# File – Riassunto dai Fondamenti

- Cos'è un file?
  - È un'***astrazione fornita dal sistema operativo***, per consentire la memorizzazione di informazioni su memoria di massa
  - È un'astrazione di memorizzazione di ***dimensione potenzialmente illimitata ad accesso sequenziale***
- Cosa occorre fare per operare correttamente su un file?
  - Conoscere il nome assoluto
  - Aprire il file
    - Formato: bin / txt
    - Modo: read / write / append
  - Leggere e scrivere sul file ricordandosi che le due operazioni implicano uno stato mantenuto dalla testina di lettura/scrittura → esiste un concetto di posizione corrente
  - Leggere e scrivere (e comportarsi) in modo diverso a seconda che il file sia di testo (**`fscanf`**, **`fprintf`**, **`fgets`**, **`fputs`**, **`fgetc`**, **`fputc`**,...) o binario (**`fread`**, **`fwrite`**)
  - Ricordare **assolutamente** di **chiudere** il file al termine delle operazioni → un file aperto è una risorsa che non può essere utilizzato da nessun'altro!!!

1

# File binari

- In un file binario i dati sono memorizzati in un formato non *user friend*, ma molto *machine friend*
- Tipicamente si leggono e scrivono direttamente porzioni di memoria → l'importante è conoscere le dimensioni esatte di ciò che si sta leggendo

```
unsigned int fread(void *addr, unsigned int dim,
  unsigned int n, FILE *f);
unsigned int fwrite(void *addr, unsigned int dim,
  unsigned int n, FILE *f);
```

Lettura/scrittura da/su file **`f`** di **`n`** elementi di dimensione **`dim`** sull'/dell'area di memoria che parte da **`addr`**; restituzione del numero di elementi effettivamente letti/scritti

2

## File binari

- È bene sapere che in realtà la *signature* dei metodi non è esattamente quella vista, ma

```
size_t fread(void *addr, size_t dim, size_t n,
  FILE *f);
```

**`size_t`** è un tipo definito tramite **`typedef`** ed è il tipo restituito dall'operatore **`sizeof`**

**`size_t`** è mappato su un intero senza segno → **`unsigned int`**

Perché proprio **`unsigned int`**?

Scrittura in un numero negativo di elementi di dimensione negativa??

3

## File binari – Pattern

- È indispensabile conoscere ordine e dimensione dei dati letti e scritti
- Ordine → dipende solo dalle convenzioni "interne" all'applicazione → dipende dalle scelte del programmatore
- Dimensione → usare l'operatore **`sizeof(...)`**
- Punti critici:
  - Terminazione del file
  - Passaggio del valore giusto come parametro **`addr`** a **`fread`** o **`fwrite`**
    - Se array, l'array stesso (è un indirizzo!)
    - Se puntatore, il puntatore stesso
    - Se altro, l'indirizzo (**`&`**)!

4

## Person & Address – Definizioni

```
#ifndef PERSONTYPEDEFS
#define PERSONTYPEDEFS

typedef struct addressStruct
{
    char street[80];
    char postalCode[8];
    char city[30];
    char state[20];
} Address;
```

```
typedef struct personStruct
{
    char firstName[50];
    char secondName[50];
    char phone[18];
    char cell[18];
    Address address;
} Person;

#define PERSONARRAYDIM 100

typedef Person
  PersonArray[PERSONARRAYDIM];

#endif
```

5

## Person & Address – Persistenza

- Aggiungere i servizi che mancano per rendere "funzionale" l'applicazione
- Fondamentalmente → Persistenza dei dati... altrimenti occorre ricominciare da capo tutte le volte che si attiva l'applicazione
- Creare una funzione **`readFromBin`** che, dato un il nome di un file ed un array di Person, legga in modo opportuno il contenuto del file e lo inserisca nell'array (restituzione del numero di Person letti)

```
int readFromBin(char fileName[],
    PersonArray persons);
```

