Gestione dei File

Perché i file?

- Sono strutture dati **persistenti**
- Sono solitamente memorizzati sui dischi
  - Si usano dall'interno dei programmi
- Realizzano la *persistenza dei dati*
  - cioè del contenuto delle variabili
- Tramite i file, i dati possono sopravvivere al termine dell'esecuzione del programma
- N.B.: i file sono usati anche per memorizzare i programmi !!
  - Quando se ne chiede l'esecuzione, il sistema operativo copia il programma (eseguibile, conservato in un file) in memoria centrale e inizia a eseguirlo

File binari, file di testo

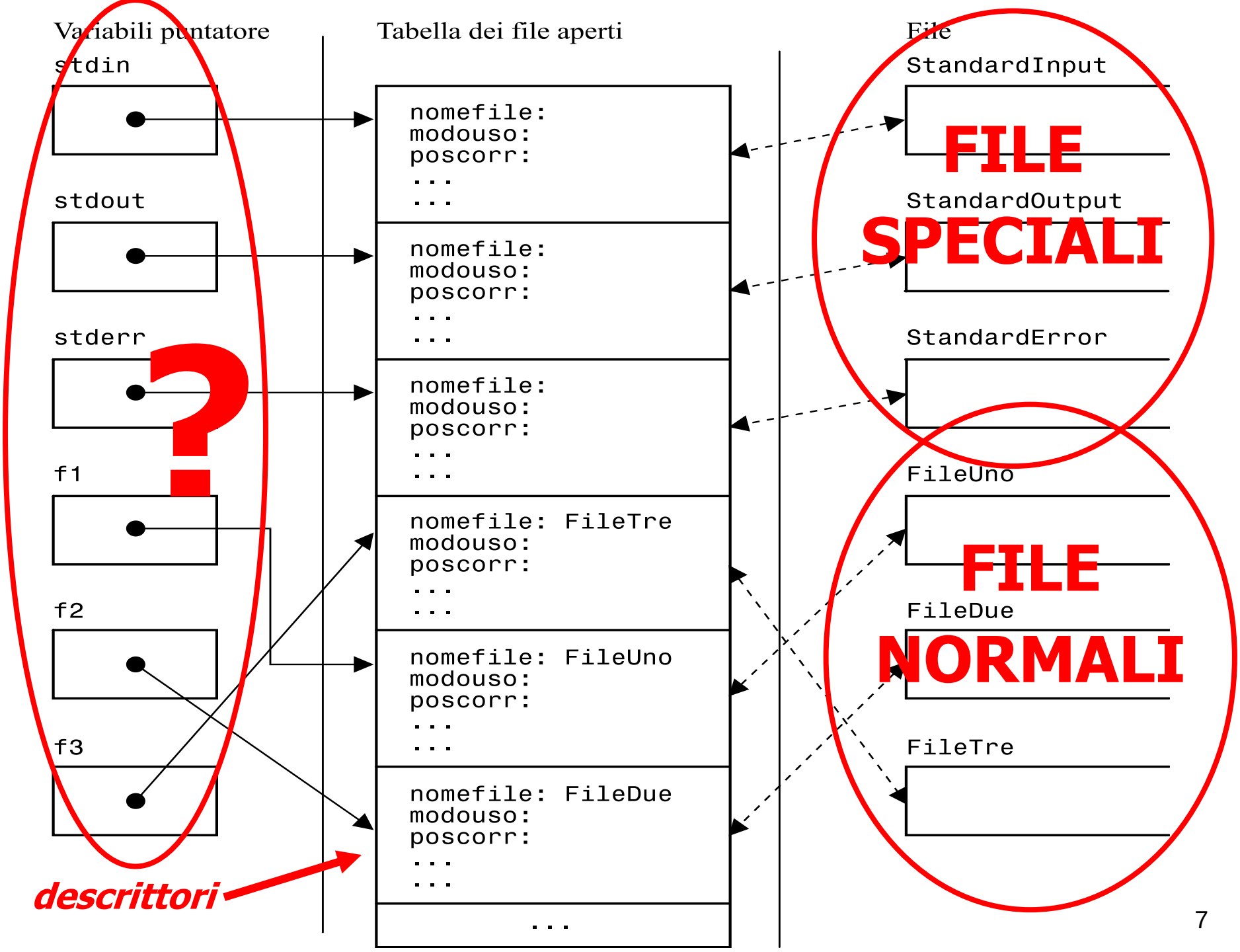
- I file sono strutture dati **sequenziali**
  - Sequenziale significa: si leggono (e scrivono) gli elementi del file in sequenza
- Un file ***binario*** è una sequenza di byte che non è "interpretata" in alcun modo
- Un file ***di testo*** è una sequenza di caratteri "interpretata":
  - Alcuni caratteri rappresentano separatori
    - Esempio: il carattere di "newline" è interpretato dalla stampante come "salto alla riga successiva"

