sequential File Updating



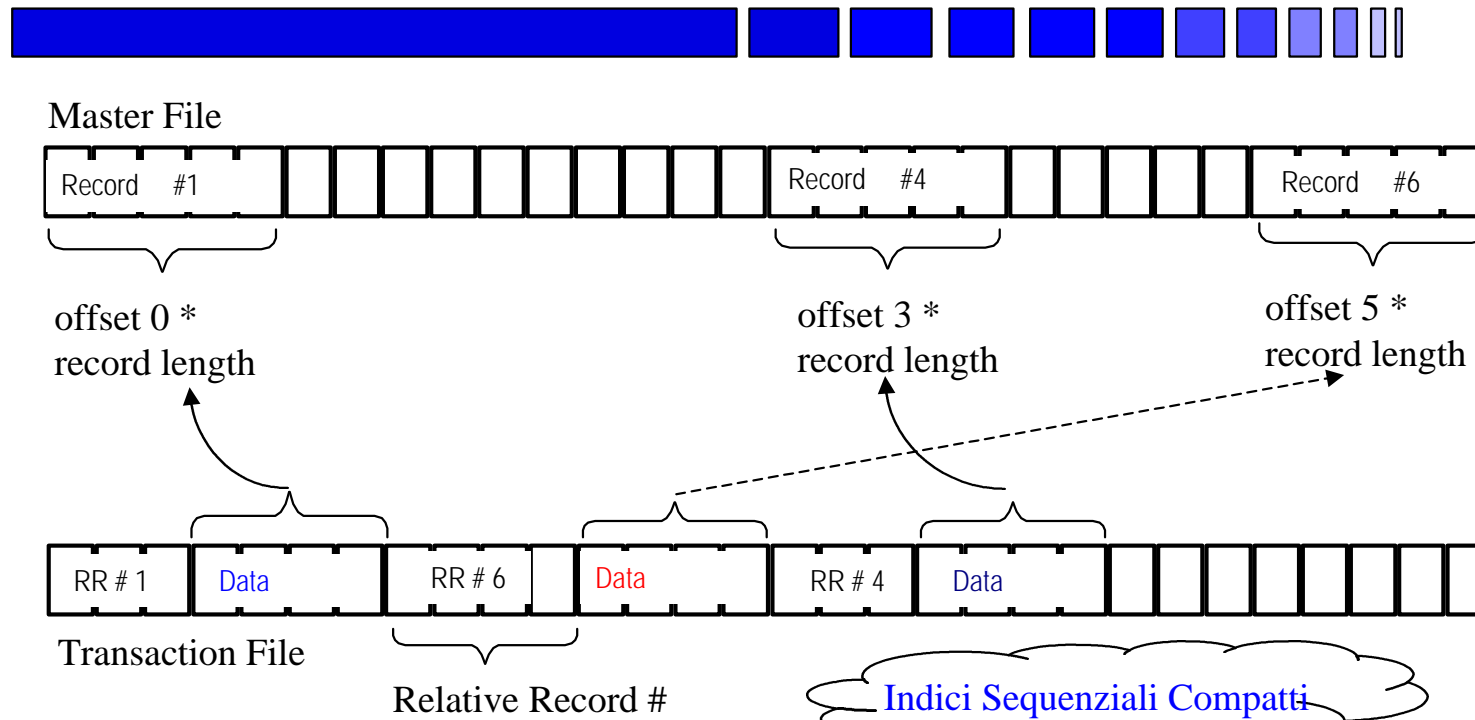


## Random Access (1)- Relative Records



- ❑ simple form of random access
- ❑ Need to know size of “record”
  - Used with only **fixed-length** records
- ❑ Relative record number
  - Used to compute offset into file
  - represents multiple of record length
  - record # “relative” to beginning of file
  - **need not be stored** as data item in record
- ❑ may be used in conjunction with true “key” to access data record

# Random Access (1) - Relative Records



e.g., if record size (length) is 120 bytes:  
 for relative record #6, offset = (record # - 1) \* record length = (6 - 1) \* 120 = 600

# I File in C

Faremo SOLO i File  
Sequenziali 😊

# FILE E PROGRAMMI C

In C l'accesso a file da parte di un programma avviene tramite le funzioni disponibili nella Standard Library (come tutta l'ingresso e uscita). In particolare, le funzioni su file sono definite in `stdio.h`.

Le funzioni di libreria della `stdio.h` interagiscono con il sistema operativo per consentire l'accesso a file. Questo vuol dire che al loro interno le **funzioni di libreria** contengono delle chiamate a **funzioni di sistema operativo**.

**programma C** → **funzioni di stdio.h** → **funzioni di Sistema Operativo**

In ambiente C per **utilizzare un file** all'interno di un programma è necessario:

1. **aprire** un «flusso di comunicazione» (aprire il file)
2. **accedere** a file in lettura e/o scrittura
3. **chiudere** il «flusso di comunicazione» (chiudere il file)

Tipi di flussi di comunicazione («tipi» di file)

- **binari:** sequenza di byte
- **testuali:** sequenza di caratteri: ciascun byte è la codifica ASCII di un carattere alfanumerico (in particolare, alcuni caratteri possono essere *non printable* e quindi di controllo)

Se un flusso di comunicazione viene aperto in un certo modo (binario o testuale), le operazioni sul file corrispondente devono essere effettuate in modo congruente, e l'utilizzo successivo deve tener conto del tipo flusso.

# Files in C

---

---

- A file descriptor, or file pointer, is an abstraction in C. The library function **fopen** returns a file **pointer**, which you can then use in your program. Afterward, you must “**fclose**” the file pointer.

