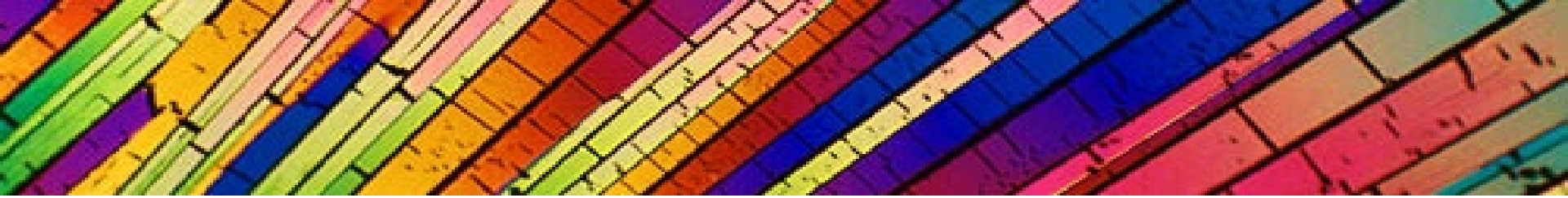
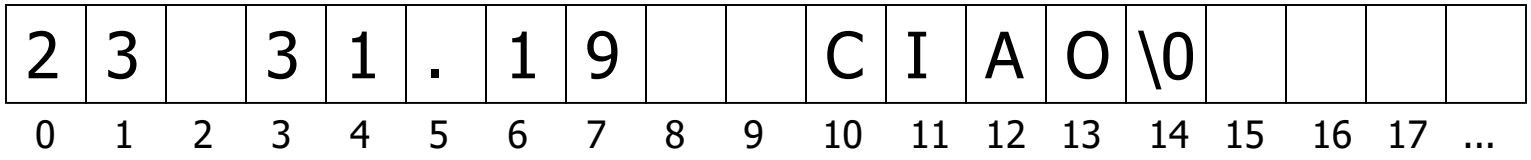


File sequenziali

Ver. 3

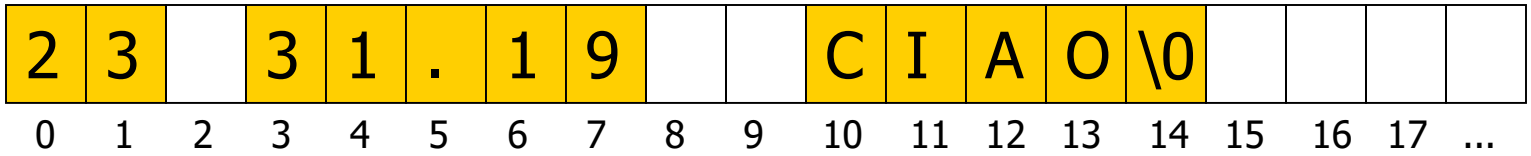
I file

- *File:* sequenza di byte a cui il Sistema Operativo dà un nome ed una collocazione sul dispositivo di memoria di massa (es. disco)
- Può essere paragonato ad una lunga stringa
- Uno *stream* è un generico flusso di caratteri
- Un file può essere una *sorgente di stream* (anche la tastiera è una sorgente di stream)
- Un file può essere una *destinazione di stream* (anche il video è una destinazione di stream)



I file

- I byte di un file sono idealmente raggruppati in blocchi detti *record* o *campi (field)*
- Un *record* è un blocco di byte che costituisce un elemento (numero, stringa, struttura, etc.)



- Ogni lettura/scrittura legge/scrive un *record*
- Tutte le funzioni e le costanti che riguardano i file sono dichiarate/definite in `<stdio.h>`

Classificazione per accesso

- Per accedere ad uno specifico record è necessario posizionarsi al byte da dove inizia
- File ad accesso sequenziale
  - hanno record di lunghezza variabile
  - per accedere al record numero *n* bisogna leggere tutti i precedenti *n* -1 record per determinare il byte da dove esso inizia
- File ad accesso diretto (casuale, random)
  - hanno record di lunghezza fissa
  - la lunghezza *L* del record viene decisa dal programmatore file per file
  - per accedere al record numero *n* si determina il byte da dove esso inizia con il calcolo: *n* **L*

Classificazione per contenuto

- File di testo
  - Sono sequenze di caratteri organizzati in righe
  - Ogni riga è terminata da un ritorno a capo, l'ultima può non averlo
  - Le funzioni di I/O per stringhe e caratteri gestiscono il carattere `'\n'` come generico ritorno a capo
- File binari
  - Sono sequenze "grezze" di byte: nessun carattere né sequenza di caratteri viene interpretata in modo speciale (fine riga, `'\0'`, ecc.)
- In questo capitolo si considerano solo i file di testo ad accesso sequenziale

File e stream

- Dal punto di vista del linguaggio C, non c'è differenza tra leggere da un file o dalla tastiera, o tra scrivere su un file o sul video: sono in ogni caso degli *stream* di caratteri
- Le funzioni di I/O per i file richiedono che sia indicato lo stream da utilizzare, mentre quelle per video e tastiera utilizzano implicitamente degli stream associati a questi dispositivi

Ritorno a capo nei file di testo

- A seconda del sistema operativo utilizzato, il ritorno a capo è in realtà costituito da una sequenza di uno o più caratteri:
  - MS-DOS/Win: CR+LF (codici ASCII 13 e 10: `\r\n`)
  - Unix/Linux/OSX: LF (codice 10: `\n`)
- Le funzioni di I/O viste considerano in carattere '`\n`' come generico "ritorno a capo":
  - **in input** la sequenza propria del sistema operativo per il ritorno a capo viene trasformata in '`\n`' (cioè legge e converte automaticamente 1 o 2 caratteri)
  - **in output** il carattere '`\n`' viene sostituito con la sequenza propria del sistema operativo (1 o 2 car.)

