File binari

Ver. 3

File binari

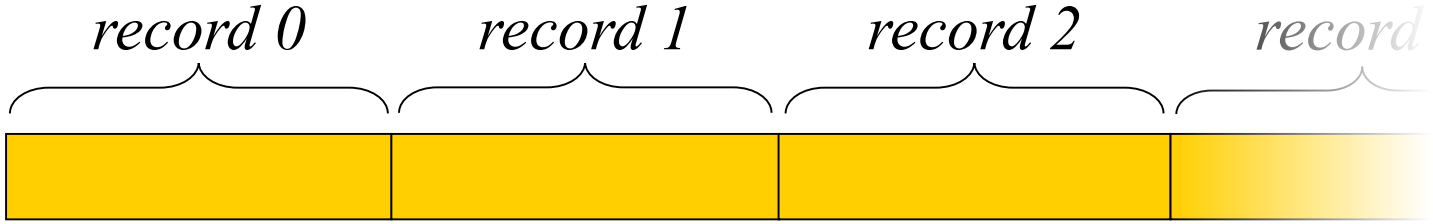
- I file binari non sono file di testo: non contengono righe di caratteri ASCII terminate da '`\n`', ma sono una generica sequenza di byte "grezzi", potrebbero contenere dati di qualsiasi tipo: immagini, suoni, ecc.
- Nei file di testo il *carattere* '`\n`' quando viene scritto su file viene convertito nella sequenza di ritorno a capo propria del Sistema Operativo (es. CR-LF) e viceversa nella lettura
- Nei file binari non c'è alcuna conversione: se viene letto il corrispondente ASCII del '`\n`', viene in realtà letto un byte con valore 10

File binari

- Nei file binari si parla più propriamente di byte (anche se si usa comunque il termine carattere) e nessuno di questi ha un significato speciale
- Quando i byte sono raggruppati in blocchi, questi vengono chiamati ***record***
- Ogni record viene idealmente letto/scritto in un'unica operazione (tutti i suoi byte insieme)
- Un blocco di byte può essere allocato definendo una variabile, un vettore o mediante le funzioni di allocazione dinamica (`malloc`)

File binari

- Quando i record hanno tutti la stessa dimensione, l'accesso al record n-esimo è molto veloce in quanto, per determinare il numero del byte da dove esso inizia rispetto all'inizio del file (il suo *offset*), basta calcolare: $n * L$, dove L è la lunghezza dei record in byte, e pozionare lì il *file position pointer*
- Si parla allora di *file ad acceso diretto*, detti anche *file con accesso random* (*file random*)



Modalità di apertura

- I file binari si aprono con `fopen` indicando i modi binari (contengono la lettera `b`):
  `"rb"`, `"wb"`, `"ab"`, `"r+b"` o `"rb+"`, `"w+b"` o `"wb+"` e `"a+b"` o `"ab+"`
- La differenza tra `"r+b"` e `"w+b"` è solo nell'apertura:
  `"r+b"` → se il file non c'è, dà errore,
  `"w+b"` → anche se il file c'è, lo ricrea vuoto
- Nelle modalità `"ab"` e `"a+b"` il *file position pointer* (offset) si posiziona in modo da aggiungere dati in fondo al file

Modalità di apertura

- Il modo `"a+b"` è simile a `"w+b"`, salvo che permette di scrivere solo in fondo al file; anche se con `fseek` si può spostare l'offset indietro, la scrittura è ammessa solo al fondo del file e i dati precedenti non possono essere modificati
- La chiusura dei file binari si ottiene con la funzione `fclose`
- I file binari non sono portabili in quanto i dati composti da più byte possono essere memorizzati in modi diversi su architetture diverse (es. *big-endian* o *little-endian*)

Lettura

- La lettura di blocchi di byte si ottiene mediante la funzione
  `fread(`*p*`, `*dim_oggetto*`, `*num_oggetti*`, `*fp*`);`
  - legge dal file *fp* un numero di oggetti pari a *num_oggetti* (`size_t`) ciascuno di dimensione pari a *dim_oggetto* byte (`size_t`) e li colloca nel **vettore** di oggetti (allocato!) puntato da *p*
  - restituisce (`size_t`) il numero di *oggetti* (non di byte) effettivamente letti (può essere inferiore a *num_oggetti* in caso di errore o fine file)
  - fa avanzare il file position pointer del numero di *byte* effettivamente letti
  - si devono usare le funzioni `feof` e `ferror` per distinguere gli errori dalla fine del file

Lettura

- Esempi
  - `struct X var1, var2;`
    `fread(&var1, sizeof(var1), 1, fp);`
    `fread(&var2, sizeof(struct X), 1, fp);`
    ogni `fread` legge **un** blocco di byte (oggetto, record) della dimensione di una `struct X` e lo mette nella variabile di cui viene fornito il puntatore (rispettivamente `var1` e `var2`)
  - `struct X vet[100];`
    `fread(vet, sizeof(struct X), 100, fp);`
    vengono letti **100** record corrispondenti a 100 `struct X` e memorizzati ordinatamente nel vettore `vet`