Declare a variable to be of type **FILE \***, for example:

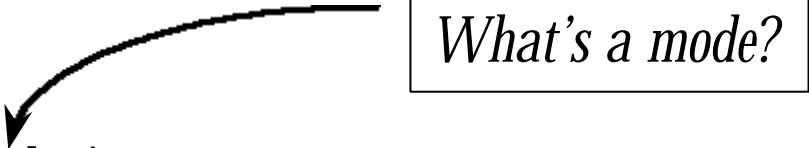
```
FILE * fd;
```

```
fd = fopen("filename", "mode");
```

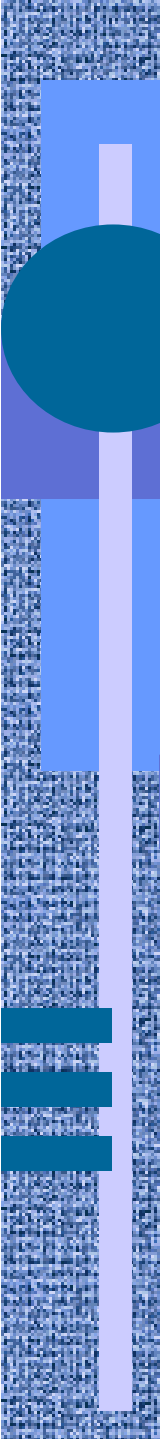
```
if (NULL == fd) /* if fopen fails, it returns NULL */  
{  
    printf("error message");  
    exit(EXIT_FAILURE);  
}
```

.....reads and writes happens here!!!

```
fclose(fd);
```



*What's a mode?*



# Modello dell'Ambiente File

# APERTURA E CHIUSURA DI UN FLUSSO DI COMUNICAZIONE (FILE) IN C

L'insieme dei file aperti da un programma in ambiente C può essere rappresentato da una **struttura dati gestita dal Sistema Operativo**. Questa struttura dati può essere pensata come

**FILE**      **TabFileAperti[NumMaxFile]**

cioè come un array (tabella) di NumMaxFile elementi dove ciascun elemento è di **tipo** FILE.

FILE è da intendersi come un tipo strutturato che rappresenta il tipo del **descrittore del file**. I campi più significativi sono:

- nome del file
- modalità di utilizzo
- posizione corrente sul file (prossimo byte a cui accedere)
- indicatore di fine file
- indicatore di errore

In `stdio.h` sono definiti tra gli altri:

- l'identificatore simbolico FILE, che viene usato da un programma per indicare il tipo associato (al descrittore di) file
- l'identificatore simbolico EOF per indicare la fine del file
- l'identificatore simbolico NULL (***spostato in* `stddef.h`**)
- i prototipi di tutte le funzioni per accesso a file

# 1. APERTURA DI UN FILE

**FILE** \* **fopen**(<nome\_file>, <modo>)

- è una **funzione** che riceve in ingresso il nome del file da aprire (specificato secondo le convenzioni del SO) e il modo con cui si vuole aprire il file.
- restituisce il puntatore al descrittore di file creato

## **FUNZIONAMENTO:**

alla chiamata, il SO (che viene attivato in modo opportuno) crea un nuovo elemento nella struttura dati che costituisce la tabella dei file aperti (crea un nuovo descrittore di file), inizializza i campi del descrittore e restituisce il puntatore a tale elemento. Quindi:

- è il SO che gestisce la struttura dati e i contenuti di tutti gli elementi
- a livello di programma C un file viene visto come un puntatore (è rappresentato da un puntatore). Un programma C che utilizza un file deve dichiarare una variabile puntatore, ad esempio

**FILE** \* **fp**;

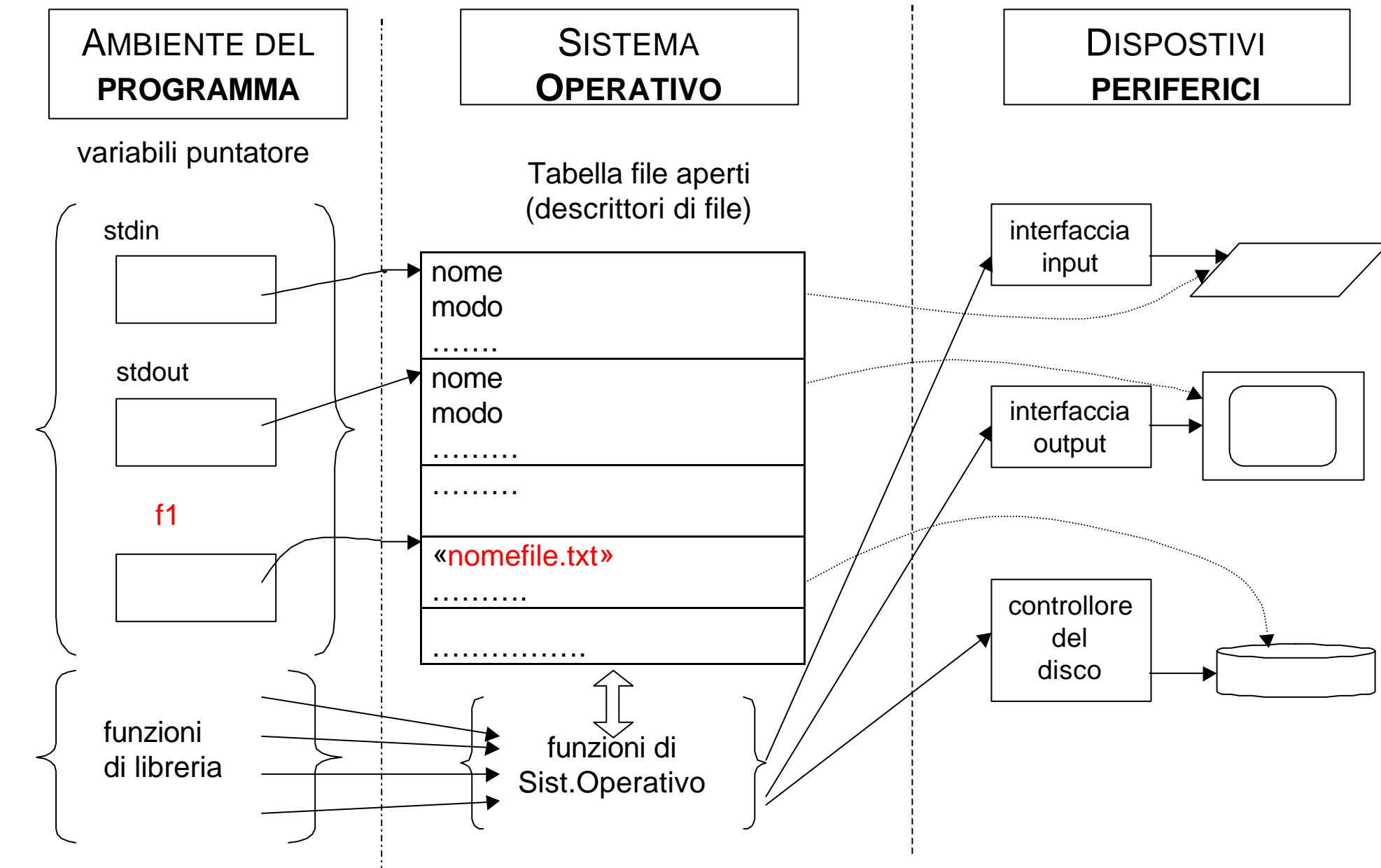
e far riferimento al file nel suo complesso tramite tale variabile (**fp**).