La costante EOF

- EOF (acronimo di End Of File) è una costante simbolica di tipo `int` definita da una `#define` in `<stdio.h>` e di solito pari al valore –1
- Il valore EOF viene restituito da alcune funzioni di I/O per segnalare il raggiungimento della fine del file (alcune la usano anche come generica segnalazione di errore)
- I file di testo non hanno al fondo un *carattere di fine file* (almeno in Windows, Linux e OSX), le funzioni di input riconoscono di essere alla fine del file grazie al Sistema Operativo che, conoscendone la lunghezza, segnala ad esse quando non ci sono più caratteri da leggere

Il tipo FILE

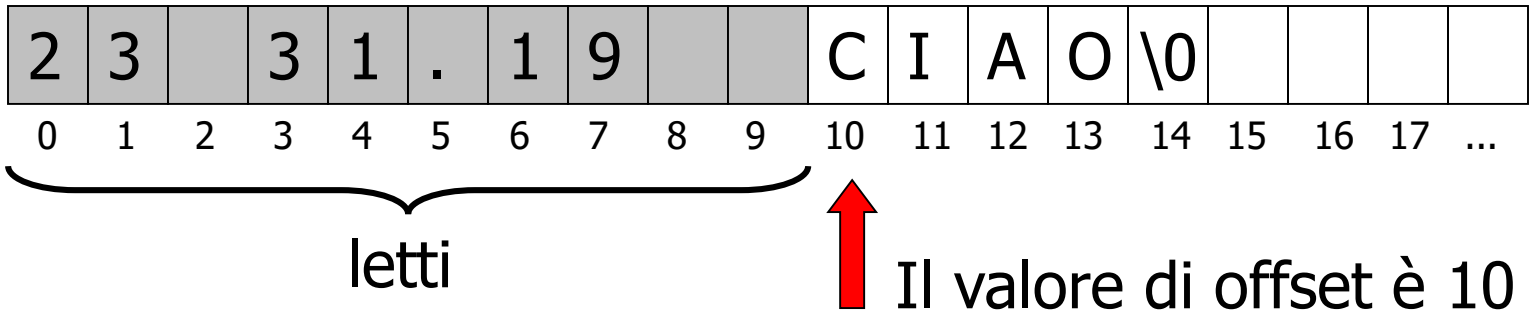
- Per utilizzare un file è necessario che il sistema legga dal disco e tenga in memoria le sue caratteristiche (lunghezza, ecc.)
- Queste informazioni sono memorizzate in una struttura dati composta (`struct`) che ha tipo `FILE` (in maiuscolo)
- Ogni accesso a un file avviene attraverso una variabile che si riferisce alla struttura `FILE` associata a quel file, questa variabile è di tipo *puntatore* ed è detta ***file pointer*** :

  `FILE` *******variabile*;

  `FILE *fp;`

Il tipo FILE

- La struttura dati `FILE` contiene tra le informazioni relative al file il cosiddetto *offset* (o *file position pointer*) che memorizza il punto in cui la lettura o scrittura è arrivata (indica la posizione del prossimo **byte** da leggere o da scrivere, calcolata rispetto al primo byte del file che ha offset=0)
- Esempio per file in lettura



Apertura di un file

- Per utilizzare un file bisogna richiederne l'accesso al Sistema Operativo indicando anche per quale scopo (lettura o scrittura), questa operazione è detta *apertura* del file
- La funzione `fopen` apre un file, creando e riempiendo la struttura `FILE` da associargli `fp=fopen(`*nome*`, `*modo*`);`
- *nome* è una **stringa** (costante o variabile) contenente il nome del file e l'eventuale percorso, la dimensione massima e tipica di tale stringa è `FILENAME_MAX` (costante definita in `<stdio.h>`)
- *modo* è una **stringa** indicante la modalità di apertura del file (lettura o scrittura)

Apertura di un file

- Modi di apertura semplice per *file di testo:*
  - `"r"` – sola lettura (read), il file deve già esistere
  - `"w"` – sola scrittura (write), crea *vuoto* il file da scrivere; se esiste già, lo elimina e lo ricrea nuovo (quindi se ne perde il contenuto)
  - `"a"` – sola scrittura (append), se non esiste viene creato nuovo e vuoto, se invece esiste NON viene cancellato e i dati vengono aggiunti al fondo di quelli già presenti
- Le modalità `"r+"`, `"w+"`, `"a+"` sono dette *di aggiornamento* e permettono di leggere e scrivere lo stesso file (si veda più avanti)

Apertura di un file

- `fopen` restituisce `NULL` se non riesce ad aprire il file, altrimenti restituisce il puntatore alla struttura `FILE` corrispondente al file aperto, da assegnare al file pointer
  `fp=fopen(miofile,"r");`
- Bisogna verificare *sempre* il valore di `fopen`:
  - un file in input potrebbe non esistere (o il percorso essere sbagliato)
  - un file in output potrebbe non poter essere creato (ad es. si trova su un disco o su una chiavetta USB protetta dalla scrittura o il file da scrivere esiste già ed è protetto dalla cancellazione – read only)