Lettura

- Definito il blocco di byte mediante `char x[DIM],`
  si noti la differenza tra le istruzioni:
  - `fread(x, DIM, 1, fp);`
    legge 1 blocco di dimensione DIM (byte) e lo salva in `x` , restituisce 1 se l'istruzione riesce a leggere effettivamente DIM byte
  - `fread(x, 1, DIM, fp);`
    legge un numero DIM blocchi di dimensione 1 (byte) e li salva in `x`, restituisce il numero di caratteri effettivamente letti

  In entrambi i casi vengono letti DIM byte, ma il valore restituito è diverso

Scrittura

- La scrittura di un blocco di byte si ottiene mediante la funzione

  `fwrite(`*p*`,` *dim_oggetto*`,` *num_oggetti*`,` *fp*`);`

  - scrive nel file *fp* un numero di oggetti pari a *num_oggetti* (`size_t`) ciascuno di dimensione pari a *dim_oggetto* byte (`size_t`) prelevandoli dal **vettore** di oggetti puntati da *p*
  - restituisce il numero di *oggetti* (non di byte) (`size_t`) effettivamente scritti, può essere inferiore a *num_oggetti* in caso di errore
  - fa avanzare il file position pointer del numero di *byte* effettivamente scritti

Scrittura

- Esempi
  - `struct X var;`
    `fwrite(&var, sizeof var, 1, fp);`
    `fwrite(&var, sizeof(struct X), 1, fp);`
    ogni `fwrite` scrive un blocco di byte (oggetto, record) della dimensione di una `struct X` prelevandolo dalla variabile `var` di cui viene fornito il puntatore (qui ce n'è uno solo)
  - `struct X vet[10];`
    `fwrite(vet, sizeof(struct X), 10, fp);`
    vengono scritti 10 record della dimensione di una `struct X` prelevandoli ordinatamente dal vettore `vett`

Scrittura

- *Attenzione!*
  Per scrivere in due record successivi la stessa `var` È SBAGLIATO scrivere:
  ```
  fwrite(&var, sizeof var, 2, fp);
  ```
  In questo modo la `fwrite` scrive sul disco `2*sizeof(var)` byte a partire dal primo byte di `var`, sforando la dimensione di `var`, trattandolo come se fosse un vettore di 2 elementi
- Bisogna invece scrivere 2 `fwrite`:
  ```
  fwrite(&var, sizeof var, 1, fp);
  fwrite(&var, sizeof var, 1, fp);
  ```

Lettura e scrittura di blocchi

- Le funzioni `fread` e `fwrite` leggono e scrivono un *blocco di byte* di dimensione pari a *dim_oggetto* * *num_oggetti*
  Le funzioni ricevono un semplice puntatore al blocco e non hanno quindi alcuna informazione sul contenuto del blocco stesso: potrebbe essere una variabile o un vettore qualsiasi
- Solo quando quel blocco viene letto dalla `fread` e gli viene fatto un cast (anche implicito) assegnandolo a una variabile esso diventa un oggetto dal contenuto ben preciso

Lettura e scrittura di blocchi

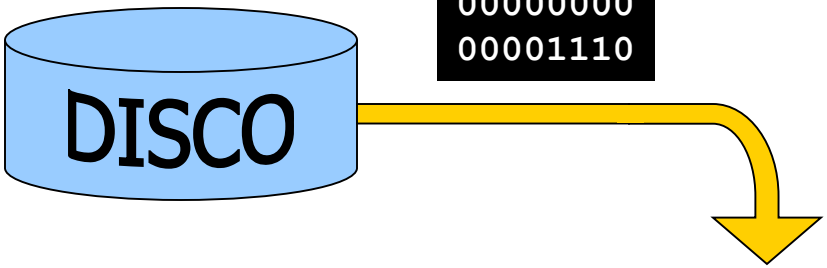
- Ad esempio
  - ```
    struct parola
    {
        char word[5];
        int num;
    } var; /* alloca memoria per var */
    fread(&var, sizeof(var), 1, fp);
    ```
    È solo dopo che il blocco letto è stato collocato in `var` che il suo contenuto è identificabile come una variabile `char*` e un'altra `int`

    Si può pensare che all'intero blocco letto sia stato fatto un cast a `struct parola`, ossia che al blocco di 5 byte sia stato fatto un cast a vettore di `char` e al blocco di 4 byte successivo un cast a `int`
    ```

Lettura e scrittura di blocchi

Blocco di byte letto

```
01000011
01000011
01000001
01001111
00000000
????????
00000000
00001110
```




```
struct parola
{
     char word[5];
     int num;
};
```

È la `struct` che dà significato ai byte del blocco

`char word[5];`