- Creare una funzione **`writeToBin`** che, dato il nome di un file ed un array di Person, scriva in modo opportuno le prime **`count`** strutture dell'array nel file

```
void writeToBin(char fileName[],
    PersonsArray persons, int count);
```

6

## Person & Address - Persistenza

- **`readFromBin`**
  - Aprire il file in sola lettura ed in modalità binaria
  - Leggere dal file specificando la dimensione dell'elemento (Person) e il numero massimo di elementi da leggere (dimensione array)
  - Chiudere il file
- **`writeToBin`**
  - Aprire il file in sola scrittura ed in modalità binaria
  - Scrivere sul file specificando la dimensione dell'elemento (Person) ed il numero effettivo di elementi da scrivere (presenti nell'array)
  - Chiudere il file

7

## Person & Address - Persistenza

```
int readFromBin(char fileName[], PersonArray persons)
{
    int count = 0;
    FILE *f = fopen(fileName, "rb");
    count = fread(persons, sizeof(Person), PERSONARRAYDIM, f);
    fclose(f);
    return count;
}

void writeToBin(char fileName[], PersonArray persons,
   int count)
{
    FILE *f = fopen(fileName, "wb");
    fwrite(persons, sizeof(Person), count, f);
    fclose(f);
}
```


8

## Persistenza Binaria

- Usando le strutture si risolvono molti problemi → lettura in un colpo solo di tutta la struttura o di tutto l'array di strutture
- Senza usare le strutture è altrettanto semplice: l'importante è scrivere (e leggere) dati di tipo e dimensioni prefissate
  - Una stringa di 30+1 caratteri
  - Un **`int`**
  - Un **`double`**
  - ...
- Se la lettura è così facile... Proviamo a complicare un po' le cose → ricerca direttamente su file!

9

## Person & Address – Ricerca su File

- Si faccia riferimento alla ricerca parziale
- L'algoritmo è simile a quello visto in precedenza per la ricerca su array
- Si leggono le strutture una alla volta e si copiano in un array quelle che rispondono alle caratteristiche cercate...
- ...finché il file non termina

10

## Person & Address – Ricerca su File

```
int findPartialByFirstName_File(char firstName[50],
  char fileName[], PersonArray outputPersons)
{
    int outputIndex = 0;
    Person aPerson;
    FILE *f = fopen(fileName, "rb");
    while (fread(&aPerson, sizeof(Person), 1, f) > 0)
    {
        if (startsWith(aPerson.firstName, firstName))
            outputPersons[outputIndex++] = aPerson;
    }
    fclose(f);
    return outputIndex;
}
```

11

## ...è facile sbagliare!

- I file binari sono oggetti a basso livello → non vengono effettuati particolari controlli di compatibilità di tipo
- È possibile, ad esempio, leggere il file binario delle persone ed inserire i dati in strutture diverse da quelle utilizzate per la scrittura
- Ovviamente i dati letti saranno “non utilizzabili”, però...

12

## ...è facile sbagliare!

```
char s[81];
File *f = fopen("persone.bin", "rb");
fread(s, sizeof(char), 80, f);
s[81] = '\0';
printf("Ho letto: %s", s);
```

*Cosa stampa?*


13

## File di testo

- In un file di testo i dati sono memorizzati in un formato non *machine friend*, ma molto *user friend*
  - I dati sono perfettamente leggibili dall'utente
  - Sono da interpretare perché la macchina, così come sono, non li capisce affatto
  - Prima si leggono le stringhe corrispondenti ai diversi campi, poi si trasformano nel tipo di destinazione

*Nome Cognome Voto Lode*

```
Guido La Vespa 30 L
Gustavo L'Olio 24
```

`char[30] char[30] short char`

```
Guido La Vespa 30 L
Gustavo L'Olio 24
```

- Le tecniche più spesso usate per discriminare i vari campi, sono due:
  - Campi separati da separatore (un carattere "speciale")
  - Campi a dimensione fissa

14

## File di testo - Separatore

- Si tratta di decidere un carattere di separazione che non compaia MAI nei dati memorizzati
  - Se così non fosse, la lettura risulterebbe senz'altro errata!
- Nomi e cognomi

Guido La Vespa
Gustavo L'Olio

*Separato da spazio?*

```
Guido La Vespa
Gustavo L'Olio
```

*Separato da apice?*

```
Guido'La Vespa
Gustavo'L'Olio
```

*Separato da virgola?*

```
Guido,La Vespa
Gustavo,L'Olio
```

15

## File di testo – Separatore

- In generale, si può pensare di usare un separatore diverso per ogni campo...
- Si scriva una funzione che dato un file e un carattere di separazione **`sep`**, estragga, partendo dalla posizione corrente nel file, il campo terminato da **`sep`** e lo inserisca in un buffer anch'esso dato
- Il carattere di fine linea funge sempre e comunque da separatore → evitare separatore + fine linea

caratteri letti

```
int readField(char buffer[], char sep, File *f);
```

*E la dimensione del buffer?*

16

## File di testo – `readField`

```
int readField(char buffer[], char sep, FILE *f)
{
    int i = 0;
    char ch = fgetc(f);          LF
    while (ch != sep && ch != 10 && ch != EOF)
    {
        buffer[i++] = ch;
        ch = fgetc(f);
    }
    buffer[i] = '\0';
    return i;
}
```

Legge un carattere per volta e continua ad inserire nel **`buffer`** finché non incontra il separatore o il fine linea.

17

## Person & Address – Scrittura

- La scrittura è piuttosto semplice → si può tranquillamente usare **`fprintf`**!

```
void writeAddressToTxt(Address address, FILE *f)
{
    fprintf(f, "%s;%s;%s;%s\n", address.street,
      address.postalCode, address.city,
  address.state);
}

void writePersonToTxt(Person person, FILE *f)
{
    fprintf(f, "%s;%s;%s;%s\n", person.firstName,
      person.secondName, person.phone, person.cell);
    writeAddressToTxt(person.address, f);
}
```

18

## Person & Address – Lettura

- Con la funzione readField è molto più semplice che dover fare tutto da zero!

```
int readAddressFromTxt(Address *address, FILE *f)
{
    int ok = readField(address->street, ';', f);
    ok = ok && readField(address->postalCode, ';', f);
    ok = ok && readField(address->city, ';', f);
    ok = ok && readField(address->state, ';', f);
    return ok;
}
int readPersonFromTxt(Person *person, FILE *f)
{
    int ok = readField(person->firstName, ';', f);
    ok = ok && readField(person->secondName, ';', f);
    ok = ok && readField(person->phone, ';', f);
    ok = ok && readField(person->cell, ';', f);
    ok = ok && readAddressFromTxt(&(person->address), f);
    return ok;
}
```

19

## Person & Address – Lettura

- Le funzioni della slide precedente sono scritte bene?
  - Sono senza errori, ok…
  - Ma c'è un po' di replicazione specie nel passaggio del separatore (una **`define`**?)
  - Forse sarebbe meglio definire una funzione specifica per la lettura di persone ed indirizzi che si appoggi sulla **`readField`** generica…

```
#define PASEP ';'

int readPAField(char buffer[], FILE *f)
{
    return readField(buffer, PASEP, f);
}
```

