

# File – qualche nota riassuntiva

---

- Che 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, tipicamente ad accesso sequenziale**
- Cosa occorre fare per operare correttamente su un file?
  - Conoscere il nome assoluto/relativo
  - 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 il concetto di posizione corrente
  - Leggere e scrivere (e comportarsi) in modo diverso a seconda che il file sia di testo (**fscanf, fprintf, fgetc, fputc,...**) o binario (**fread, fwrite**)
  - Ricordare **assolutamente** di **chiudere** il file al termine delle operazioni  
→ un file aperto è una risorsa in utilizzo da parte di quel programma in esecuzione

# File binari

---

- In un file binario i dati sono memorizzati in formato “*machine-friendly*”
- 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

# 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`

# 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(...)` sul tipo di dato
- 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 (`&`)

# Person & Address – Definizioni

---

```
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];
```

# 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 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);
```

# Person & Address - Persistenza

---

## ■ **readFromBin ()**

- Aprire il file in sola lettura e 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 e 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

# 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);
}
```

**Facile!**

# 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!

# Person & Address – Ricerca su File

---

Si faccia riferimento alla ricerca parziale

- Si leggono le strutture una alla volta e si copiano in un array solo quelle che rispondono alle caratteristiche cercate...
- ...finché il file non termina
- Nessuna necessità di mantenere un array di appoggio con copia dei dati

# 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 (!strcmp(aPerson.firstName, firstName)) {
            outputPersons[outputIndex] = aPerson;
            outputIndex++;
        }
    }
    fclose(f);
    return outputIndex;
}
```

## ...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 e inserire i dati in strutture diverse da quelle utilizzate per la scrittura
- Ovviamente i dati letti saranno “*non utilizzabili*”, però...

# ...è facile sbagliare!

---

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

*Che cosa stampa?*

**BOH?!**

# File di testo

---

- In un file di testo i dati sono memorizzati in formato maggiormente “*user-friendly*”
  - 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

<i>Nome</i>	<i>Cognome</i>	<i>Voto</i>	<i>Lode</i>	<i>char[30]</i>	<i>char[30]</i>	<i>short</i>	<i>char</i>
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

# 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

# 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?*

# File di testo – readField

---

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

# 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);
}
```