File e sistema operativo

- I file sono gestiti dal S.O.
  - Sono resi visibili all'interno del linguaggio per essere manipolati attraverso opportune funzioni di libreria
- Per essere usato, un file deve essere prima ***aperto***, e dopo l'uso andrà ***chiuso***
  - *Aprire e chiudere il **"flusso di comunicazione"** tra il programma e il file*
- In C anche le periferiche sono viste come file (chiamati "file speciali")
  - **stdin** e **stdout** (terminali, stampanti, ecc)
  - **Si può "leggere" e "scrivere" con le stesse modalità (quelle dei file) da ogni device di I/O**

Rappresentazione interna dei file

- Ogni file aperto da un prog. ha un **descrittore**
  - Risiede nella *tabella dei file aperti*, una delle strutture dati che il S.O. associa ai programmi in esecuzione
- Il descrittore memorizza:
  - la **modalità** d'uso (read, write)
  - la **posizione** corrente all'interno del file
  - l'indicatore di eventuale **errore**
  - l'indicatore di **eof** (end-of-file)
- L'**apertura** del file restituisce un descrittore
  - Per la precisione, un **puntatore** a un descrittore


Variabili puntatore
Tabella dei file aperti
File
stdin
stdout
stderr
f1
f2
f3
nomefile:
modouso:
poscorr:
...
...
nomefile: FileTre
nomefile: FileUno
nomefile: FileDue
StandardInput
StandardOutput
StandardError
FileUno
FileDue
FileTre
FILE SPECIALI
FILE NORMALI
descrittori

Dichiarare e aprire un file

- Puntatore al descrittore**: FILE * fp**
- Apertura del file:
  - FILE * **fopen** (char * *nomefile*, char * *modalità*)

    nomefile e *modalità* sono stringhe

    nomefile dà il percorso (path), oppure il nome è interpretato nella cartella in cui si lancia l'eseguibile

    apre il file (oppure lo crea, se è inesistente)
  - modalità di apertura

    "r" lettura modalità testo, posizionamento inizio file (**read**)

    "w" scrittura modalità testo, posizionamento inizio file (**write**)

    "a" scrittura in modalità testo, posizionamento fine file (**append**)

    "rb", **"wb"** e **"ab"** (idem, ma considerando il file come **binario**)
- Se si verifica un errore, fopen() restituisce **NULL**

Cancellare, ridenominare, chiudere

int remove (char * nomefile)

- cancella file nomefile
- restituisce 0 se buon fine, != 0 altrimenti

int rename (char *oldname, char *newname)

- cambia nome al file
- restituisce 0 se buon fine, !=0 altrimenti

int fclose (FILE * fp)

- fp diventa NULL, descrittore di tipo FILE rilasciato
- restituisce 0 se buon fine, altrimenti EOF

Gestione degli errori

int ferror (FILE * fp)

- restituisce 0 (falso) se NON è stato commesso errore

int **feof** (FILE * fp)

- restituisce 0 (falso) se NON si è alla fine

void clearerr (FILE * fp)

- riporta al valore normale gli indicatori di errore e eof

Lettura e scrittura

- Si opera sui file in quattro modi possibili
- Tre modi per i file di testo:
  - Precisando la **formattazione** dell' I/O
  - Un **carattere** alla volta
  - Per **linee** di testo
    - Fino ad ogni prossimo '\n'
- Un modo per i file binari:
  - Per **blocchi** di byte
    - approccio "à-la-sizeof"

Lettura / scrittura formattata

- scanf e printf fanno riferimento a stdin e stdout
  - Non serve specificare su quale file agiscono!!
- **f**printf e **f**scanf fanno riferimento a file generici e si usano esattamente come scanf e printf

int fprintf (FILE * **fp**, *str_di_controllo*, *elementi*)

int fscanf (FILE * **fp**, *str_di_controllo*, *indirizzo_elementi*)

- Restituiscono il numero di elementi effettivamente letti/scritti, o zero se errore

Leggere, mostrare a video e salvare il contenuto di una struct

```
#include <stdio.h>
int main ()  {
    FILE * fp1, * fp2;
    struct { int numero; char c; } dato;
    fp1 = fopen (“nomeFile1“,"r");/*file lettura, modalità testo */
    fp2 = fopen (“nomeFile2“,“w");/*file scrittura,modalità testo*/
    if (fp1 != NULL && fp2 != NULL ) {
        fscanf(fp1,“%d%c”,&dato.numero,&dato.c);
        printf(“%d%c”,dato.numero,dato.c);
        fprintf(fp2,“%d%c”,dato.numero,dato.c);
        fclose (fp1);
        fclose (fp2);
    } else
        printf ("Il file non può essere aperto.\n");
    return 0;
}
```