20

## Person & Address – Lettura

- L'idea non è quella di risparmiare sui caratteri scritti, ma di razionalizzare il codice → racchiudere in una funzione funzionalità di uso comune → fattorizzare
- Se si vuole cambiare il separatore, si cambia SOLO la **`define`**, se si vuole cambiare la modalità di lettura, si cambia la **`readPAField`** → una volta per tutte!

```
int readAddressFromTxt(Address *address, FILE *f)
{
    int ok = readPAField(address->street, f);
    ok = ok && readPAField(address->postalCode, f);
    …
    return ok;
}
int readPersonFromTxt(Person *person, FILE *f)
{
    int ok = readPAField(person->firstName, f);
    ok = ok && readPAField(person->secondName, f);
    …
    ok = ok && readAddressFromTxt(&(person->address), f);
    return ok;
}
```

21

## Person & Address – Lettura

- Manca solo la funzione di lettura di un array di persone → costruita sui mattoni già disponibili

strutture lette

```
int readFromTxt(char fileName[], PersonArray persons)
{
    int i = 0;
    FILE *f = fopen(fileName, "r");
    if (f != NULL)
    {
        while (readPersonFromTxt(&persons[i++], f));
        fclose(f);
    }
    return i;
}
```

22

## Considerazioni

- Perché non usare la **`fscanf`** in lettura?
  - Se il separatore è lo spazio, tutto ok
  - Se il separatore è un altro carattere (ed è così perché lo spazio fa parte dei dati), ci sono grossi problemi → i separatori di default della **`fscanf`** sono lo spazio, il tab e il nuova linea e, in generale, non possono essere cambiati!!
  - Se i dati non sono di tipo stringa, il problema di cui sopra non si pone!

```
int a, b, c;
char s1[20], s2[20];
FILE *f = fopen("…", "r");
fscanf(f, "%d%d%d", &a, &b, &c);          //sep. di default
fscanf(f, "%d;%d;%d", &a, &b, &c);        //sep. è ';'
fscanf(f, "%s%s", s1, s2);                //sep. di default
fscanf(f, "%s;%s", s1, s2);//specifiche contraddittorie
  //→legge solo s1…
```

23

## Studenti e Voti

- Si definisca una struttura dove possa essere memorizzato:
  - Nome e cognome dello studente
  - Voto conseguito
  - Lode
  - Data di registrazione
- Si prevedano, per tale struttura, adeguate funzioni di lettura e salvataggio su file di testo

24

## Studenti e Voti

```
typedef struct studenteStruct
{
    char nome[50];
    char cognome[50];
    unsigned short int voto;
    boolean lode;
    Date dataRegistrazione;
} Studente;

#define STUDENTEARRAYDIM 100

typedef Studente StudenteArray[STUDENTEARRAYDIM];

int writeStudenteToTxt(Studente *studente, FILE *f);
int readStudenteFromTxt(Studente *studente, FILE *f);
int readStudentiFromTxt(char fileName[],
   StudenteArray studenti);
int writeStudentiToTxt(char fileName[],
   StudenteArray studenti, int count);
```

*Chi è* **`Date`***?*

25

## Date

```
typedef struct dateStruct
{
    unsigned short int day, month, year;
} Date;

int readDateFromTxt(Date *d, FILE *f);
int writeDateToTxt(Date *d, FILE *f);

//Altre funzioni "comode"
int readDate(Date *d);
void formatDate(char *buffer, Date *d);
void printDate(Date *d);
```

26

## Date

Il formato della data deve essere: **Giorno/Mese/Anno**

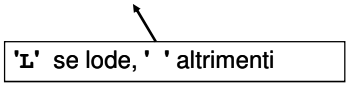
```
int readDateFromTxt(Date *d, FILE *f)
{
    return fscanf(f, "%hu/%hu/%hu", &d->day, &d->month,
       &d->year) == 3;
}

int writeDateToTxt(Date *d, FILE *f)
{
    return fprintf(f, "%hu/%hu/%hu", d->day, d->month,
       d->year);
}

int readDate(Date *d)
{
    return readDateFromTxt(d, stdin);
}
```

27

## Studenti e Voti – Scrittura

Il formato della linea deve essere:

```
Cognome;Nome;VotoLode Giorno/Mese/Anno
```




```
int writeStudenteToTxt(Studente *studente, FILE *f)
{
    int ok = fprintf(f, "%s;%s;%hu%c ", studente->cognome,
       studente->nome, studente->voto,
       studente->lode ? 'L' : ' ');
    ok = ok && writeDateToTxt(&studente->dataRegistrazione, f);
    fprintf(f, "\n"); //Fine linea
    return ok;
}
```

28

## Studenti e Voti – Scrittura

```
int writeStudentiToTxt(char fileName[],
   StudenteArray studenti, int count)
{
     int i = 0;
     FILE *f;
     f = fopen(fileName, "w");
     if (f != NULL)
     {
          while (i < count)
              writeStudenteToTxt(&studenti[i++], f);
          fclose(f);
     }
     return i;
}
```

29

## Studenti e Voti – Lettura

Porre attenzione a:

- Lettura stringhe
- Lettura booleano
- Controllo d'errore
- Lettura fine linea

Dice il saggio: *"ad ogni codifica corrisponde una opportuna corrispondente decodifica!"*

```
#define SSEP ';'
int readStudenteFromTxt(Studente *studente, FILE *f)
{
    char lode;
    int ok = readField(studente->cognome, SSEP, f);
    ok = ok && readField(studente->nome, SSEP, f);
    ok = ok && fscanf(f, "%hu", &studente->voto) == 1;
    ok = ok && fscanf(f, "%c ", &lode) == 1;
    studente->lode = lode == 'L';
    ok = ok && readDateFromTxt(&studente->dataRegistrazione, f);
    fscanf(f, "\n");
    return ok;
}
```

30

## Studenti e Voti – Lettura

```
int readStudentiFromTxt(char fileName[],
  StudenteArray studenti)
{
    int i = 0;
    FILE *f = fopen(fileName, "r");
    if (f != NULL)
    {
        while (readStudenteFromTxt(&studenti[i], f))
            i++;
        fclose(f);
    }
    return i;
}
```

31

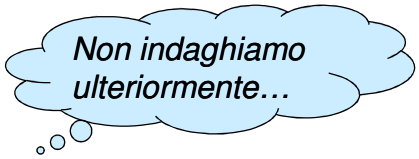
## Considerazioni

- Notare che, per come sono state organizzate le cose, le funzioni di lettura e scrittura di array di strutture sono tutte molto simili tra loro
- In C ci sarebbero alcuni artifizi che consentirebbero di rendere totalmente generiche tali funzioni
- Nei linguaggi ad oggetti moderni (C++, Java, C#,…) ci sono meccanismi ben codificati che consentono di specializzare gli *oggetti* in modo da riuscire a fattorizzare ulteriormente il codice…
- *…nel corso di Fondamenti di Informatica L-B*

32

## ...saltare qua e là…

- Per la cronaca, è possibile accedere ai file in modo diretto, spostando a piacimento la testina di lettura/scrittura tramite le primitive:
  - `fseek`
  - `fgetpos`
  - `fsetpos`
  - `ftell`
  - `frewind`




- Se file di testo, l'unità di spostamento è il singolo carattere
- Se file binario, l'unità di spostamento è il singolo byte
- Bisogna anche fare attenzione a saltare "nel posto giusto"
  - Se file binario con strutture memorizzate, saltare all'inizio di una delle strutture → occorre fare alcuni calcoli…
  - Se file di testo, saltare all'inizio di un campo o di una riga

33

## Altre Considerazioni

- Supponiamo che il file da leggere non sia prodotto da un nostro programma ma che provenga dal "mondo esterno"
- Supponiamo che il file contenga più informazioni di quante il nostro array staticamente dimensionato possa contenere
- Che si fa?
  - Si aumenta la dimensione dell'array (ricompilazione)
  - Si cambia mestiere…
  - Oppure, meglio, ci si affida a strutture dati dinamiche → allocazione dinamica della memoria!!!

34