All'inizio dell'esecuzione di un programma C vengono automaticamente aperti **3 (4) flussi di comunicazione standard** rappresentati da 3 (4) «variabili implicite» di tipo puntatore a FILE

- **stdin** associato al «file» che rappresenta il dispositivo di ingresso standard (tastiera)
- **stdout** associato al «file» che rappresenta il dispositivo di uscita standard (video)
- **stderr** associato al «file» che rappresenta il dispositivo di uscita standard (video)
- **stdprn** associato al «file» che rappresenta il dispositivo di uscita su carta standard (stampante: porta di uscita lpt1: o prn)







# Apertura File

# Opening a File



- **each file needs a pointer**
  - program will reference each file by a pointer
  - declare a pointer to FILE
  - FILE \*fptr;
  - FILE \*in\_file, \*out\_file;
  - (stdin, stdout, and stderr are all file pointers!)
  
- **use fopen() with file name and mode**
  - returns pointer of type FILE
  - creates **File Control Block (FCB)**
    - an area of memory where details of data transfer are kept
    - a struct of type FILE

**FILE** \* fopen(<nome\_file>, <modo>)

**<nome\_file>**

è una stringa di caratteri (racchiusa tra " e ") che rappresenta il nome (pathname) del file specificato secondo le convenzioni del SO ("anna.txt").

**<modo>**

è una stringa di caratteri (racchiusa tra " e ") che rappresenta il modo con cui si intende aprire il file: **lettura (r)**, **scrittura inizio file (w)**, **scrittura fine file (a)**, lettura e scrittura (+), .. se il file è binario (b) o testuale.

Se il file esiste, lo apre; se il file non esiste lo crea e lo apre.

Restituisce:

- il puntatore al descrittore, se l'operazione è andata a buon fine
- il valore NULL se l'operazione non è andata a buon fine (apertura in lettura di un file che non esiste,.....)


## Modalità di apertura dei file di testo

<b>r</b>	Aprire un file di testo esistente in modalità lettura, posizionandosi all'inizio del file.
<b>w</b>	Crea un nuovo file di testo e lo apre in modalità scrittura. Posizionandosi all'inizio del file (se il file esiste, i dati precedenti vengono eliminati).
<b>a</b>	Aprire un file di testo esistente in modalità append (accoda): la scrittura può avvenire solo a partire dalla fine del file; anche se l'indicatore di posizione nel file viene modificato volontariamente. La scrittura avviene comunque alla fine del file.
<b>r+</b>	Aprire un file di testo esistente in modalità lettura e scrittura. Posizionandosi all'inizio del file.
<b>w+</b>	Crea un nuovo file di testo e lo apre in modalità lettura e scrittura (se il file esiste, i dati precedenti vengono eliminati).
<b>a+</b>	Aprire un file di testo esistente o ne crea uno nuovo in modalità append; la lettura può avvenire in una posizione qualsiasi del file, la scrittura solo alla fine.


### Proprietà dei file e flussi delle modalità operative di *fopen()*

	<b>r</b>	<b>w</b>	<b>a</b>	<b>r+</b>	<b>w+</b>	<b>a+</b>
<b>Il file deve esistere prima dell'apertura</b>	✓			✓		
<b>Il vecchio file viene azzerato</b>		✓			✓	
<b>Il flusso può essere letto</b>	✓			✓	✓	✓
<b>Il flusso può essere scritto</b>		✓	✓	✓	✓	✓
<b>Il flusso può essere scritto solo alla fine</b>			✓			✓

# Using fopen()

- 
- ❑ **takes filename, open mode as arguments**
    - `fptr = fopen("payroll.dat", "r");`
    - text modes are `r`, `w`, `a`, `r+`, `w+`, `a+`
    - binary modes are `rb`, `wb`, `ab`, `rb+`, `wb+`, `ab+`
    - **file name and mode are strings**
  - ❑ **test pointer for NULL**
    - if NULL returned, could not open file
    - if not NULL, valid pointer to file
  - ❑ **`stdin`, `stdout`, `stderr` are also pointers to FILE**
    - can use as you would any other (except for opening & closing)

# File Names

- 
- ❑ **As used with fopen()**
    - filename is a string
    - open mode is a string
  - ❑ can use **hardcoded string**, as in

```
fptr = fopen("payroll.dat", "r");
```
  - ❑ can use **variable string**, as in

```
char filename[20] = "payroll.dat";  
char open_mode[] = "r";  
fptr = fopen(filename, open_mode);
```

    - what does this allow?



## Examples



- ❑ to open a disk file called “mydata”, read-only  
`fptr = fopen(“mydata”, “r”)`
- ❑ to open/create an output file “report.txt”  
`fptr = fopen(“report.txt”, “w”)`
- ❑ to open a disk file, “file.dat”, to read **binary** data  
`fptr = fopen(“file.dat”, “rb”)`
- ❑ to open the standard output device  
you do not open stdout, just use it!



# Uso dei File

## 2. CHIUSURA DI UN FILE

`int fclose(FILE *fp)`

- è una funzione che riceve in ingresso il puntatore del file da chiudere.
- restituisce il valore 0 se l'operazione è andata a buon fine, il valore EOF se l'operazione non è andata a buon fine.

### FUNZIONAMENTO:

alla chiamata, il SO (che viene attivato in modo opportuno) chiude il file referenziato dal puntatore passato come parametro (assegna al puntatore il valore NULL) e considera disponibile l'elemento che conteneva il descrittore del file chiuso.

## 3. ALTRE FUNZIONI DI GESTIONE FILE

~~remove~~

~~rename~~

~~ridirezione (freopen)~~

## 4. GESTIONE DEGLI ERRORI

`int feof(FILE *fp)`      `macro`

restituisce 0 se non si è incontrata la fine del file, !=0 se end of file nell'ultima lettura

`int ferror(FILE *fp)`      `macro`

restituisce 0 se non si è verificato errore, !=0 se si è verificato errore

# Closing a File



- ❑ **do not need to close stdin, stdout, stderr**
  - opened automatically, closed automatically
  
- ❑ **if you open it, you close it!**
  - could cause problems by terminating a program without closing files
  
- ❑ **to close a file:**
  - use `fclose()` with each file pointer
  - `fclose(fptr);`**

## Gestione di file in C

I file hanno una struttura *sequenziale*:

- i record logici sono organizzati in una sequenza
- per accedere ad un particolare record logico, e' necessario "scorrere" tutti quelli che lo precedono.