Apertura di un file

- `FOPEN_MAX` (definito in `<stdio.h>`) indica il minimo numero di file che il processo può avere contemporaneamente aperti, a seconda dei sistemi può essere intorno a 20, ma normalmente se ne possono aprire di più (potrebbe creare problemi di portabilità)
- È possibile aprire contemporaneamente più file
- È possibile aprire contemporaneamente più volte lo stesso file, sia con lo stesso *modo* sia in *modi* diversi, ma bisogna fare attenzione alla *bufferizzazione* dei file in output (vedere più avanti)

Apertura di un file

- Lo schema tipico di apertura del file è:

```
printf("File da aprire: ");
gets(miofile);
fp=fopen(miofile,"r");
if (fp==NULL)
{
   fprintf(stderr, "Errore fopen\n");
   return EXIT_FAILURE;
}
```

`stderr` indica il video (descritto più avanti).

Poiché il nome del file potrebbe contenere spazi e `scanf("%s")` si ferma al primo spazio, è preferibile usare una `gets`

Apertura di un file

- Una forma più compatta di apertura è la seguente :

```
if ((fp=fopen(miofile,"r"))==NULL)
{
  fprintf(stderr, "Errore fopen\n");
  return EXIT_FAILURE;
}
```

Notare le parentesi evidenziate: permettono di assegnare *prima* il valore restituito dalla `fopen` a `fp` e *poi* di verificare se `fp` è `NULL`

Apertura di un file

- Se il nome del file è noto a priori, lo si può indicare come stringa costante nella `fopen`:

```
if ((fp=fopen("file.txt","r"))==NULL)
{
  fprintf(stderr, "Errore fopen\n");
  return EXIT_FAILURE;
}
```

ma è preferibile anche in questo caso (come per i *Magic numbers*) utilizzare una `#define`:

```
#define MIOFILE "file.txt"
if ((fp=fopen(MIOFILE,"r"))==NULL)
```

Apertura di un file

- Nei sistemi operativi Windows le parti di un percorso (*pathname* o *path*) sono separate dal carattere backslash `'\'`; se in una stringa costante si indica un percorso, poiché in C il backslash `'\'` è il carattere di escape, questo deve essere raddoppiato:
  `"C:\\documenti\\file.txt"`
- Non è invece necessario raddoppiarlo se viene inserito dall'utente (es. con una `gets`)

Chiusura di un file

- Informa il Sistema Operativo che è terminato l'utilizzo di un certo file
  `fclose(`*fp*`);`
- Libera le risorse di sistema legate a quel file
- È automatica al termine del programma (ma è buona norma chiuderlo non appena non serve più, in particolare se file di output)
- Nel caso di file di output, prima della chiusura effettua automaticamente il *flush* dei buffer (vedere più avanti)

Scrittura in un file di testo

- Ogni scrittura aggiunge caratteri al fondo del file (l'*offset* indica la posizione dove verrà scritto il prossimo byte)
- Le funzioni di output su file hanno lo stesso comportamento di quelle di output su video, salvo l'indicazione esplicita del *file pointer*
- `fprintf(`*fp*`,`*stringaFormato*`,`*listaEspressioni*`);` vedere `printf` ; dà `EOF` in caso di errore
- `fputc(`*var*`,` *fp*`);` manda il carattere *var* nel file *fp*; se *var* è `\n`, lo converte; dà `EOF` in caso di errore
- `fputs(`*str*`,` *fp*`);` come `puts`, salvo che **non aggiunge un** `'\n'` al fondo; dà `NULL` in caso di errore

Buffer di output

- Poiché le operazioni su disco sono lente, normalmente i dati diretti a file di output vengono temporaneamente accumulati in memoria in uno spazio detto *buffer*
- Quando il buffer è pieno o si esegue una `fclose` i dati vengono inviati tutti insieme sul disco e viene svuotato (*flush*)
- Se è necessario effettuare un flush del buffer associato a *fp* si usa la funzione:
  `fflush(`*fp*`);`
  Per svuotare i buffer di tutti gli stream aperti in output si può usare:
  `fflush(NULL);`

Lettura da un file di testo

- I caratteri vengono automaticamente prelevati dal file man mano che sono richiesti dalle funzioni di input (e l'offset avanza)
- Quando le funzioni leggono da un file, cercano di leggere il maggior numero di caratteri possibile in base al tipo di valore richiesto (es. se deve leggere un `int` la lettura termina solo quando incontra il primo carattere non-cifra)
- Ogni lettura inizia dal carattere successivo a quello dove era terminata la precedente lettura
- Le funzioni di input da file hanno lo stesso comportamento di quelle di input da tastiera, salvo l'indicazione del file pointer

Lettura da un file di testo

- `fscanf(`*fp*`,` *stringaFormato*`,` *listaVariabili*`);`
  restituisce il numero di elementi letti da *fp*, `EOF` in caso di errore e fine file, vedere `scanf`
- `x = fgetc(`*fp*`);`
  legge un carattere da *fp* e lo mette in `x` (che *deve* essere un `int`); restituisce `EOF` in caso di errore o fine file; restituisce `'\n'` dopo aver letto tutta la sequenza di caratteri che nel S.O. in uso costituiscono il ritorno a capo

