

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, fgets, fputs, 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

1

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

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`?

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(...)` 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 (&)

4

Person & Address – Definizioni

```
#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];
```

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 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 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

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);
}
```

Facile!

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

- 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

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 (!strcmp(aPerson.firstName, firstName)) {
            outputPersons[outputIndex] = aPerson;
            outputIndex++;
        }
    }
    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 e 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);
```

Che cosa stampa?

BOH?!

13

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

Nome	Cognome	Voto	Lode	char[30]	char[30]	short	char		
Guido	La	Vespa	30	L	Guido	La	Vespa	30	L
Gustavo	L'Olio	24		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);
    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

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

PUNTATORI A STRUTTURE

È possibile utilizzare i puntatori per accedere a variabili di tipo struct

Ad esempio:

```
typedef struct { int Campo_1, Campo_2;
} TipoDato;

TipoDato S, *P;
P = &S;
```

Operatore . di selezione campo ha precedenza su *

=> necessarie parentesi tonde `(*P).Campo_1=75;`

Operatore -> consente l'accesso ad un campo di una struttura

referenziata da puntatore in modo più sintetico: `P->Campo_1=75;`

19

Person & Address – Lettura

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

```
boolean readAddressFromTxt(Address *address, FILE *f)
{
    boolean ok = readField(address->street, ';', f) > 0;
    ok = ok && readField(address->postalCode, ';', f) > 0;
    ok = ok && readField(address->city, ';', f) > 0;
    ok = ok && readField(address->state, ';', f) > 0;
    return ok;
}

boolean readPersonFromTxt(Person *person, FILE *f)
{
    boolean ok = readField(person->firstName, ';', f) > 0;
    ok = ok && readField(person->secondName, ';', f) > 0;
    ok = ok && readField(person->phone, ';', f) > 0;
    ok = ok && readField(person->cell, ';', f) > 0;
    ok = ok && readAddressFromTxt(&(person->address), f);
    return ok;
}
```

20

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);
}
```

21

Person & Address – Lettura

- L'idea principale è di razionalizzare il codice → racchiudere in un solo modulo 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!

```
boolean readAddressFromTxt(Address *address, FILE *f)
{
    boolean ok = readPAField(address->street, f) > 0;
    ok = ok && readPAField(address->postalCode, f) > 0;
    ...
    return ok;
}

boolean readPersonFromTxt(Person *person, FILE *f)
{
    boolean ok = readPAField(person->firstName, f) > 0;
    ok = ok && readPAField(person->secondName, f) > 0;
    ...
    ok = ok && readAddressFromTxt(&(person->address), f) > 0;
    return ok;
}
```

22

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;
}
```

23

Considerazioni

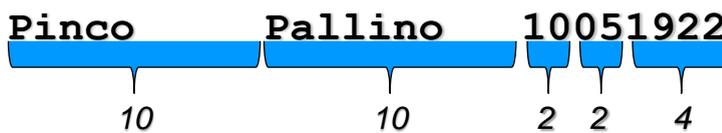
- Perché non usare la `fscanf()` in lettura?
 - Se il separatore è lo spazio, tutto ok
 - Se il separatore è un altro carattere (è così perché lo spazio fa parte dei dati), ci sono grossi problemi → i separatori di default della `fscanf()` sono lo spazio, il tab e il nuova linea e, in generale, non possono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

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

24

Lettura di campi a dim. fissa

- Si usa `fscanf()` per “mangiare” il numero di caratteri corretto
- Si usa `sscanf()` per convertire (eventualmente) in tipi diversi dalla stringa



25

Lettura di campi a dim. fissa

```
#define NOME_LEN 21
#define PERSONA_ARRAY_DIM 20
typedef struct
{
    int giorno;
    int mese;
    int anno;
} Data;

typedef struct
{
    char nome[NOME_LEN];
    char cognome[NOME_LEN];
    Data nascita;
} Persona;
```

26

Lettura di campi a dim. fissa

```
void readPersona(FILE *f, Persona *pers)
{
    char giorno[3], mese[3], anno[5];
    fscanf(f, "%10c%10c", pers->cognome, pers->nome);
    pers->cognome[10] = '\0';
    pers->nome[10] = '\0';

    fscanf(f, "%2c%2c%4c", giorno, mese, anno);
    giorno[2] = '\0';
    mese[2] = '\0';
    anno[4] = '\0';

    sscanf(giorno, "%d", &pers->nascita.giorno);
    sscanf(mese, "%d", &pers->nascita.mese);
    sscanf(anno, "%d", &pers->nascita.anno);
}
```

Servono i terminatori

Servono i terminatori

27

Qualche considerazione a latere

- Per la cronaca, è possibile accedere ai file in modo diretto, spostando a piacimento la testina di lettura/scrittura tramite le primitive:
 - `fseek()`
 - ma anche `fgetpos()`, `fsetpos()`, `ftell()`, `frewind()`
- Supponiamo che il file da leggere non sia prodotto da un nostro programma e che contenga più informazioni di quante il nostro **array staticamente dimensionato** possa contenere
- Che si fa?
 - Si aumenta la dimensione dell'array (ricompilazione)
 - Oppure, meglio, ci si affida a strutture dati dinamiche → **allocazione dinamica della memoria**

28