Lettura carattere per carattere

- *int getchar (void)*
  - *legge un carattere **da standard input**, restituendolo come intero*
- *int putchar (int c)*
  - *scrive un carattere **su standard output***
- int **f**getc (FILE * fp)
- int **f**putc (int c, FILE * fp)
  - leggono/scrivono un carattere dal/sul **f**ile descritto da *fp, restituendolo come inter

Se fp è stdin/stdout è identico scrivere getchar() e putchar(c)

Leggere e mostrare a video un file

```
#include <stdio.h>

int main ()  {
    FILE * fp;
    char c;
    fp = fopen ("filechar", "r");  /* file lettura, modalità testo */
    if (fp != NULL) {
        c = fgetc (fp);
        while (c != EOF)  {      /* oppure while (! feof (fp)) */
                putchar (c);
                c = fgetc (fp);
        }
        fclose (fp);
    } else
        printf ("Il file non può essere aperto.\n");
    return 0;
}
```

```
while ((c=fgetc(fp)) != EOF)
    putchar(c);
```

Lettura / scrittura per linee di testo

- Su **stdin** e **stdout**:
  - char * gets (char * s)
    - s è l'array in cui copiare la stringa letta da stdin
      - s risulta terminata da un '\0', aggiunto in automatico
    - Non si può limitare la dimensione dei dati in input
      - Non controlla che la stringa s sia sufficientemente grande
    - In caso di errore, restituisce **NULL**
  - int puts (char * s)
    - scrive la stringa s, escluso il '\0'
      - al posto del '\0' che si trova nella stringa scrive un '\n'
    - Restituisce n>=0 se OK, EOF in caso di errore

Lettura / scrittura per linee di testo

- Su **file qualunque** (fp):
  - char * **f**gets (char * s, int n, FILE * fp)
    - legge al più n-1 caratteri, fino a '\n' o EOF
    - se incontra '\n' lo inserisce tra gli n-1, e mette alla fine **anche** il terminatore '\0'
    - In caso di errore, restituisce **NULL**
  - int **f**puts (char * s, FILE * fp)
    - come puts
      - Ma **non** aggiunge il '\n', si limita a non scrivere il '\0'
    - Restituisce 0 se OK, EOF in caso di errore

```
int copiaselettiva (char refstr []) {
    char line [MAXLINE];
    FILE * fin, * fout;
    fin = fopen ("filein", "r");
    if (fin == NULL)
            return ERROR;
    fout = fopen ("fileout", "w");      /* aperto in scrittura, modalità testo */
    if (fout == NULL) {
            fclose (fin);
            return ERROR;
    }
    while (fgets (line, MAXLINE, fin) != NULL)
            /* fgets legge da filein al più MAXLINE–1 caratteri */
            if ( strstr(line, refstr) != NULL)
                    fputs (line, fout);
    /* strstr rest. posiz. della prima occorrenza di refstr in line; se non c'è, NULL */
    fclose (fin);
    fclose (fout);
    return OK;
}
```

```
#define OK 1
#define ERROR 0
# define MAXLINE 100
```

Lettura / scrittura per blocchi di byte

- Ci sono funzioni che consentono di scrivere o leggere un intero blocco di dati testuali o binari
  - Utili, per esempio, quando si vuole scrivere su file un'intera struct

- int fread (void *punt, dim_blocco, num_blocchi, FILE *fp)
  - Legge dal file fp un numero di byte pari a **dim_blocco*num_blocchi** e li memorizza nell'area di memoria puntata da **punt**
  - Restituisce il numero di blocchi letti
- Es:
  - fread(a, 1, 100, fp) /* restituisce un valore tra 0 e 100 */
  - fread(a, 100, 1, fp) /* restituisce 1 o 0 */

- int fwrite (void *punt, dim_blocco, num_blocchi, FILE *fp)
  - Scrive sul file fp un numero di byte pari a dim_blocco*num_blocchi letti dall' area di memoria puntata da punt
  - Restituisce il numero di blocchi scritti