				X				...
--	--	--	--	---	--	--	--	-----

Per accedere ad un file da un programma C, e' necessario predisporre una variabile che lo rappresenti (**puntatore a file**)

### Puntatore a file:

e' una variabile che viene utilizzata per riferire un file nelle operazioni di accesso (lettura e scrittura.).

Implicitamente essa indica:

- il file
- l'elemento corrente all'interno della sequenza

### Ad esempio:

```
FILE *fp;
```

→ il tipo FILE e' un tipo non primitivo dichiarato nel file **stdio.h**.

## Gestione di file in C

### Apertura di un file:

Prima di accedere ad un file e' necessario *aprirlo*: l'operazione di apertura compie le azioni preliminari necessarie affinché il file possa essere acceduto (in lettura o in scrittura). L'operazione di apertura inizializza il puntatore al file.

### Accesso ad un file:

Una volta aperto il file, e' possibile leggere/scrivere il file, riferendolo mediante il puntatore a file.

### Chiusura di un file:

Alla fine di una sessione di accesso (lettura o scrittura) ad un file e' necessario chiudere il file per memorizzare permanentemente il suo contenuto in memoria di massa:

## Apertura di un File

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

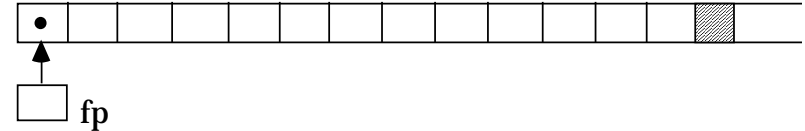
dove:

- **name** e` un array di caratteri che rappresenta il nome (assoluto o relativo) del file nel file system
- **mode** esprime la modalita` di accesso scelta.
  - "r", in lettura (read)
  - "w", in scrittura (write)
  - "a", scrittura, aggiunta in fondo (append)
  - "b", a fianco ad una delle precedenti, indica che il file e` binario
  - "t", a fianco ad una delle precedenti, indica che il file e` di testo
- Se eseguita con successo, l'operazione di apertura ritorna come risultato un *puntatore* al file aperto
- Se, invece, l'apertura fallisce (ad esempio, perche` il file non esiste), fopen restituisce il valore NULL.

## Apertura in lettura

```
fp = fopen("filename", "r")
```

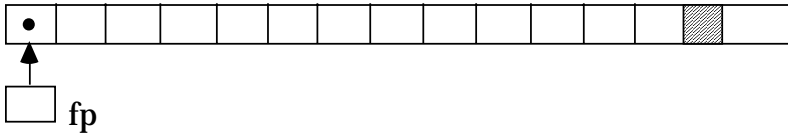
Se il file non e` vuoto:



## Apertura in scrittura

```
fp = fopen("filename", "w")
```

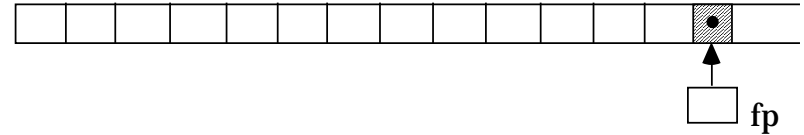
Anche se il file non e' vuoto:



- Se il file esisteva già, il suo contenuto viene **perso**.

## Apertura in aggiunta (append)

```
fp = fopen("filename", "a")
```



Il puntatore al file si posiziona sull'elemento successivo all'ultimo significativo del file ➡ se il file esisteva già, il suo contenuto non viene perso.

### Ad esempio:

```
File *fp;  
fp=fopen("c:\anna\dati", "r");  
<uso del file>
```

- fp rappresenta, dall'apertura in poi, il riferimento da utilizzare nelle operazioni di accesso a c:\anna\dati. Esso individua, in particolare:
  - il file
  - l'elemento corrente all'interno del file

## Chiusura di un File

Al termine di una sessione di accesso al file, esso deve essere **chiuso**.

L'operazione di chiusura si realizza con la funzione **fclose**:

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

dove:

- fp rappresenta il puntatore al file da chiudere.

### **fclose** ritorna come risultato un intero:

- Se l'operazione di chiusura e` eseguita correttamente restituisce il valore 0
- se la chiusura non e` andata a buon fine, ritorna la costante EOF.



## Esempio:

```
#include <stdio.h>
main()
{
    FILE *fp;
    fp = fopen("prova.dat", "w")
    <scrittura di prova.dat>
    fclose(fp);
    return 0;
}
```

## File standard di I/O

Esistono tre file testo che sono aperti automaticamente all'inizio di ogni esecuzione:

- *stdin*, standard input (tastiera), aperto in lettura
  - *stdout*, standard output (video), aperto in scrittura
  - *stderr*, standard error (video), aperto in scrittura
- ➔ *stdin*, *stdout*, *stderr* sono variabili di tipo *puntatore a file* automaticamente (ed **implicitamente**) definite ➔ **non vanno definite**.

## Funzione feof()

Durante la fase di accesso ad un file e' possibile verificare la presenza della marca di fine file con la funzione di libreria:

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

- feof(fp) controlla se e' stata raggiunta la fine del file fp nella operazione di lettura o scrittura **precedente**. Restituisce il valore 0 (falso logico) se non e' stata raggiunta la fine del file, altrimenti un valore diverso da zero (vero logico).

## Letture e Scrittura di file

Una volta aperto un file, su di esso si puo' accedere in lettura e/o scrittura, compatibilmente con quanto specificato in fase di apertura.

### Per file di testo sono disponibili funzioni di:

- Lettura/scrittura **con formato**
- Lettura/scrittura di **caratteri**
- Lettura/scrittura di **stringhe di caratteri**

### Per file binari si utilizzano funzioni di:

- Lettura/scrittura di **blocchi**

## Accesso a file di testo: Lettura/scrittura con formato

Funzioni simili a `scanf` e `printf`, ma con un parametro aggiuntivo rappresentante il puntatore al file **di testo** sul quale si vuole leggere o scrivere:

### Letture con formato:

Si usa la funzione `fscanf`:

```
int fscanf(FILE *fp, stringa-controllo, ind-elem);
```

dove:

- `fp` e' il puntatore al file
- *stringa-controllo* indica il formato dei dati da leggere
- *ind-elem* e' la lista degli indirizzi delle variabili a cui assegnare i valori letti.

### Esempio:

```
FILE *fp;  
int A; char B; float C;  
fp=fopen("dati.txt", "r");  
fscanf(fp, "%d%c%f", &A, &B, &C);  
...  
fclose(fp);
```

Restituisce il numero di elementi letti, oppure un valore negativo in caso di errore.

### Scrittura con formato:

Si usa la funzione `fprintf`:

```
int fprintf(FILE *fp, stringa-controllo, elementi);
```

dove:

- `fp` e' il puntatore al file
- *stringa-controllo* indica il formato dei dati da scrivere
- *elementi* e' la lista dei valori (espressioni) da scrivere.

### Esempio:

```
FILE *fp;  
float C=0.27;  
fp=fopen("risultati.txt", "w");  
fprintf(fp, "Risultato: %f", C*3.14);  
...  
fclose(fp);
```

Restituisce il numero di elementi scritti, oppure un valore negativo in caso di errore.

### printf/scanf e fprintf/fscanf:

- Notiamo che:

```
printf(stringa-controllo, elementi)  
scanf(stringa-controllo, ind-elementi);
```

### equivalgono a:

```
fprintf(stdout, stringa-controllo, elementi);  
fscanf(stdin, stringa-controllo, ind-elementi);
```

## Esempio:

Visualizzazione del contenuto di un file di testo:

```
#include <stdio.h>

main()
{
char buf[80]
FILE *fp;

fp=fopen("testo.txt", "r");
fscanf(fp,"%s",buf); /* 1 lettura in più */
while(!feof(fp))
{   printf("%s", buf);
    fscanf(fp,"%s",buf);
}
fclose(fp);
}
```

### oppure:

```
#include <stdio.h>

main()
{
char buf[80]
FILE *fp;

fp=fopen("testo.txt", "r");
while (fscanf(fp,"%s",buf)>0)
    printf("%s", buf);
fclose(fp);
}
```



R/W a caratteri

# LETTURA E SCRITTURA DI FILE TESTUALI

## 1. LETTURA/SCRITTURA DI CARATTERI

<code>int getc(FILE *fp)</code>	macro
<code>int getchar (void)</code>	macro (=getc(stdin))
<code>int fgetc(FILE *fp)</code>	funzione

leggono dal file specificato come parametro (o da stdin) il prossimo carattere e lo restituiscono come intero. Restituiscono EOF in caso di errore

<code>int putc(int c, FILE *fp)</code>	macro
<code>int putchar (int c)</code>	macro (=putc(c, stdout))
<code>int fputc(int c, FILE *fp)</code>	funzione

scrivono sul file specificato come parametro (o su stdout) il carattere specificato come parametro e lo restituiscono come intero. Restituiscono EOF in caso di errore



## 2. LETTURA/SCRITTURA FORMATTATA

```
int fscanf(FILE*fp,»stringa_di_controllo»,elenco_ind_elem)
int scanf(»stringa_di_controllo»,elenco_ind_elem)
```

leggono dal file specificato come parametro (o da stdin) gli elementi specificati. Restituiscono il numero di elementi effettivamente letti o un numero negativo in caso di errore

```
int fprintf(FILE*fp,»stringa_di_controllo»,elenco_elem)
int printf(»stringa_di_controllo»,elenco_elem)
```

scrivono sul file specificato come parametro, che può essere **stdprn** (o su stdout) gli elementi specificati. Restituiscono il numero di elementi effettivamente scritti o un numero negativo in caso di errore.

---

# Sequential File Access



## Character I/O

`fgetc`      `c = fgetc(fp_ptr);`

- returns next char read if successful
- returns EOF on error
- (if `fp_ptr` is for a disk file, gets next char in file)
- (if `fp_ptr` is `stdin`, gets next char from (typically) keyboard)

`fputc`      `fputc(c, fp_ptr);`

- returns char written if successful
- returns EOF on error
- (if `fp_ptr` is for a disk file, writes char to file)
- (if `fp_ptr` is `stdout`, writes char to (typically) screen)

# Reading and Writing from Standard Input

---

## Recall C library procedures for reading/writing from standard input:

- `printf("control string",...)` writes to standard output
- `scanf("control string",...)` reads from standard input
- `int getchar(void)` read one character from standard input
- `putchar(int)` write one character to standard output

### Example:

```
/* this proc writes what it reads */
void echo(void)
{
    int ch;          /* must be an integer since EOF is -1 */

    while ((ch = getchar()) != EOF)
    {
        putchar(ch);
    }
}
```



# Reading from and writing to a file

---

There are versions of these routines that read from and write to files.

- `fprintf(fd, "control string"...) writes to *fd`
- `fscanf(fd, "control string"...) reads from *fd`
- `int fgetc(fd) read character from *fd`
- `int fputc(int ch, FILE *fd) write character ch to *fd`

```
void echo(void) /* this proc writes what it reads */
{
    int ch; /* must be an integer since EOF is -1 */
    FILE *fdin, *fdout;

    fdin = fopen("inputfile", "r");
    fdout = fopen("outputfile", "w");

    while ((ch = fgetc(fdin)) != EOF)
    {
        fputc(ch, fdout);
    }
    fclose(fdin); fclose(fdout);
}
```

## LETTURA/SCRITTURA CARATTERI

```
#include <stdio.h>

void main(void)
{   FILE *f1, *f2;
    int carattere;

    if ((f1=fopen("prova1.txt","w"))!=NULL)
    {
        printf("Inserire i caratteri da memorizzare in prova1.txt\n");

        while ((carattere=getchar())!='\n')
        {
            putchar(carattere,f1);
        }
        fclose(f1); /*f1 viene chiuso in scrittura*/

        if((f1=fopen("prova1.txt","r"))!=NULL)
        {
            if((f2=fopen("prova2.txt","w"))!=NULL)
            {
                printf("\nprova1.txt viene visualizzato sullo schermo e
copiato          in prova2\n");
                /*ciclo che copia f1 in f2 carattere per carattere*/
                while((carattere=getc(f1))!=EOF)
                {
                    putchar(carattere);/*visualizza ogni carattere */
                    putc(carattere,f2);
                } /* end while */

                fclose(f1);
                fclose(f2);
            } /* fine ramo apertura corretta di f1 e f2 */
            else /* apertura non corretta di f2 */
            {fclose(f1);
                printf("il file prova2.txt non puo' essere aperto in
                    scrittura\n");
            }
        } /* fine ramo apertura corretta di f1 in lettura */
        else /* f1 non puo' essere aperto in lettura */
        {
            printf("il file prova1.txt non puo'essere aperto in
lettura\n");
        }
    } /* fine ramo apertura corretta di f1 in scrittura */
    else
    {
        printf("il file prova1.txt non puo' essere aperto in
scrittura");
    }
}/*fine main */
```

## Esempio:

Programma che concatena i file dati come argomento in un unico file (*stdout*):

```
#include <stdio.h>

main(int argc, char **argv)
{
    FILE *fp;
    void filecopy(FILE *, FILE *);

    if (argc==1) filecopy(stdin, stdout);
    else
        while (--argc>0)
            if ((fp=fopen(++argv, "r"))==NULL)
                {
                    printf("\nImpossibile aprire il
                        file %s\n", *argv);
                    exit(1);
                }
            else
                {filecopy(fp, stdout);
                 fclose(fp);
                }
        return 0;
}

void filecopy(FILE *inputFile,
              FILE *outputFile)
{int c;

    while((c=getc(inputFile))!=EOF)
        putc(c, outputFile);
}
```

## Note sull'esempio:

- se non ci sono argomenti (*argc*=1), il programma copia lo standard input nello standard output;
- la funzione **filecopy** effettua la copia del file carattere per carattere;
- se uno dei file indicati come argomento non esiste, la funzione **fopen** fallisce, restituendo il valore NULL. In questo caso il programma termina (**exit**) restituendo il valore 1 e stampando un messaggio di errore;
- sarebbe meglio scrivere i messaggi di errore sullo standard error (*ridirezione*).

```
printf(stderr, "\nImpossibile aprire
            il file %s\n", *argv);
```

Per non perdere dati, la funzione **exit** chiude automaticamente ogni file aperto.

- il ciclo:

```
while((c=getc(inputFile))!=EOF)
    putc(c, outputFile);
```

poteva essere scritto anche come:

```
c=getc(inputFile);
while(!feof(inputFile))
{ putc(c, outputFile);
  c=getc(inputFile);
}
```



# R/W Formattata

## 2. LETTURA/SCRITTURA FORMATTATA

```
int fscanf(FILE*fp,»stringa_di_controllo»,elenco_ind_elem)  
int scanf(»stringa_di_controllo»,elenco_ind_elem)
```

leggono dal file specificato come parametro (o da stdin) gli elementi specificati. Restituiscono il numero di elementi effettivamente letti o un numero negativo in caso di errore

```
int fprintf(FILE*fp,»stringa_di_controllo»,elenco_elem)  
int printf(»stringa_di_controllo»,elenco_elem)
```

scrivono sul file specificato come parametro, che può essere **stdprn** (o su stdout) gli elementi specificati. Restituiscono il numero di elementi effettivamente scritti o un numero negativo in caso di errore.

## LETTURA/SCRITTURA FORMATTATA (ARRAY DI STRUTTURE)

```
#include <stdio.h>
#define Lmax 10
#define N 5
typedef struct {
    char nome[Lmax];
    int somma;
    float media;
}ELE;
FILE *f1;

void main(void)
{
    ELE elenco[N];
    int i;
    void Scrivi_su_file(const char *nome_file, ELE dati[]);

    printf("inserire gli elementi dell'elenco\n");
    for (i=0;i<N;i++)
    {
        printf("valori relativi all'elemento %d\n",i);
        scanf("%s%d%f",elenco[i].nome, &elenco[i].somma,
&elenco[i].media);
    }

    printf("\nvisualizzazione degli elementi nell'elenco\n");
    for (i=0;i<N;i++)
    {
        printf("valori relativi all'elemento %d\n",i);
        printf("%s %d %.2f\n",elenco[i].nome,elenco[i].somma,
                elenco[i].media);
    }
    Scrivi_su_file("anna.txt",elenco);
}/* fine main */

void Scrivi_su_file(const char *nome_file, ELE dati[])
{ int i;

    if ((f1=fopen(nome_file,"w"))!=NULL)
    {
        for (i=0;i<N;i++)
        { printf("valori relativi all'elemento %d\n",i);
          fprintf(f1,"%s %d %.3f\n",dati[i].nome,dati[i].somma,
                dati[i].media);
        }
    }
    else
        printf("il file non puo' essere aperto\n");
} /* fine procedura */
```



# File Text vs File Binari

# File Types



## □ Text files

- for portability; **all systems support these**
- a text stream is a sequence of characters divided into lines
- each line is 0 or more characters followed by a newline character
- C not concerned with how it is physically stored
- (DOS inserts <cr><lf> in place of newline)
- File ends with EOF condition set (defined in stdio.h)

## □ Binary files

- **not supported by all systems**; might get treated as a text file
- use when speed or storage conditions require
- does not insert or change characters



# Writing a Text File



- **Typically for a report to disk**
  - just like a printed report, except data goes to disk, not printer
  - several choices for output; simplest to use is `fprintf()`
  - file is opened in text mode
- **Different data format for characters & “numbers”**
  - text output is made up of ASCII characters
  - numbers (used for math calculations) are binary values that must be converted to text before display/output
  - `printf()/fprintf()` does the conversion for you!

# File Open Mode vs Data Format



## Text Mode

- converts ‘\n’ to CR/LF for DOS
- adds EOF (1Ah)

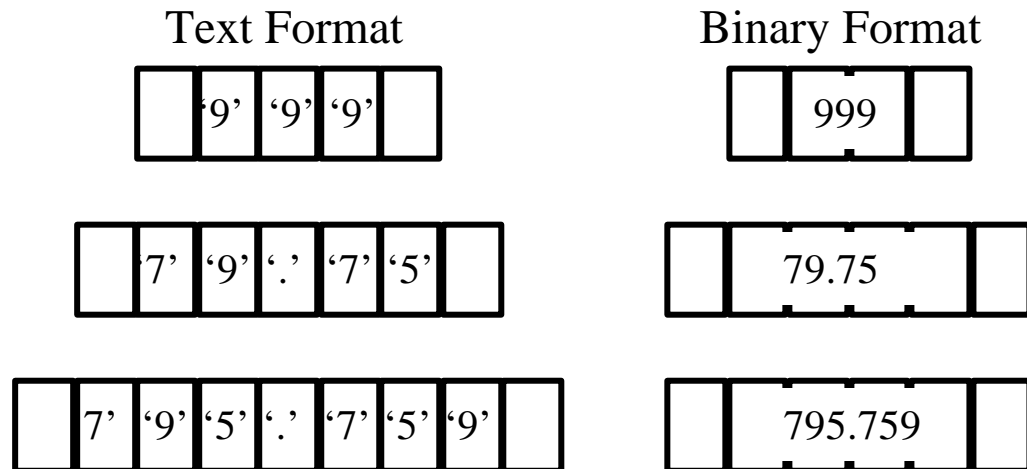
## Binary Mode

- **no conversion**

## Data: Text vs Binary Format

- how numbers are stored
- If need binary format data, use binary mode

# Text vs Binary Format

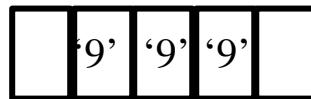


- text format usually requires more bytes for disk storage
- in text format, numbers are stored as ASCII characters
- using binary number storage for text files causes problems!
- need to use binary format in memory for math operations
- requires conversion!

# A Little Bit of Storage



Text Format

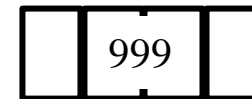


Bit Pattern:

00111001 00111001 00111001

Decimal Value:  
3,750,201

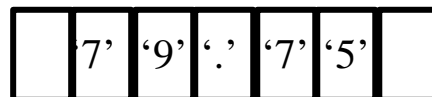
Binary Format



Bit Pattern:

00000011 11100111

Decimal Value:  
999



Bit Pattern:

00110111 00111001 00101110 00110111 00110101

Decimal Value:  
959,330,101



Decimal Value:  
79.75

Bit Pattern:

00000000 00011111 00100111 00000001

mantissa

exponent

# Easy Text Output



## □ Use printf() / fprintf()

to output characters, use %c and %s - not an issue

to output numbers, use %d, %f, etc. - still not an issue!

the binary-stored number 999

(999 in decimal, 00000011 11100111 in binary)

becomes the ASCII character string 999

(3,750,201 in decimal, 00111001 00111001 00111001 in binary)

**because printf() / fprintf() does the conversion!**



# Esercizi

## LETTURA/SCRITTURA FORMATTATA (ARRAY DI STRUTTURE)

```
#include <stdio.h>
#define Lmax 10
#define N 5
typedef struct {
    char nome[Lmax];
    int somma;
    float media;
}ELE;
FILE *f1;

void main(void)
{
    ELE elenco[N];
    int i;
    void Scrivi_su_file(const char *nome_file, ELE dati[]);

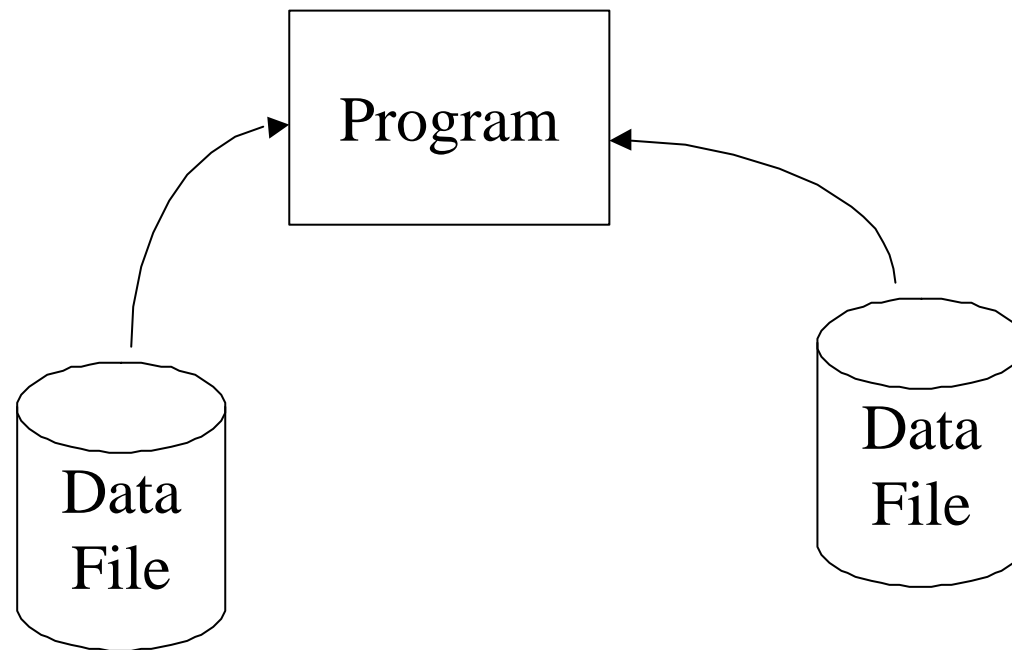
    printf("inserire gli elementi dell'elenco\n");
    for (i=0;i<N;i++)
    {
        printf("valori relativi all'elemento %d\n",i);
        scanf("%s%d%f",elenco[i].nome, &elenco[i].somma,
&elenco[i].media);
    }

    printf("\nvisualizzazione degli elementi nell'elenco\n");
    for (i=0;i<N;i++)
    {
        printf("valori relativi all'elemento %d\n",i);
        printf("%s %d %.2f\n",elenco[i].nome,elenco[i].somma,
                elenco[i].media);
    }
    Scrivi_su_file("anna.txt",elenco);
}/* fine main */

void Scrivi_su_file(const char *nome_file, ELE dati[])
{ int i;

    if ((f1=fopen(nome_file,"w"))!=NULL)
    {
        for (i=0;i<N;i++)
        { printf("valori relativi all'elemento %d\n",i);
          fprintf(f1,"%s %d %.3f\n",dati[i].nome,dati[i].somma,
                dati[i].media);
        }
    }
    else
        printf("il file non puo' essere aperto\n");
} /* fine procedura */
```

# Writing Sequential Output Files





# Writing Sequential Output Files



Create 2 output files containing data that is in order by key.  
file01.dat file02.dat

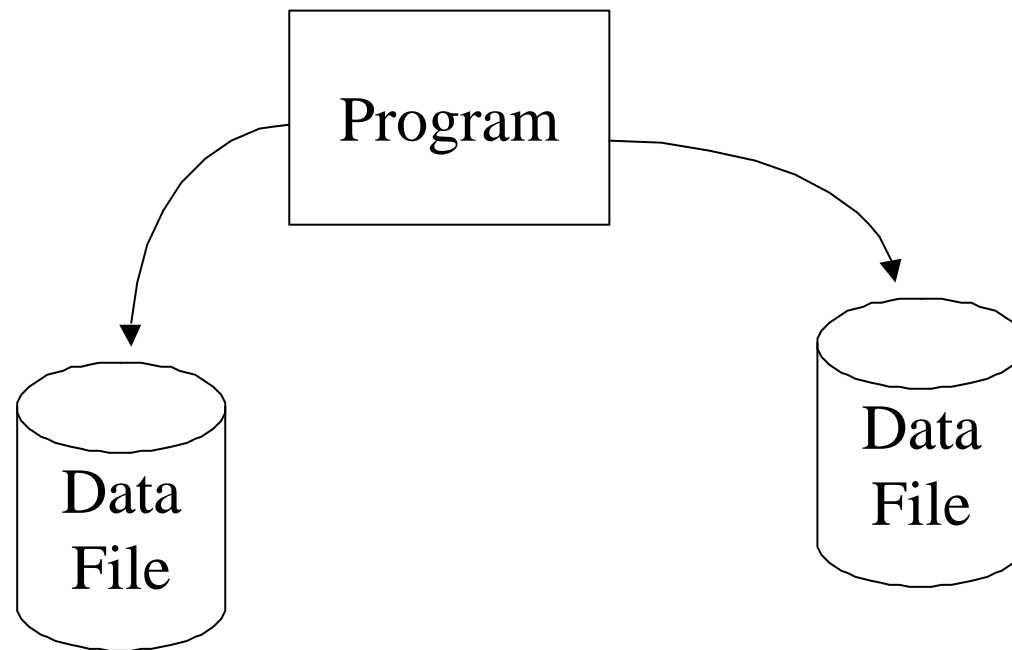
Open the files write mode and write the data sequentially from an initialized array of structs. The file contains binary data.

Each “record” of data is represented by the following struct:

```
struct my-rec
{
    int         id;        /* ordered key */
    char        last_name[20];
    char        first_name[20];
    char        acct_type;
    double      acct_bal;
};
```

**Write the program!**

# Reading Files Sequentially



# Reading Files Sequentially



Assume input files containing data that is in order by key.  
file01.dat file02.dat

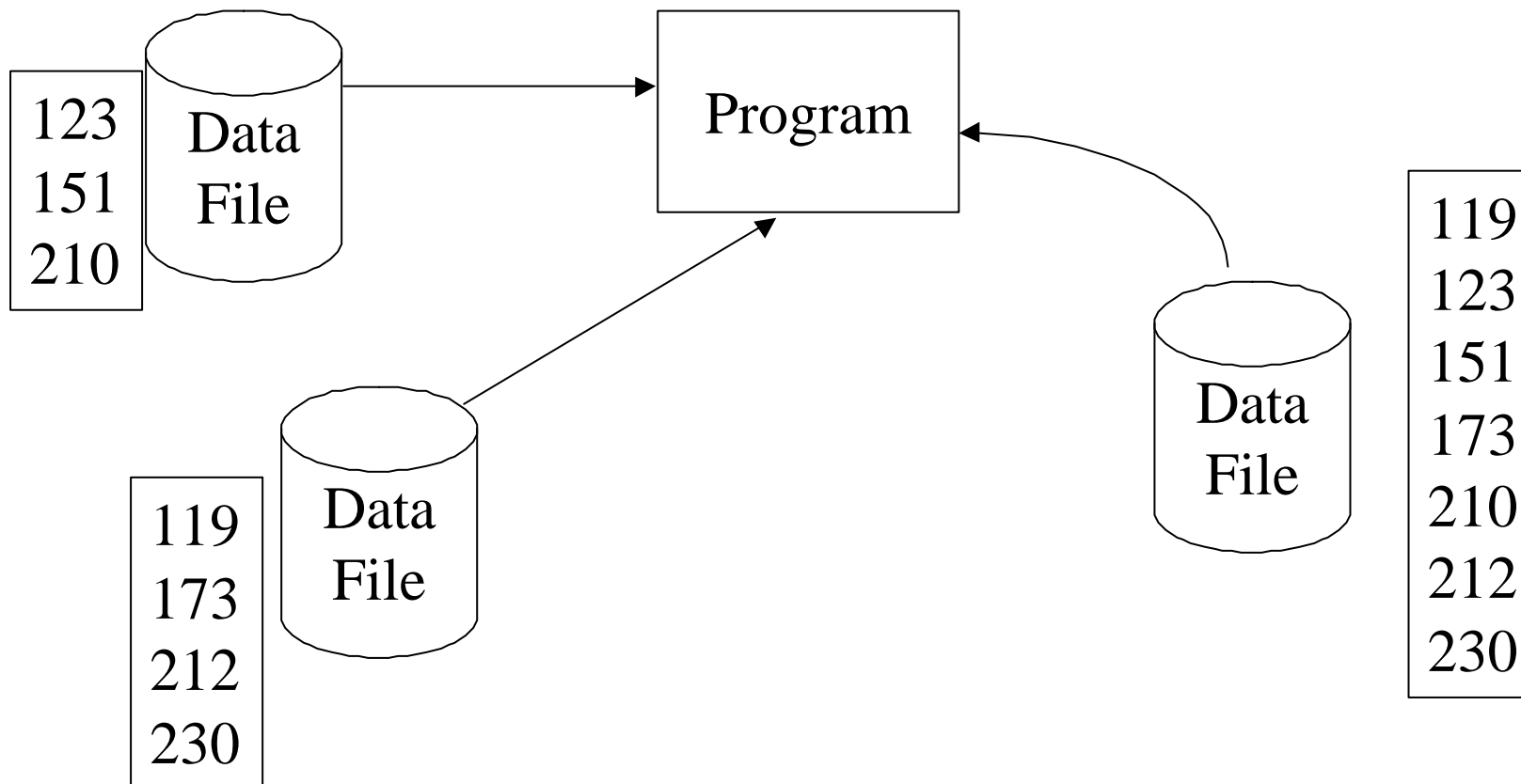
Open the files read-only and read in the data sequentially into an instance of a struct. Display the data. The file contain binary data.

Each “record” of data is represented by the following struct:

```
struct my-rec
{
    int    id;        /* ordered key */
    char   last_name[20];
    char   first_name[20];
    char   acct_type;
    double acct_bal;
};
```

**Write the program!**

# Merging Ordered Data Files



# Merging Ordered Data Files



Assume two input files containing data that is in order by key.  
file01.dat                      file02.dat

Open both files read-only. Read in data and write out to a single, ordered results file (result.dat). The files contain binary data.

Each “record” of data is represented by the following struct:

```
struct my-rec
{
    int    id;      /* ordered key */
    char   last_name[20];
    char   first_name[20];
    char   acct_type;
    double acct_bal;
};
```

**Write the program!**