```
01000011
01000011
01000001
01001111
00000000
```

→ “CIAO\0”

Area di allineamento

```
????????
```

`int num;`

```
00000000
00001110
```

→ 14

Inizializzazione

- In alcuni casi è utile prevedere una inizializzazione del file con strutture vuote, ad esempio per rendere più semplice verificare se un record è già stato scritto

```
struct X vett[MAX], vuoto={"",0};
fp = fopen(file, "wb");
for (i=0; i<MAX; i++)
 fwrite(&vuoto, sizeof vuoto,1,fp);
fclose(fp);
```

Come già detto, non si può sostituire al ciclo `for` una `fwrite` con *num_oggetti* pari a MAX

Posizionamento

- Le funzioni `fread` e `fwrite` posizionano automaticamente il file position pointer (offset) all'inizio del record successivo
- Il posizionamento generico si ottiene con la funzione `fseek` (che annulla l'indicazione eventuale di EOF e svuota il buffer di `ungetc`)

  `fseek(` *fp*`,` *offset*`,` *origine*`);`
  - *offset* (di tipo `long`) indica su quale carattere, a partire da *origine*, spostare il file position pointer, negativo per indicare un valore precedente l'*origine*
  - *origine* indica da dove calcolare l'*offset*:
    - `SEEK_SET` → da inizio file
    - `SEEK_CURR` → dalla posizione corrente
    - `SEEK_END` → da fine file (offset negativo)

Posizionamento

- Esempio
  `fseek(fp, 20, SEEK_SET);`
  sposta il file position pointer al 21º carattere del file (il primo ha posizione 0)
- La `fseek` può indicare un offset oltre la fine del file, questo "allunga" il file (i byte non scritti hanno valore indeterminato)
- Per conoscere la posizione attuale del file position pointer (quindi l'offset) si può usare la funzione `ftell`

  `posiz = ftell(`*`fp`*`)`
  `posiz` è una variabile di tipo `long`
  `ftell` restituisce `-1L` in caso di errore

Posizionamento

- Per tornare all'inizio del file si può usare la funzione `fseek`:
  `fseek(`*fp*`, 0, SEEK_SET);`
- Ma la soluzione migliore è:
  `rewind(`*fp*`);`
  che oltre a eseguire la `fseek` precedente esegue anche la `clearerr(`*fp*`)` che azzera l'indicazione di errori (l'eventuale indicazione di raggiungimento della fine del file è eliminata anche dalla `fseek`)

Posizionamento

- Altra possibilità per spostarsi in un file è data da `fgetpos/fsetpos`:
  - `fgetpos(`*fp*`, `*ptr*`)`
    memorizza in *ptr* l'offset corrente del file *fp*, dà valore 0 in caso di successo e ≠0 in caso di errore
  - `fsetpos(`*fp*`, `*ptr*`)`
    posiziona l'offset corrente del file *fp* con il valore indicato in *ptr*, dà valore 0 in caso di successo e ≠0 in caso di errore
  - *ptr* deve essere definito con:
    `fps_t *`*ptr*`;`
    il tipo `fps_t` è un tipo (non necessariamente un valore numerico) appropriato per contenere il valore di un qualsiasi offset

Posizionamento

- Le funzioni `fseek/ftell` usano valori `long`, quindi valori qualsiasi impostabili dal programmatore, ma il tipo `long` potrebbe essere una limitazione se il file ha lunghezza tale da superare `LONG_MAX`
- Le funzioni `fgetpos/fsetpos`, usando un tipo specifico (`fps_t`), non hanno limitazioni sulla lunghezza del file, ma possono essere usate solo per riposizionare il file position pointer

Troncamento di file

- La libreria standard del C non fornisce alcuna funzione in grado di eliminare la parte finale di un file
- Possono esistere funzioni proprietarie non standard quali `truncate`, `ftruncate` e altre
- Non è altresì possibile mediante funzioni standard eliminare dati dalla testa di un file
- La soluzione standard per entrambi i casi consiste nel creare un nuovo file copiandovi i dati da tenere ed escludendo quelli da rimuovere

Esercizi

1. Si scriva un programma per gestire un'agenda elettronica. I dati relativi alle persone sono: nome, cognome, indirizzo, telefono, nota, *possono contenere spazi* e sono organizzati in una `struct persona`. Si definisca una dimensione massima per il numero di nominativi. Un file binario ad accesso diretto denominato `database.dat` contiene i record di tutti i nominativi.
   (continua)

Esercizi

(*Continuazione*)

Il programma deve fornire un'interfaccia a menu per inserire, cancellare, cercare i dati relativi ad una persona (in base al cognome), visualizzare alfabeticamente tutti i cognomi/nomi senza dettagli. I dati completi siano mantenuti *esclusivamente sul file*, in memoria si tenga invece un *indice alfabetico* dei nomi (vettore di strutture contenenti solo cognome, nome e numero del record sul file) per accedere al nominativo.