Lettura da un file di testo

- `fgets(`*str*`, `*n*`, `*fp*`);`
  legge una riga intera da *fp* (ossia fino al `'\n'`) e la mette in *str* (**incluso il carattere** `'\n'`);
  legge fino a *n*`-1` caratteri: meno se la riga è più corta, mentre se la riga è più lunga gli altri caratteri restano per la lettura successiva; dà `NULL` in caso di fine file e di errore.
  str deve quindi avere spazio non solo per il `'\0'` di terminazione della stringa, ma anche per l'eventuale `'\n'` che può non esserci nell'ultima riga

Buffer di input

- Per efficienza, in una lettura da disco il sistema provvede a leggere più byte di quelli richiesti e li colloca in un buffer, così le prossime letture avvengono da questo e sono più veloci
- Non esiste una funzione standard che permetta di svuotare i buffer di input (ossia non si può usare `fflush` su stream di input)
- Alcuni compilatori permettono di svuotare i buffer di input utilizzando `fflush`, ma se si vuole un programma standard non la si usi e si trovino soluzioni alternative che non richiedano flush (non è detto esistano sempre)

fgetc restituisce un int

- La funzione `fgetc` preleva un singolo carattere da uno stream e deve:
  - restituire il codice ASCII del carattere letto
  - comunicare il raggiungimento della fine del file
- Poiché essa restituisce un solo valore, questo deve veicolare entrambe le informazioni
- I possibili diversi valori restituiti sono 257:
  - le 256 combinazioni degli 8 bit di un `char`
  - un altro valore (`EOF`) per segnalare la fine del file

quindi gli 8 bit di un `char` non bastano e ci vuole un tipo intero con almeno 9 bit: lo Standard ha scelto di usare un `int`

fgetc restituisce un int

- Esempio
  Si supponga che un `int` sia su 32 bit e la costante `EOF` sia definita pari a –1,
  il valore `int` restituito dalla `fgetc`:
  - nel caso non si sia a fine file: è il carattere letto e viene memorizzato il suo codice ASCII (es. 'A' = 01000001) preceduto da 3 byte a 0:

00000000	00000000	00000000	01000001

    N.B. per avere un dato di tipo `char` basta un cast
  - nel caso si sia a fine file: è –1 su 4 byte:

11111111	11111111	11111111	11111111

    N.B. –1 in Complemento a 2 è composto da tutti 1

Ciclo di lettura da file

- Bisogna sempre controllare se si è giunti alla fine del file
- Per determinare se si è raggiunta la fine del file (ossia non c'è più niente da leggere):
  - si controlla il valore restituito dalle funzioni di input *dopo* la lettura (restituiscono la segnalazione di fine file quando la lettura di un record *fallisce*, NON subito dopo aver letto l'ultimo valore)
  - si usa la funzione `feof` prima di fare la lettura

Ciclo di lettura da file

- Esempio corretto di lettura di interi da file (schema tipico di lettura):

```
while (fscanf(fp,"%d", &x) != EOF)
{
    istruzioni
}
```

  la funzione `fscanf` tenta di leggere un valore (uno solo), se il risultato è:
  - `EOF` : ha incontrato solo *eventuali* white spaces e poi il file è terminato, `x` non è cambiato
  - non `EOF` : ha letto un valore dal file e l'ha messo in `x`, l'offset viene fatto avanzare del numero di byte pari ai caratteri letti (se legge "12" avanza di 2 caratteri); in realtà potrebbe non aver letto nulla, ma di certo è che il file non è terminato (vedi →)

Ciclo di lettura da file

- In realtà la soluzione precedente *funziona solo se nel file ci sono soltanto valori interi*.
  Infatti se trova un carattere non-cifra la lettura non dà `EOF` (non è finito il file), ma dà 0 per indicare che non ha letto nulla (e la `x` non viene modificata), lasciando quel carattere nel buffer per la successiva lettura.
  Ma la successiva lettura tenta di rileggere quello stesso carattere, ossia la `fscanf` nel `while` continua indefinitamente, non termina mai (ciclo infinito, si dice che "*va in loop* ")

Ciclo di lettura da file

- Una soluzione migliore è la seguente:

```
while (fscanf(fp,"%d", &x) == 1)
…
```

Qui si controlla se la `fscanf` ha letto e *assegnato* un valore (ossia ha restituito 1), se non ha dato 1 può aver restituito:

- `EOF` : se ha incontrato solo *eventuali* white spaces e poi il file è terminato, `x` non ha cambiato valore
- `0` : se il file aveva dei caratteri ancora da leggere, ha consumato eventuali white spaces iniziali (è il comportamento del `%d`), ma poi ha trovato un carattere non-cifra (es. un carattere alfabetico), per cui si è fermata, non ha assegnato nulla a `x` ma non c'è EOF e il carattere (non i white space che lo precedevano) resta da leggere

Ciclo di lettura da file

- Esempio errato:

```
while (fscanf(fp,"%d%d",&x,&y)!=EOF)
{
    istruzioni
}
```

In questo esempio, se nel file è rimasto un solo valore, all'ultima iterazione la `fscanf` dà 1 e non `EOF` e viene eseguito il blocco di istruzioni con la nuova `x` e la precedente `y` (non è stata modificata)

Ciclo di lettura da file

- È più corretto specificare il risultato atteso per la funzione `fscanf`:

```
while (fscanf(fp,"%d%d",&x,&y)==2)
{
    istruzioni
}
```

Ma si ha ancora un problema: se il numero di valori nel file è dispari (il file non è corretto), l'ultimo viene letto comunque e messo in `x` (ma diversamente da prima esce subito dal ciclo senza usarlo), ma non viene segnalato il problema

Ciclo di lettura da file

- Per risolvere anche quest'ultimo problema bisogna verificare l'effettivo risultato della funzione `fscanf`:

```
while ((n=fscanf(fp,"%d%d",&x,&y))==2)
{
    elaborazione che utilizza x e y
}
if (n==1)
    ha letto solo x, mentre y è quello precedente
else
    ha terminato (n contiene EOF)
```

La funzione feof

- Per verificare se si è raggiunta la fine del file si può anche usare la funzione `feof)`, ma non è il metodo preferibile
- La funzione `feof(`*fp*`)` dà un risultato:
  - diverso da 0 se il file è terminato (è stato letto tutto)
  - pari a 0 se c'è almeno un byte da leggere (ma non indica quanti)
- Attenzione che in C la fine del file è rilevata *solo quando una funzione di input cerca di leggere dal file* e la `feof` non è una funzione di input: non la si può usare per stabilire se un file è vuoto prima di iniziare a leggerlo

La funzione feof

- Per *determinare se un file è vuoto*, si può provare a leggerlo e verificare se la lettura è andata a buon file. Poi o si "rimette il carattere nel file" con `ungetc` o si riporta l'offset all'inizio del file con la funzione `rewind`:

```
if (fgetc(fp)==EOF)
{ fprintf(stderr, "File vuoto\n");
  return EXIT_FAILURE;
}
rewind(fp);
while (!feof(fp)) /* ancora prob! */
{
  fscanf(fp, "%d", &x);
  printf("%d", x);
}
```

La funzione feof

- Il ciclo `while` precedente ha ancora un problema: se l'ultimo valore del file è seguito da *uno o più white spaces*, la `feof` indica (correttamente) che il file non è terminato, quindi il ciclo continua e l'esecuzione della `fscanf` non va a buon fine (dà `EOF` ma non viene controllata): `x` non viene assegnata e mantiene il valore precedente, quindi la `printf` lo visualizza una volta di troppo
- Per risolvere il problema si può controllare anche la `printf` interna, ma è molto meglio evitare `feof` e preferire la forma:

```
while (fscanf(fp, "%d", &x) == 1)
```

La funzione feof

- Anche la lettura effettuata con `fgetc` può creare problemi:

```
while (!feof(fp))
{
   x=fgetc(fp);
   istruzioni che utilizzano x
}
```

- La `fgetc` legge un solo carattere e non cerca di leggere oltre questo, quando legge l'ultimo non può sapere che è proprio l'ultimo e quindi permette erroneamente un'altra iterazione, dove la `fgetc` restituisce `EOF` e questo viene usato come – 1 dalle *istruzioni che utilizzano* `x`

La funzione feof

- Per risolvere il problema quando è necessario leggere singoli caratteri, si può utilizzare uno schema come il seguente:

```
while ((x=fgetc(fp) != EOF)
      elabora x;
```

- Quando possibile, si eviti di usare `feof`

Rimandare indietro un car letto

- La funzione `ungetc(`*c*`, `*fp*`)` "rimanda" il carattere *c* nel file *fp* come se non fosse mai stato letto e azzera gli indicatori di EOF
- Il carattere *c* non viene in realtà "rimandato" nel file, ma viene collocato in un buffer di 1 byte e tutte le funzioni di input prima di prelevare i caratteri dal file prelevano il byte dal buffer (se c'è)
- Si può rimandare indietro *un solo* carattere se il buffer è vuoto (inizialmente e dopo ogni lettura è vuoto), quindi non si può scrivere un file con chiamate successive a `ungetc`...

Rimandare indietro un car letto

- Esempio di utilizzo di `ungetc`
  Estrarre i numeri interi all'interno di parole (ad es. da "`Beta123abc`" estrarre il numero `123`)

```
while((c=fgetc(fp))!=EOF &&
        !isdigit(c))
   ;
ungetc(c, fp);
fscanf(fp, "%d", &x);
```

  - Il ciclo `while` continua a leggere e scartare caratteri fino a trovare una cifra
  - Quando trova l' `1` esce e `unget` lo rimanda in `fp`
  - Ora la `fscanf` può leggere il `123`
  - In un codice reale bisognerebbe considerare che potrebbe non esserci alcuna cifra

Stream preesistenti

- Quando il programma viene mandato in esecuzione, il sistema operativo gli fornisce 3 stream già aperti:
  - `stdin` collegato alla tastiera
  - `stdout` collegato al video
  - `stderr` collegato al video (non bufferizzato)
- Vi sono due stream di output perché:
  - `stdout` dovrebbe essere utilizzato solo per visualizzare i risultati dell'elaborazione
  - `stderr` dovrebbe essere utilizzato solo per la *diagnostica* (gli errori del programma, non sono i dati prodotti come risultato dell'elaborazione)

```
fprintf(stderr, "File vuoto");
```

Stream preesistenti

- Si noti che:
  `fprintf(`**`stdout`**`,"%d",…);`
  equivale in tutto alla funzione
  `printf("%d",…);`
- L'interprete dei comandi della console può essere impostato in modo da mandare su dispositivi di output diversi gli stream dei risultati (`stdout`) e quello degli errori (`stderr`) mediante la redirezione dell'I/O, le modalità dipendono dal SO utilizzato, a titolo di esempio in DOS/Windows/Linux/OSX si ha:
  - `programma >`*file_per_stdout*
  - `programma 2>`*file_per_stderr*
  - `programma >`*file_per_stdout* `2>`*file_per_stderr*
  - `programma >`*file_per_stdout_e_stderr* `2>&1`

Altre funzioni sui file

- La funzione `freopen(`*nome*`, `*modo*`, `*fp*`)`:
  - chiude lo stream puntato da *fp* (se è aperto, ignora eventuali errori)
  - apre il file *nome* con la modalità *modo* e lo associa al file pointer *fp*
  - Restituisce `NULL` se non ha successo e lo stesso *fp* se ha successo

Altre funzioni sui file

- L'uso principale (ma non unico) della `freopen` è modificare il file associato a uno stream standard (`stderr`, `stdin`, `stdout`):

```
if (freopen("log.txt", "w",
                          stdout)==NULL)
{
   /* errore */
}
```

dopo questa apertura le `printf` scriveranno nel file indicato invece che sul video

Altre funzioni sui file

- `remove(`*filename*`)` cancella il *filename* (stringa) dal disco, restituisce 0 se ha esito positivo
- `rename(`*filevecchio*`,` *filenuovo*`)` cambia nome a *filevecchio* (stringa) sul disco, restituisce 0 se ha esito positivo
- `rewind(`*fp*`)` riporta l'offset del file all'inizio in modo che la successiva lettura/scrittura inizi dal primo carattere; azzera l'indicatore di errore e di `EOF`
- `fseek`/`ftell` e `fgetpos`/`fsetpos` riposizionano l'indicatore dell'offset del file, per i file di testo ci possono essere problemi (descritti per i file binari)

Altre funzioni sui file

- `setvbuf` e `setbuf` permettono di cambiare la modalità di buffering di uno stream e di controllare la dimensione e l'allocazione del buffer

Gestione degli errori

- Le funzioni di input in caso di *errore* o *fine file* danno una generica segnalazione (`EOF` o `NULL`), ma si tratta di condizioni ben diverse che il sistema gestisce impostando due indicatori indipendenti:
  - *end-of-file indicator*
  - *error indicator*
- Per distinguere i due casi si usano le funzioni `feof` e `ferror`, ciascuna delle quali verifica il corrispondente indicatore

Gestione degli errori

- Le seguenti funzioni richiedono `<stdio.h>`
- `ferror(`*fp*`)` dà un valore intero non nullo se si è verificato un EOF sullo stream *fp*
- `ferror(`*fp*`)` dà un valore intero non nullo se si è verificato un errore (non EOF) sullo stream *fp*
- `clearerr(`*fp*`)` cancella le indicazioni di errori e di `EOF` relativi a *fp*

Gestione degli errori

- Se la lettura di valori mediante `fscanf` produce un valore minore di quello atteso:
  - se `feof` è vera: raggiunta fine file
  - se `ferror` è vera: errore di lettura
  - altrimenti c'è un errore sul'la corrispondenza del tipo di dati (*matching failure*, es. la stringa di formato ha un `%d` e il dato in input non è numerico)

```
if (fscanf(fp, "%d", &a) !=1)
  if (feof(fp))                  → fine del file
    …
  else if (ferror(fp))           → errore nel file
    …
  else                           → matching failure
    fscanf(fp, "%*[^\n]");  es. svuota la riga
```

Gestione degli errori

- Quando si verifica un errore, le funzioni sui file (e altre) assegnano alla variabile `errno` (richiede `<errno.h>`), un valore intero che lo identifica, vedere la documentazione del compilatore per i valori)
- `perror(`*stringa*`)` visualizza *stringa* e un messaggio di errore corrispondente al valore in `errno`, richiede `<stdio.h>` e `<errno.h>`
- `strerror(`*n*`)` restituisce un puntatore a una stringa contenente la descrizione dell'errore il cui codice è *n* (normalmente per *n* viene indicato `errno`), richiede `<string.h>`

Modalità di aggiornamento

- L'apertura di un file in modalità di aggiornamento permette la lettura e la scrittura dello stesso file mediante lo stesso file pointer (`fp`)
- Modi di apertura in aggiornamento:
  `"r+"` – lettura/scrittura, il file deve esistere
  `"w+"` – lettura/scrittura, crea il file vuoto
  `"a+"` – lettura/scrittura al fondo, lo crea se non esiste, aggiunge alla fine se esiste già
- La differenza tra `"r+"` e `"w+"` è solo nell'apertura:
  `"r+"` → se il file non esiste dà errore
  `"w+"` → se il file esiste già, lo ricrea vuoto

Modalità di aggiornamento

- Il modo `"a+"` non cancella il contenuto di un file già esistente, posiziona il file position pointer alla fine del file e permette di scrivere *solo in fondo al file*
- Anche se con `fseek` si sposta l'offset indietro, la scrittura avviene sempre aggiungendo record in fondo al file e i dati precedenti non possono quindi essere modificati (neppure quelli appena scritti, nemmeno se il file non è stato ancora chiuso)
- Con opportuna `fseek` è possibile leggere i dati da qualsiasi punto del file

Modalità di aggiornamento

- Tra un'operazione di scrittura e una di lettura (o viceversa) sullo stesso file è necessario che ci sia una chiamata a `fflush` o a una funzione di posizionamento dell'offset quale: `rewind`, `fseek` o `fsetpos`
- Se si deve leggere dal file un qualcosa che è possibile sia ancora nei buffer, è necessario chiamare la funzione `fflush`

Considerazioni sull'efficienza

- Tutte le operazioni di I/O relative ai file sono intrinsecamente e notevolmente più lente di quelle che usano solo la memoria, nonostante la presenza di buffer e cache
- Conviene ridurre il più possibile l'accesso al file (ma se è molto piccolo potrebbe essere contenuto interamente nel buffer e l'accesso al file essere un mero accesso alla memoria)
- Quando è necessario utilizzare molte volte i dati contenuti in un file, conviene in genere caricarne il contenuto in memoria in opportune strutture dati

File temporanei

- Quando è necessario creare un file temporaneo per la sola durata dell'esecuzione del programma, si possono usare le seguenti funzioni:
  - `tmpfile()` – crea un file temporaneo in modalità `"wb+"` (non è un file di testo ma binario!) e restituisce un file pointer; viene chiuso o con `fclose` o automaticamente a fine; NULL in caso non riesca ad allocare

File temporanei

- `tmpnam(`*val*`)` – crea un nome di file temporaneo e ne restituisce il puntatore `char*`, `NULL` se fallisce
  Se il parametro *val* è `NULL` alloca il nome in una stringa costante (memoria statica)
  Se viene passata come argomento una stringa, viene memorizzato in essa (`tmpnam` può generare al massimo `TMP_MAX` stringhe temporanee)
  La stringa deve essere definita di lunghezza `L_tmpnam` (macro definita in `<stdio. h>`):

```
char filetemp(L_tmpnam);
…
tmpnam(filetemp);
```

Esercizi

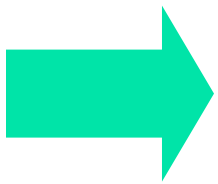
1. Scrivere con un editor un file contenente dei numeri, uno per riga. Sviluppare un programma che chieda all'utente il nome di questo file, lo apra e calcoli e stampi a video quanti sono i valori letti, il valore massimo, il valore minimo, la somma e la media di tutti i valori presenti nel file.
2. Scrivere un programma che chieda all'utente il nome di un file di testo e conti quante righe e quanti caratteri esso contiene (non si considerino i caratteri di ritorno a capo ed eventuale fine file).

Esercizi

3. Scrivere un programma che legga un file contenente coppie di numeri interi (x e y), una per riga e scriva un secondo file che contenga le differenze $x - y$, una per riga.
   Si supponga che il file di input non abbia errori e che nomi dei file vengano chiesti all'utente.

File 1

```
23    32
2     11
19    6
23    5
3     2
...   ...
```



File 2

```
-9
-9
13
18
1
...
```

Esercizi

4. Si scriva un programma che chieda il nome di un file contenente un testo qualsiasi e di questo conti quante sono le parole che iniziano con ciascuna lettera dell'alfabeto.
   Esempio di output:

```
Parole che iniziano con A: 45
Parole che iniziano con B: 12
Parole che iniziano con C: 27
…
Parole che iniziano con Z: 3
```

Esercizi

5. Un file di testo contiene, in ciascuna riga separati da spazi, 2 o 3 valori reali:
   1. nel primo caso si tratta della base e dell'altezza di un triangolo
   2. nel secondo caso si tratta della base minore, della base maggiore e dall'altezza di un trapezio

   Si scriva un programma che chieda all'utente il nome di questo file, lo analizzi e indichi qual è la riga del file corrispondente alla figura geometrica di area maggiore e il valore di quest'area. Se si trovano figure di aree uguali, si consideri la prima di queste.

Esercizi

6. Scrivere un programma che legga da un file un testo e spezzi su più righe quelle più lunghe di N caratteri (N chiesto all'utente). La riga viene spezzata anche in mezzo ad una parola o prima di uno spazio. L'output deve essere salvato in un altro file. Ad esempio, se viene letta la riga seguente (è una sola, qui è scritta su due righe perché troppo lunga):

```
Mi devo recare all'ufficio postale
precipitevolissimevolmente
```

l'output (con N pari a 22) deve essere:

```
Mi devo recare all'uf
ficio postale precipi
tevolissimevolmente
```

Esercizi

7. Come il precedente, ma le righe si possono spezzare solo dove c'è uno spazio (che non va visualizzato a capo), se negli N caratteri non c'è uno spazio, si spezzi la parola all'N-esimo carattere come nell'esercizio precedente.

   Esempio di output con N pari a 22:

```
Mi devo recare
all'ufficio postale
precipitevolissimevolm
ente
```

Esercizi

8. Merge di due file
   Due file di testo contengono, disposti variamente su più righe, dei numeri interi. Ciascuno dei due file è già ordinato in senso CRESCENTE e non è noto a priori quanti elementi contenga (possono anche avere un numero di valori differente).
   Si scriva un programma che legga questi due file e ne produca un terzo contenente i valori dei due file ordinati tutti quanti in senso CRESCENTE. Non si utilizzino vettori né algoritmi di ordinamento. I nomi dei tre file vengano chiesti all'utente.

Esercizi

9. Il cifrario di Cesare
   Si scriva un programma che cifri/decifri un file di testo in base a un codice segreto. Il programma deve chiedere il nome del file di input, il nome del file di output e il codice segreto (un numero intero). La cifratura avviene sostituendo le sole lettere (maiuscole e minuscole separatamente) con altre lettere in base al valore del codice.

Esercizi

(*Continuazione*)

Il valore del codice indica di quanti caratteri si deve traslare ciascuna delle lettere: se ad es. il codice è 3, allora 'A' → 'D', 'B' → 'E', ... 'X' → 'A', 'Y' → 'B', 'Z' → 'C'.

La corrispondenza completa delle lettere maiuscole (per le minuscole è analogo) è:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Ad esempio, con codice=3 "`Ciao`" diventa "`Fldr`". Si noti che per decifrare un testo basta inserire come codice il valore opposto a quello usato per la cifratura (qui –3).

Esercizi

10. Simile al precedente, ma la sostituzione delle lettere avviene secondo il seguente algoritmo. Il *codice segreto* di cifratura/decifratura è costituito da una parola composta di sole lettere tutte diverse. La prima lettera del codice viene fatta corrispondere alla lettera 'A', la seconda alla 'B', e così via fino all'ultima lettera del codice. Le lettere non indicate nel codice vengono riordinate alfabeticamente al contrario (dalla Z alla A) e fatte corrispondere alle restanti lettere dell'alfabeto (maiuscole e minuscole separatamente).

Esercizi

(*Continuazione*)

Il programma chiede l'introduzione di una *parola chiave* da cui, eliminando i caratteri duplicati e le non-lettere, si ha il *codice segreto*. Ad esempio, dalla parola chiave "`Fragola`" si ottiene il codice segreto "`FRAGOL`" e la corrispondenza delle lettere è:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
F	**R**	**A**	**G**	**O**	**L**	Z	Y	X	W	V	U	T	S	Q	P	N	M	K	J	I	H	E	D	C	B

Ad esempio, la parola "`Ciao`" viene cifrata in "`Axfq`" (notare maiuscole e minuscole).

Il programma deve chiedere se si vuole cifrare o decifrare il file.

Esercizi

11. Si scriva un programma che chieda all'utente il nome di un file contenente la griglia di valori di un Sudoku risolto (N righe con N valori). Il programma deve verificare ciascuna riga, ciascuna colonna e ciascun riquadro e segnalare le righe, le colonne e i riquadri che non soddisfano le condizioni di validità. Condizione di validità per righe, colonne e riquadri: non ci devono essere valori ripetuti. Il programma accetti automaticamente griglie di lato massimo 25 numeri.

Esercizi

12. Una superficie viene descritta da una matrice di valori in floating-point in doppia precisione composta di R righe e C colonne (costanti note a priori). Ogni elemento della matrice corrisponde a un quadrato di dimensioni 1cm x 1cm. I valori rappresentano l'altezza media di quella superficie in quel punto e sono letti da un file. Tale file contiene C valori (separati da spazi) per ciascuna delle R righe e non ha errori. Il primo elemento della matrice corrisponde alla parte di superficie situata più a nord-ovest e ha coordinate (0,0).

Esercizi

(*Continuazione*)

Si noti che le coordinate X corrispondono alle colonne, le Y alle righe.

Si suppone di voler simulare il comportamento di una pallina che viene collocata su uno dei riquadri della superficie (le cui coordinate sono richieste all'utente) e poi lasciata libera. Il programma deve visualizzare la sequenza delle coordinate di tutte le posizioni che vengono attraversate dalla pallina fino a quando si ferma, inclusa la posizione iniziale, con l'indicazione dell'altezza in quel punto.

Esercizi

(*Continuazione*)

Si usa un modello fisico semplificato dove la pallina:

1. può scendere in ciascuna delle 8 caselle che la circondano (se ci sono, ossia non è ai bordi)
2. scende lungo il cammino più ripido, ossia delle 8 caselle raggiunge quella più in basso (se ce ne sono più di una all'altezza minima si scelga la prima trovata)
3. non ha inerzia (cioè scende sempre lungo il cammino più ripido)
4. non può/deve uscire dai limiti della superficie
5. si ferma quando non ci sono caselle più in basso della posizione che occupa

Esercizi

(*Continuazione*)

```
Posizione iniziale
- coordinata X (1-20): 5
- coordinata Y (1-30): 3

Percorso della pallina
----------------------
0 - (5,3) [20.42]
1 - (6,2) [15.21]
2 - (7,3) [11.11]
3 - (7,4) [7.235]
4 - (8,5) [3.66]
5 - (9,4) [-1.7]
6 - (9,3) [-3.45]
```

Esercizi

13. Si modifichi l'esercizio precedente considerando un modello in cui la pallina possa spostarsi solo in diagonale
14. Come il precedente, ma la pallina possa spostarsi solo in direzione nord/sud/est/ovest (orizzontale/verticale).

Homework 5

Si scriva un programma per gestire un'agenda elettronica. I dati relativi alle persone sono: `nome`, `cognome`, `indirizzo`, `telefono` e `nota`; NON possono contenere spazi e sono contenuti in un file di testo denominato `database.txt`. Viene presentata un'interfaccia a menu per a) **inserire**, b) **togliere**, c) **cercare** i dati relativi ad una persona (solo in base al cognome), d) **visualizzare tutte** le coppie nome/cognome senza dettagli ed e) **uscire** dal programma. Le persone sono al massimo 200, i dati devono essere copiati tutti in memoria quando il programma parte e scritti solo alla fine.