Accesso diretto

- Si può accedere ad uno specifico byte come se il file fosse un array di blocchi di byte:
  - int fseek (FILE * fp, long offset, int refpoint)
    - imposta la posizione corrente a un valore pari a uno **spostamento** (positivo o negativo) pari a offset, calcolato **rispetto** a uno dei seguenti punti di partenza:
      - L'inizio del file, se refpoint vale SEEK_SET (costante di stdio.h)
      - L'attuale posizione corrente, se refpoint vale SEEK_CUR (altra costante di stdio.h)
      - La fine del file, se refpoint vale SEEK_END (costante di stdio.h)
    - restituisce 0 se l'operazione di spostamento va a buon fine, un valore diverso altrimenti
- Esempi:
  - fseek (fp,0L,SEEK_SET) /* inizio file */
  - fseek (fp,0L,SEEK_END) /* fine file */
  - fseek (fp,-10L,SEEK_CUR) /* ?? */

Accesso diretto

- long int ftell (FILE * fp)
  - restituisce posizione corrente:
    - per file binari è il numero di byte dall'inizio
    - per file testuali il numero dipende dall'implementazione
- void rewind (FILE * fp)
  - definita dall'equivalenza:
  - void rewind (f) ≡ fseek (f, 0, SEEK_SET);
    - "riavvolge" il file (la pos. corrente torna all'inizio)
    - azzera l'indicatore di errore

Esercizietto

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 100
int main() {
        char *temp, line[MAXLINE], match[MAXLINE]; FILE * cfPtr; int countMatch = 0;
        if((cfPtr = fopen("prova1.txt", "r")) != NULL) {
                printf("stringa da cercare--> ");
                if (gets(match) != NULL) {
                        printf("match = %s\n",match);
                        while (!feof(cfPtr))
                                if (fgets(line, MAXLINE, cfPtr) != NULL) {
                                        temp = strstr(line, match);
                                        if (temp !=NULL)
                                                countMatch++;
                                }
                        printf("numero match--> %d", countMatch);
                }
        }
        else printf("errore apertura file");
        return 0;
}
```

Che cosa fa?

Se non si capisce…

provare per credere!

char * strstr(char* s, char* p)
restituisce NULL, oppure un puntatore al carattere della stringa s a partire dal quale inizia la sottostringa p (se essa è presente)

Nome dei file da riga di comando

- Ci sono diversi modi per fornire a un programma il nome dei file su cui deve lavorare:
  - Usare il nome del file direttamente nel programma - **non lascia molta flessibilità!**
  - Lasciare che gli utenti specifichino in input i nomi
  - Specificare il nome dei file da linea di comando quando si lancia il programma

Nome dei file da riga di comando

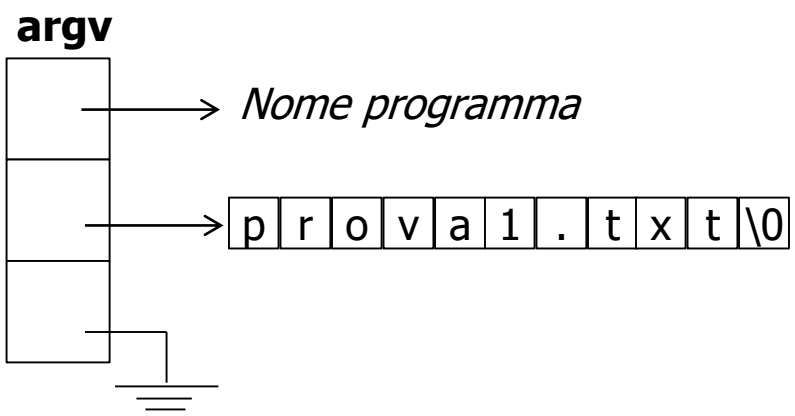
- Argomenti da linea di commando: si definisce il `main` come una funzione con due parametri:

```
int main(int argc, char *argv[])
{
   …
}
```

- **`argc`** è il numero di parametri sulla linea di comando
- **`argv`** è un array di puntatori alle stringhe dei parametri

Nome dei file da riga di comando

```
#include <string.h>
#define MAXLINE 100
int main(int argc, char * argv[]) {
        char *temp, line[MAXLINE], match[MAXLINE];
        FILE * cfPtr; int countMatch = 0;
        if (argc!=2) {
                printf("Manca il nome del file");
                return 1;
        }
        if((cfPtr = fopen(argv[1], "r")) != NULL) {
        printf("stringa da cercare--> ");
        … …
        … …
        }
        else printf("errore apertura file");
        return 0;
}
```



Nome dei file da riga di comando

Un altro esempio: il programma `demo`

```
demo names.dat dates.dat
```

- `argv[0]` punta al nome del programma
- `argv[1]` fino a `argv[argc-1]` puntano ai restanti parametri
- `argv[argc]` è un puntatore nullo
- `argc` è 3
- `argv` è strutturato come segue: