

Basso Livello, File Binari e l'Abisso della Ricorsione



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

Basso Livello, File Binari e l'Abisso della Ricorsione

Nelle lezioni precedenti abbiamo visto come salvare informazioni in file testuali leggibili. Tuttavia, la formattazione in testo ASCII introduce conversioni costose e rende difficile saltare direttamente a un dato specifico senza leggere sequenzialmente tutto ciò che lo precede.

In questa lezione esploreremo la vera forma dei dati in memoria fisica: impareremo a “fotocopiare” i byte dalla RAM al disco rigido creando **File Binari** tramite `fwrite` e `fread`, ad ispezionare il loro contenuto a basso livello svelando le convenzioni sull'ordinamento dei byte (Little Endian) e a navigare liberamente nello spazio assoluto del file tramite `fseek` e `ftell`. Nella seconda parte della lezione approfondiremo la struttura della **Ricorsione**, analizzandone l'eleganza algoritmica, l'impatto sul Record di Attivazione nello Stack, il rischio hardware di Stack Overflow e le tecniche di ottimizzazione tramite la ricorsione di coda (Tail Recursion).

1.1 I File Binari e l'Ispezione a Basso Livello

1.1.1 Testo contro Binario: L'Illusione ASCII

Quando salviamo un numero intero come 1000000 (un milione) usando `fprintf`, la funzione converte il numero in una stringa di sette caratteri ASCII ("1", "0", "0", "0", "0", "0", "0"). Sul disco rigido, questo numero occuperà esattamente **7 byte** (uno per carattere). Se scrivessimo il valore 1000000000 (un miliardo), esso occuperebbe **10 byte**.

In memoria RAM, tuttavia, un tipo intero (`int`) occupa rigorosamente **4 byte** (32 bit) di spazio a prescindere dal valore numerico immagazzinato. Il formato testuale costringe la CPU a compiere continui cicli di conversione in potenze di dieci per tradurre i bit interni in cifre leggibili.

Per ovviare a questo spreco di tempo di calcolo e spazio, possiamo memorizzare i dati su disco nella loro rappresentazione binaria nativa. Un **file binario** è una copia esatta byte per byte (un *dump*) di una determinata area di memoria RAM trasferita direttamente su disco senza alcuna traduzione o formattazione.

1.1.2 La Scrittura Binaria con `fwrite`

Per aprire un file in modalità binaria si aggiunge il carattere `b` alla modalità di apertura in `fopen` (ad es. `"wb"` per la scrittura e `"rb"` per la lettura).

La funzione standard `fwrite` consente di riversare porzioni di memoria RAM sul file stream e accetta quattro argomenti:

1. L'indirizzo di memoria di partenza da cui leggere i dati.
2. La dimensione in byte di un singolo elemento (ad es. `sizeof(int)`).
3. Il numero di elementi consecutivi da scrivere.
4. Il puntatore allo stream del file (`FILE *`).

Codice C: Confronto tra scrittura testuale e binaria

```
1  #include <stdio.h>
2
3  int main(void) {
4      int numero = 1000000; // 1 milione
5
6      // 1. Scrittura Testuale (fprintf)
7      FILE *fp_txt = fopen("numero.txt", "w");
8      if (fp_txt != NULL) {
9          fprintf(fp_txt, "%d", numero);
10         fclose(fp_txt);
11     }
12
13     // 2. Scrittura Binaria (fwrite)
14     FILE *fp_bin = fopen("numero.bin", "wb");
15     if (fp_bin != NULL) {
16         fwrite(&numero, sizeof(int), 1, fp_bin);
17         fclose(fp_bin);
18     }
19
20     printf("File generati correttamente.\n");
21     return 0;
22 }
```

Verificando le proprietà fisiche dei file sul disco, si noterà che `numero.txt` ha una dimensione pari a 7 byte, mentre `numero.bin` occupa solo 4 byte, contenendo la rappresentazione binaria del numero.

1.1.3 Hexdump e la Rappresentazione Little Endian

Se proviamo ad aprire il file `numero.bin` con un comune editor di testo, visualizzeremo caratteri incomprensibili (come `@B`). Questo accade perché l'editor tenta erroneamente di interpretare i byte numerici come caratteri ASCII stampabili.

Per ispezionare correttamente il contenuto a basso livello di un file binario, si utilizzano utility di terminale come `hexdump`:

```
1 $ hexdump -C numero.bin
```

L'output esadecimale mostrerà la seguente configurazione:

```
1 00000000 40 42 0f 00 |@B..|
```

Matematicamente, il numero 1.000.000 in base esadecimale corrisponde a 0x000F4240. Nel file binario, tuttavia, l'ordine dei byte risulta invertito (40 42 0F 00).

Questo fenomeno deriva dall'architettura dei processori moderni (in particolare la famiglia x86-64 di Intel e AMD), la quale adotta la convenzione **Little Endian**: i byte che costituiscono una parola multi-byte vengono memorizzati in RAM partendo dal byte meno significativo (a destra) fino a quello più significativo (a sinistra). La funzione `fwrite` si limita a copiare fedelmente l'esatta sequenza di byte presente in RAM sul disco rigido.

1.2 Accesso Casuale e Gestione dei Record Binari

1.2.1 Salvataggio di Array in Blocco

Poiché in C gli elementi di un array sono memorizzati in posizioni di memoria contigue, è possibile scrivere l'intero array su disco con un'unica invocazione di `fwrite`, evitando i cicli iterativi:

```
1 int dati[5] = {10, 20, 30, 40, 50};
2 // Copia diretta di (5 * sizeof(int)) = 20 byte consecutivi
3 fwrite(dati, sizeof(int), 5, fp_bin);
```

Specularmente, per leggere i dati salvati si utilizza `fread` su un file aperto in modalità "rb":

```
1 int letti[5];
2 fread(letti, sizeof(int), 5, fp_bin);
```

1.2.2 Navigazione nel File con `fseek` e `ftell`

La lettura sequenziale ci costringe a leggere il file dall'inizio alla fine (ad esempio, per leggere il centesimo record di un file di testo, siamo costretti a leggere a vuoto le prime 99 righe).

Nei file binari, grazie alla dimensione costante e nota dei dati, possiamo posizionare il puntatore di lettura in qualsiasi punto dello stream istantaneamente tramite la funzione `fseek`:

```
1 int fseek(FILE *stream, long offset, int origin);
```

Il parametro `origin` può assumere tre costanti standard:

- `SEEK\{\}_SET`: Spostamento relativo all'inizio del file (offset positivo).
- `SEEK\{\}_CUR`: Spostamento relativo alla posizione corrente del cursore.

- `SEEK\{\}_END`: Spostamento relativo alla fine del file (generalmente con offset negativo).

Per conoscere in quale byte esatto del file si trova attualmente il cursore di lettura, si utilizza la funzione `ftell`:

```
1 long posizione = ftell(fp);
```

Codice C: Lettura diretta di un elemento specifico

```
1 FILE *fp = fopen("array.bin", "rb");
2 if (fp != NULL) {
3     // Saltiamo i primi due interi (2 * 4 = 8 byte) a partire dall'inizio
4     // ↳ del file
5     fseek(fp, 2 * sizeof(int), SEEK_SET);
6
7     int terzo_numero;
8     fread(&terzo_numero, sizeof(int), 1, fp);
9     printf("Terzo elemento dell'array letto direttamente: %d\n",
10    // ↳ terzo_numero); // Stampa 30
11     fclose(fp);
12 }
```

1.2.3 Record a Dimensione Fissa e Stringhe Variabili

Se decidiamo di salvare record di testo a lunghezza variabile preceduti dalla loro dimensione (ad esempio memorizzando prima un intero con la lunghezza e poi i caratteri), perdiamo la possibilità di utilizzare `fseek` ad offset costante:

Tabella

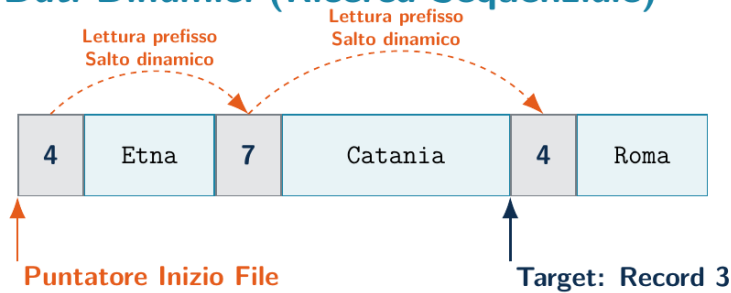
Record			Record			Record		
1	4	"Etna"	2	7	"Catania"	3	4	"Roma"

Per raggiungere il terzo record, non conoscendo a priori la lunghezza dei primi due, saremmo costretti a una scansione sequenziale passiva dei prefissi.

Per poter sfruttare l'efficienza dell'accesso diretto, è necessario adottare **record a dimensione fissa** (ad esempio riservando sempre 50 byte per il campo nome: `char nome[50]`). Imponendo dimensioni costanti a ciascun blocco di dati, la posizione iniziale del record d'indice i sarà calcolabile matematicamente con la formula:

$$\text{Offset} = i \times \text{Dimensione Record}$$

Dati Dinamici (Ricerca Sequenziale)



Dati a Blocco Fisso (Accesso Diretto con fseek)

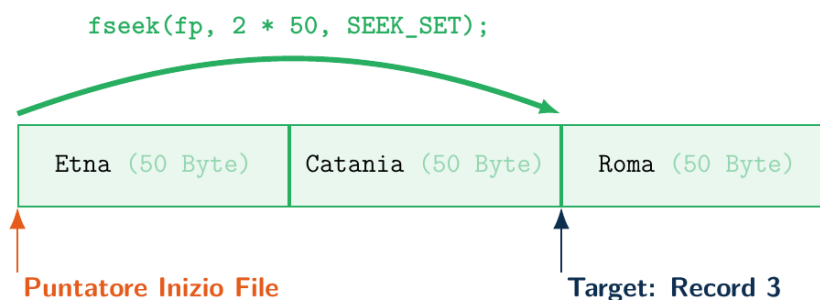


Figura 1.1: Navigazione a Blocchi vs Navigazione Sequenziale

1.3 L'Abisso della Ricorsione e i Limiti dello Stack

1.3.1 I Principi della Ricorsione

La ricorsione è un metodo di programmazione in cui una funzione risolve un problema richiamando se stessa su sotto-problemi più piccoli dello stesso tipo.

Per prevenire l'esecuzione indefinita delle chiamate, ogni funzione ricorsiva deve implementare due parti essenziali:

1. **Il Caso Base (o condizione di arresto):** La situazione più semplice del problema la cui soluzione è banale e immediata. Rappresenta l'ancora di salvataggio del programma.
2. **Il Passo Ricorsivo:** La chiamata ricorsiva alla funzione stessa. Per garantire che l'esecuzione termini, i parametri passati alla chiamata successiva devono convergere progressivamente verso il caso base.

1.3.2 Esempi Algoritmici Classici

Il Calcolo del Fattoriale ($N!$)

- **Caso Base:** Il fattoriale di 0 o di 1 vale 1.
- **Passo Ricorsivo:** Il fattoriale di N è pari a $N \times (N - 1)!$.

```
1 long long fattoriale(int n) {
2     // Caso Base
3     if (n == 0 || n == 1) {
4         return 1;
5     }
6     // Passo Ricorsivo
7     return n * fattoriale(n - 1);
8 }
```

Stampa Inversa di una Stringa

Sfruttando la memoria dello Stack, possiamo stampare a schermo una stringa al contrario senza dover manipolare gli indici in modo iterativo:

```
1 void stampa_inverso(char *str) {
2     // Caso Base: Carattere terminatore raggiunto
3     if (*str == '\0') {
4         return;
5     }
6     // Passo Ricorsivo: Avvia la chiamata per i caratteri successivi
7     stampa_inverso(str + 1);
8
9     // Al ritorno dallo srotolamento delle funzioni, stampa il carattere
10    ↪ corrente
11    putchar(*str);
12 }
```

La Sequenza di Fibonacci

La definizione ricorsiva della sequenza di Fibonacci prevede:

- **Casi Base:** Se $N = 0$ ritorna 0; se $N = 1$ ritorna 1.
- **Passo Ricorsivo:** Ritorna la somma dei due numeri precedenti: $\text{Fibonacci}(N - 1) + \text{Fibonacci}(N - 2)$.

```
1 int fibonacci(int n) {
2     // Casi Base
3     if (n == 0) return 0;
4     if (n == 1) return 1;
5
6     // Passo Ricorsivo (Doppia chiamata)
7     return fibonacci(n - 1) + fibonacci(n - 2);
8 }
```

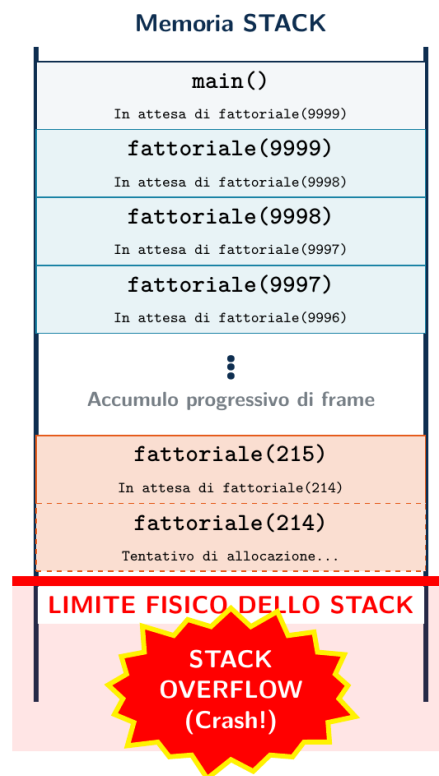


Figura 1.2: Lo Stack Overflow Ricorsivo

1.3.3 Lo Stack Frame e la Catastrofe dello Stack Overflow

Ogni volta che viene invocata una funzione, il sistema operativo riserva una porzione di memoria RAM in cima all'area dello Stack, denominata **Record di Attivazione (Stack Frame)**. Questo record contiene:

- Le variabili locali e i parametri della funzione.
- L'indirizzo di ritorno (l'istruzione del codice a cui tornare quando la funzione termina).

Nelle funzioni ricorsive, i vari Stack Frame si accumulano uno sopra l'altro nello Stack, rimanendo "sospesi" in attesa che l'ultima chiamata termini raggiungendo il caso base.

Se commettiamo un errore di logica escludendo il caso base, o se impostiamo una ricorsione estremamente profonda, lo Stack (la cui dimensione tipica allocata per il processo è di circa 8 MB) esaurisce il suo spazio fisico. Questo evento, denominato **Stack Overflow**, provoca la terminazione immediata del programma da parte del sistema operativo tramite un segnale di errore di segmentazione (*Segmentation Fault*).

1.3.4 Inefficienza e la Ricorsione di Coda (Tail Recursion)

Prendiamo in esame l'implementazione classica e intuitiva della Sequenza di Fibonacci:

```
1 int fibonacci(int n) {
2     if (n == 0) return 0;
```

```
3     if (n == 1) return 1;
4     return fibonacci(n - 1) + fibonacci(n - 2);
5 }
```

L'esecuzione di `fibonacci(45)` provocherà un visibile blocco della CPU. Questo accade perché lo sdoppiamento della chiamata ricorsiva crea un albero delle chiamate che cresce in modo esponenziale ($O(2^N)$), costringendo il processore a calcolare ripetutamente lo stesso sotto-problema migliaia di volte su rami differenti dell'albero.

La Ricorsione di Coda (Tail Recursion)

Una funzione ricorsiva si dice **ricorsiva di coda** (o *tail recursive*) se la chiamata ricorsiva rappresenta l'ultimissima istruzione eseguita, non lasciando alcuna computazione in sospeso al ritorno del controllo.

Nei compilatori moderni, l'abilitazione delle ottimizzazioni (ad es. `gcc -O3`) attiva la **Tail Call Optimization (TCO)**. Il compilatore rileva la ricorsione di coda e riutilizza lo stesso Stack Frame per tutta la catena di chiamate, azzerando l'overhead spaziale sullo Stack e riducendolo a $O(1)$.

Possiamo ottimizzare la funzione di Fibonacci introducendo variabili accumulatrici per renderla tail-recursive, riducendo la complessità a $O(N)$ temporale:

```
1 // Funzione ausiliaria ottimizzata per la ricorsione di coda
2 long long fib_tail(int n, long long a, long long b) {
3     if (n == 0) return a;
4     if (n == 1) return b;
5     return fib_tail(n - 1, b, a + b); // Nessuna operazione rimasta dopo
    ↪ la chiamata
6 }
7
8 long long fibonacci_efficiente(int n) {
9     return fib_tail(n, 0, 1);
10 }
```

Attenzione

Attenzione In C, sebbene il compilatore cerchi di ottimizzare le chiamate ricorsive di coda, la ricorsione espone a rischi strutturali legati ai limiti fisici dello Stack. Per elaborazioni sequenziali lineari (come Fibonacci o Fattoriale), l'uso dei classici cicli iterativi (`for`, `while`) rimane la scelta ottimale, garantendo un'occupazione di memoria costante $O(1)$ ed eliminando alla radice il rischio di Stack Overflow.

1.4 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Quale delle seguenti affermazioni esprime la differenza fondamentale tra scrittura con `fprintf` e `fwrite`?

- A) `fprintf` scrive in formato binario, mentre `fwrite` scrive in ASCII.
- B) `fprintf` traduce i dati in stringhe ASCII leggibili, mentre `fwrite` esegue una copia grezza byte per byte della memoria RAM.
- C) `fwrite` inserisce automaticamente caratteri di nuova riga `\{\}\{\}\n` alla fine di ogni dato.
- D) `fwrite` può operare solo su variabili di tipo `char`.

Soluzione
Risposta corretta: B
Spiegazione: `fprintf` effettua una conversione formattata traducendo i valori numerici in caratteri leggibili (testo), consumando tempo di calcolo e spazio variabile a seconda del numero. `fwrite` riversa direttamente i byte così come appaiono nella RAM fisica, in modo estremamente efficiente e a dimensione fissa.

Quiz 2

Domanda

2. Ispezionando un file binario contenente l'intero 1 su architettura Intel x86-64, l'hexdump mostra la sequenza di byte `01 00 00 00`. Perché?

- A) Si tratta di un errore di compilazione.
- B) L'architettura adotta la convenzione Big Endian, memorizzando prima il byte più significativo.
- C) L'architettura adotta la convenzione Little Endian, memorizzando prima il byte meno significativo.
- D) Il file inserisce byte di controllo casuali per allineare la memoria.

Soluzione
Risposta corretta: C
Spiegazione: Le architetture x86-64 sono di tipo Little Endian. In un intero a 4 byte (32 bit) avente valore 1 (`0x00000001`), il byte meno significativo (`01`) viene salvato all'indirizzo di memoria più basso, e quindi appare per primo nel dump sequenziale del file.

Quiz 3

Domanda

3. Cosa restituisce la chiamata `fwrite(array, sizeof(double), 10, fp)` in caso di scrittura completata con successo?

- A) Il valore intero 0.
- B) Il numero totale di byte scritti (ovvero 80).
- C) Il numero di elementi scritti con successo (ovvero 10).
- D) Il puntatore al file modificato.

Soluzione
Risposta corretta: C
Spiegazione: Sia `fwrite` che `fread` restituiscono il numero di blocchi/elementi scritti o letti correttamente (il terzo parametro passato alla funzione), e non il numero totale di byte.

Quiz 4

Domanda

4. Quale parametro della funzione `fseek` indica lo spostamento a partire dalla fine del file?

- A) `SEEK\{}_SET`
- B) `SEEK\{}_CUR`
- C) `SEEK\{}_END`
- D) `SEEK\{}_EOF`

Soluzione
Risposta corretta: C
Spiegazione: `SEEK\{}_END` indica al cursore di spostarsi a partire dalla fine del file (generalmente usando offset negativi per risalire il file all'indietro).

Quiz 5

Domanda

5. Cosa accade se si tenta di eseguire un `fseek` con un offset calcolato in modo fisso su un file binario con stringhe a lunghezza variabile senza prefisso fisso?

- A) Il programma crasha immediatamente.
- B) Si rischia di posizionare il cursore a metà di un dato, leggendo byte spuri che corrompono le variabili.

- C) fseek corregge automaticamente la posizione allineandosi all'inizio della parola.
- D) Il file binario si converte automaticamente in file di testo.

Soluzione
Risposta corretta: B
Spiegazione: Nei file binari non ci sono delimitatori naturali come gli spazi o i ritorni a capo. Se saltiamo ad un offset arbitrario non allineato con l'inizio di una variabile, i byte estratti verranno interpretati in modo completamente errato.

Quiz 6

Domanda

6. Cos'è uno Stack Frame (o Record di Attivazione)?

- A) Un blocco di memoria Heap usato per le stringhe globali.
- B) Una porzione di memoria dello Stack allocata ad ogni chiamata di funzione per conservare variabili locali, parametri e indirizzo di ritorno.
- C) Una modalità di compilazione sicura.
- D) Un descrittore per la navigazione dei file binari.

Soluzione
Risposta corretta: B
Spiegazione: Lo Stack Frame contiene tutte le informazioni contestuali a una singola chiamata di funzione. Quando una funzione chiama se stessa ricorsivamente, un nuovo stack frame viene inserito in cima allo Stack, restando in attesa che la chiamata modificata si risolva.

Quiz 7

Domanda

7. Quale grave malfunzionamento hardware/software si verifica se una funzione ricorsiva non raggiunge mai il caso base?

- A) Memory Leak irreversibile nello Heap.
- B) Stack Overflow, causato dall'esaurimento della memoria Stack destinata ai record di attivazione del processo.
- C) Divisione per zero spontanea a livello di ALU.
- D) Corruzione fisica del file binario sul disco rigido.

Soluzione
Risposta corretta: B
Spiegazione: L'accumulo infinito di record di attivazione senza alcun caso base che svuoti lo stack porta rapidamente al superamento del limite fisico dello Stack, provocando la terminazione forzata del programma da parte dell'OS.

Quiz 8

Domanda

8. Nella sequenza di Fibonacci calcolata ricorsivamente (`fibonacci(n-1) + fibonacci(n-2)`), perché le prestazioni sono pessime?

- A) Perché il tipo `int` va continuamente in overflow.
- B) Perché la RAM del sistema operativo si satura istantaneamente.
- C) Perché l'albero delle chiamate ricorsive ricalcola ripetutamente le stesse sotto-funzioni su rami separati, portando a una complessità temporale esponenziale $O(2^N)$.
- D) Perché `scanf` rallenta il processore.

Soluzione
Risposta corretta: C
Spiegazione: L'inefficienza è di natura algoritmica: la stessa sotto-chiamata (ad esempio `fibonacci(5)`) viene calcolata in modo ridondante migliaia di volte a causa dell'albero di sdoppiamento ricorsivo.

Quiz 9

Domanda

9. Cos'è la Tail Recursion (Ricorsione di Coda)?

- A) Una ricorsione eseguita solo alla fine del programma `main()`.
- B) Una tecnica per posizionare il cursore dei file alla fine dello stream.
- C) Una ricorsione in cui la chiamata ricorsiva rappresenta l'ultima istruzione in assoluto della funzione, permettendo l'ottimizzazione e il riciclo dello stack frame corrente.
- D) Una ricorsione applicabile solo su array di tipo `double`.

Soluzione**Risposta corretta: C**

Spiegazione: Essendo la chiamata ricorsiva l'ultimo passo, il compilatore sa che non vi sono operazioni residue da compiere dopo il ritorno. Di conseguenza, riutilizza lo stesso frame di attivazione senza sprecare ulteriore memoria sullo stack (Tail Call Optimization).

Quiz 10**Domanda**

10. Quale complessità spaziale in termini di occupazione dello Stack possiede un algoritmo iterativo (for o while) rispetto a un algoritmo ricorsivo non ottimizzato?

- A) L'algoritmo iterativo occupa $O(N)$ memoria dello Stack.
- B) L'algoritmo iterativo occupa $O(1)$ memoria dello Stack, mentre quello ricorsivo non ottimizzato richiede $O(N)$ frame di attivazione.
- C) Entrambi hanno complessità spaziale esponenziale $O(2^N)$.
- D) L'algoritmo iterativo non usa memoria RAM.

Soluzione**Risposta corretta: B**

Spiegazione: I cicli iterativi mantengono le stesse variabili all'interno dello stesso frame di attivazione (spazio costante $O(1)$). La ricorsione crea invece un frame per ciascun livello di ricorsione, portando a un'occupazione di memoria lineare $O(N)$ rispetto alla profondità.

1.5 Esercizi Pratici

Esercizio 1: Scrittura e Lettura Binaria di Array**Domanda**

(bin\{}_array\{}_io.c)

Si scriva un programma C che inizializzi un array di 10 numeri decimali double. Il programma deve salvare l'intero array in un file binario chiamato `valori.bin` con una singola chiamata a `fwrite`. Successivamente, riaprire lo stesso file in lettura binaria, caricare i dati in un secondo array usando `fread` e stamparli a schermo verificando la corretta corrispondenza dei dati.

Soluzione

Spiegazione dell'algoritmo: Invece di iterare elemento per elemento, la funzione `fwrite` riceve come parametro l'indirizzo base dell'array originale ed esegue una copia esatta contigua di $10 \times 8 = 80$ byte. Il medesimo blocco viene poi riletto linearmente tramite `fread`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define N 10
5
6  int main(void) {
7      double originali[N] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8,
8          ↪ 9.9, 10.0};
9      double letti[N];
10     // 1. Scrittura
11     FILE *out = fopen("valori.bin", "wb");
12     if (out == NULL) {
13         fprintf(stderr, "Errore in scrittura\n");
14         return 1;
15     }
16     fwrite(originali, sizeof(double), N, out);
17     fclose(out);
18
19     // 2. Lettura
20     FILE *in = fopen("valori.bin", "rb");
21     if (in == NULL) {
22         fprintf(stderr, "Errore in lettura\n");
23         return 1;
24     }
25     size_t n_letti = fread(letti, sizeof(double), N, in);
26     fclose(in);
27
28     if (n_letti != N) {
29         ↪ fprintf(stderr, "Errore di lettura: letti solo %zu elementi\n", n_letti);
30         return 1;
31     }
32     printf("Dati letti con successo:\n");
33     for (int i = 0; i < N; i++) {
34         printf("    Elemento %d: %.1f\n", i, letti[i]);
35     }
36     }
37     return 0;
38 }

```

Esercizio 2: Calcolo della Dimensione di un File**Domanda**

(file\{}_size.c)

Si scriva un programma C che accetti da riga di comando il nome di un file (di qualsiasi tipo, testo o binario). Utilizzando esclusivamente le funzioni `fseek` e `ftell`, il programma deve determinare la dimensione esatta del file in byte e visualizzarla a schermo.

Spiegazione dell'algoritmo: Spostando il cursore interno alla fine assoluta del file tramite `fseek(fp, 0, SEEK_END)`, interroghiamo lo stream tramite `ftell(fp)`. Il valore intero restituito indica quanti byte precedono il cursore attuale, corrispondendo quindi alla grandezza fisica in byte del file.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          fprintf(stderr, "Uso: %s [nome_file]\n", argv[0]);
7          return 1;
8      }
9
10     // Apriamo in binario per evitare conversioni di newline che
11     // alterano i conteggi byte
12     FILE *fp = fopen(argv[1], "rb");
13     if (fp == NULL) {
14         fprintf(stderr, "Errore: impossibile aprire il file '%s'\n",
15             argv[1]);
16         return 1;
17     }
18     // Saltiamo alla fine del file
19     fseek(fp, 0, SEEK_END);
20     // ftell ci restituisce la posizione corrente, ovvero la
21     // dimensione del file
22     long dimensione = ftell(fp);
23     fclose(fp);
24     printf("Il file '%s' ha una dimensione di %ld byte.\n", argv
25         return 0;
26     }
```

Soluzione**Esercizio 3: Lettore di Record Binario Fissato****Domanda**

(record\{}_reader.c)

Si scriva un programma C che simuli un archivio di record di dimensioni fisse. Il record è composto esattamente da 50 byte contenenti una stringa (es. un cognome). Il programma deve:

1. Scrivere un file `nomi.bin` contenente 5 stringhe da 50 byte l'una.
2. Richiedere all'utente di inserire un indice numerico (da 0 a 4).
3. Utilizzare `fseek` per saltare direttamente a quell'indice senza scorrere i precedenti, leggere il cognome tramite `fread` e stamparlo.


```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define RECORD_SIZE 50
6  #define N_RECORDS 5
7
8  int main(void) {
9      // 1. Creazione file con 5 record da 50 byte fissi ciascuno
10     FILE *out = fopen("nomi.b.in", "wb");
11     if (out == NULL) return 1;
12     char buffer[RECORD_SIZE];
13     const char *nomi[N_RECORDS] = {"Rossi", "Bianchi", "Verdi", "
14                                     Russo", "Ferrari"};
15     for (int i = 0; i < N_RECORDS; i++) {
16         memset(buffer, '\0', RECORD_SIZE); // Puliamo il buffer con
17         byte null?
18         strcpy(buffer, nomi[i], RECORD_SIZE - 1);
19         fwrite(buffer, RECORD_SIZE, 1, out);
20     }
21     fclose(out);
22
23     // 2. Lettura mirata inserita da utente
24     int indice;
25     printf("Inserisci l'indice del record da leggere (0..%d): ",
26           N_RECORDS - 1);
27     if (scanf("%d", &indice) != 1 || indice < 0 || indice >=
28         N_RECORDS) {
29         fprintf(stderr, "Indice non valido.\n");
30         return 1;
31     }
32     FILE *in = fopen("nomi.b.in", "rb");
33     if (in == NULL) return 1;
34     // Saltiamo direttamente al record desiderato
35     fseek(in, indice * RECORD_SIZE, SEEK_SET);
36     char record_letto[RECORD_SIZE];
37     fread(record_letto, RECORD_SIZE, 1, in);
38     fclose(in);
39     printf("Record %d letto: \"%s\\n\"", indice, record_letto);
40     return 0;
41 }

```

Spiegazione dell'algoritmo: Garantendo record costanti di 50 byte tramite inizializzazione controllata con `memset` e `strcpy`, ogni record `i` si trova esattamente all'offset `i * RECORD_SIZE`. La `fseek` posiziona la testina a quell'indirizzo e la `fread` carica esclusivamente i successivi 50 byte.

Esercizio 4: Massimo Elemento Ricorsivo**Domanda**

(recursive\{}_max.c)

Si scriva una funzione C ricorsiva `int trova\_max\_ric(const int arr[], int size)` che trovi e restituisca il valore massimo all'interno di un array di interi.

Spiegazione dell'algoritmo: Scomponiamo il problema: il massimo di un array è pari al massimo tra il primo elemento (`arr[0]`) e il massimo di tutto il resto dell'array (`size - 1`). La convergenza è garantita dallo scioglimento del puntatore `arr + 1` e dal decremento di `size` fino al caso limite `size == 1`.

```

1  #include <stdio.h>
2
3  int trova_max_ric(const int arr[], int size) {
4      // Caso base: array con un solo elemento
5      if (size == 1) {
6          return arr[0];
7      }
8
9      // Passo ricorsivo: trova il massimo della restante parte dell'
10     // array
11     int max_resto = trova_max_ric(arr + 1, size - 1);
12     // Confronto tra l'elemento corrente e il massimo del resto
13     return (arr[0] > max_resto) ? arr[0] : max_resto;
14 }
15
16 int main(void) {
17     int dati[] = {12, 45, 7, 89, 32, 54, 21};
18     int n = 7;
19     printf("Il massimo valore dell'array e': %d\n", trova_max_ric(
20     // dati, n));
21     return 0;
22 }
```

Soluzione**Esercizio 5: Palindromia Ricorsiva****Domanda**

(recursive\{}_palindrome.c)

Si scriva una funzione C ricorsiva `int is\_palindrome(const char *str, int start, int end)` che restituisca 1 se la stringa compresa tra gli indici `start` ed `end` è palindroma, e 0 altrimenti.

Spiegazione dell'algoritmo: Verifichiamo se il carattere d'inizio start coincide con quello finale end. In caso positivo, demandiamo la validazione alla stringa interna incrementando start e decrementando end. Se raggiungiamo un solo carattere o lo scontro degli indici senza discrepanze, la parola è palindroma.

```

1  #include <stdio.h>
2  #include <string.h>
3
4  int is_palindrome(const char *str, int start, int end) {
5      // Caso base: se gli indici si incrociano o sono uguali, la
6      // parola e' palindroma
7      if (start >= end) {
8          return 1;
9      }
10     // Se i caratteri estremi sono diversi, non e' palindroma
11     if (str[start] != str[end]) {
12         return 0;
13     }
14     // Passo ricorsivo: stringiamo l'intervallo escludendo gli
15     // estremi verificati
16     return is_palindrome(str, start + 1, end - 1);
17 }
18
19 int main(void) {
20     const char *parola1 = "radar";
21     const char *parola2 = "catania";
22
23     printf("%s\n", is_palindrome(parola1, 0, strlen(parola1) - 1) ? "SI" : "NO");
24     printf("%s\n", is_palindrome(parola2, 0, strlen(parola2) - 1) ? "SI" : "NO");
25
26     return 0;
27 }

```

Soluzione

Esercizio 6: Copia Inversa di File Binari

Domanda

(bin\{}_reverse\{}_copy.c)

Si scriva un programma C che legga un file binario di interi (numeri.bin) e ne crei una copia chiamata numeri\{}_inversi.bin contenente gli stessi valori ma disposti in ordine speculare (dall'ultimo al primo). Sfruttare opportunamente la funzione fseek con SEEK\{}_END.

Spiegazione dell'algoritmo: Dopo aver dedotto il numero totale N di elementi leggendo la grandezza fisica del file, eseguiamo un ciclo iterativo decrescente. Ad ogni iterazione ci posizioniamo tramite `fseek` sul record `i` ad offset `i * sizeof(int)`, ne acquisiamo il singolo intero e lo accodiamo sequenzialmente nel file di output.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      // Creiamo un file di test binario con i numeri da 1 a 5
6      FILE *out_test = fopen("numeri.bin", "wb");
7      if (out_test != NULL) {
8          int temp[] = {1, 2, 3, 4, 5};
9          fwrite(temp, sizeof(int), 5, out_test);
10         fclose(out_test);
11     }
12     FILE *in = fopen("numeri.bin", "rb");
13     if (in == NULL) return 1;
14     // Calcoliamo quanti interi sono presenti
15     fseek(in, 0, SEEK_END);
16     long size_in_bytes = ftell(in);
17     long n_elements = size_in_bytes / sizeof(int);
18     FILE *out = fopen("numeri_inversi.bin", "wb");
19     if (out == NULL) {
20         fclose(in);
21         return 1;
22     }
23     // Leggiamo ed invertiamo
24     for (long i = n_elements - 1; i >= 0; i--) {
25         int val;
26         fread(&val, sizeof(int), 1, in);
27         fwrite(&val, sizeof(int), 1, out);
28     }
29     fclose(in);
30     fclose(out);
31     printf("Inversione binaria completata con successo.\n");
32     return 0;
33 }

```

Soluzione

Esercizio 7: Fibonacci Tail-Recursive vs Esponenziale

Domanda

(fibonacci\{}_compare.c)

Si scriva un programma C che implementi sia la versione classica a doppia ricorsione di Fibonacci sia la versione tail-recursive. Il programma deve richiedere all'utente un intero N ed evidenziare visivamente la differenza di velocità di esecuzione stampando i risultati di entrambi i calcoli.

Soluzione

Spiegazione dell'algoritmo: La ricorsione classica esplode con una complessità temporale di $O(2^N)$ a causa del ricalcolo ridondante delle foglie dell'albero delle chiamate. La versione tail-recursive linearizza l'algoritmo ($O(N)$) passando i risultati parziali intermedi come argomenti accumulati (a e b) senza mantenere chiamate in sospeso nello Stack.

```

1  #include <stdio.h>
2  #include <time.h>
3
4  // Versione standard non ottimizzata:  $O(2^N)$ 
5  long long fib_standard(int n) {
6      if (n == 0) return 0;
7      if (n == 1) return 1;
8      return fib_standard(n - 1) + fib_standard(n - 2);
9  }
10
11 // Versione tail-recursive ottimizzata:  $O(N)$  temporale,  $O(1)$  stack
12 long long fib_tail_helper(int n, long long a, long long b) {
13     if (n == 0) return a;
14     if (n == 1) return b;
15     return fib_tail_helper(n - 1, b, a + b);
16 }
17
18 long long fib_tail(int n) {
19     return fib_tail_helper(n, 0, 1);
20 }
21
22 int main(void) {
23     int n = 40; // Valore di test (soglia in cui lo standard
24               // incomincia ad essere molto lento)
25     printf("Calcolo di Fibonacci(%d):\n", n);
26     clock_t start = clock();
27     long long res_tail = fib_tail(n);
28     clock_t end = clock();
29     double time_tail = (double)(end - start) / CLOCKS_PER_SEC;
30     printf("[Tail Recursive] Risultato: %lld | Tempo: %f secondi\n",
31           res_tail, time_tail);
32
33     printf("Avvio calcolo standard non ottimizzato (attendere...):\n");
34     start = clock();
35     long long res_std = fib_standard(n);
36     end = clock();
37     double time_std = (double)(end - start) / CLOCKS_PER_SEC;
38     printf("[Standard] Risultato: %lld | Tempo: %f secondi\n",
39           res_std, time_std);
40     return 0;
41 }

```

Esercizio 8: Ricerca Binaria Ricorsiva

Domanda

(recursive\{}_binary\{}_search.c)

Si scriva un programma C contenente una funzione ricorsiva per effettuare la ricerca binaria su un array ordinato di interi: `int binary\{}_search\{}_ric(const int arr[], int low, int high, int target)`.

Spiegazione dell'algoritmo: Scomponiamo l'array calcolando l'indice mediano `mid`. Se il valore coincide, lo restituiamo. Altrimenti verifichiamo se il `target` sia inferiore o superiore ad `arr[mid]`, lanciando la medesima funzione ricorsiva solo sulla sottoporzione ordinata d'interesse.

```

1  #include <stdio.h>
2
3  int binary_search_ric(const int arr[], int low, int high, int
4      // Caso base: elemento non trovato
5      if (low > high) {
6          return -1;
7      }
8
9      int mid = low + (high - low) / 2;
10
11     // Caso base: elemento trovato al centro
12     if (arr[mid] == target) {
13         return mid;
14     }
15
16     // Passo ricorsivo: decidiamo quale meta' esplorare
17     if (arr[mid] > target) {
18         return binary_search_ric(arr, low, mid - 1, target);
19     } else {
20         return binary_search_ric(arr, mid + 1, high, target);
21     }
22 }
23
24 int main(void) {
25     int elenco[] = {2, 4, 8, 12, 16, 23, 38, 56, 72, 91};
26     int n = 10;
27     int target = 23;
28
29     int pos = binary_search_ric(elenco, 0, n - 1, target);
30     if (pos != -1) {
31         printf("Elemento %d trovato alla posizione %d\n", target,
32             pos);
33     } else {
34         printf("Elemento %d non presente\n", target);
35     }
36     return 0;

```

Soluzione

Esercizio 9: Hexadecimal File Dumper**Domanda**

(hex\{}_dumper.c)

Si realizzi una semplice utility C che prenda da riga di comando il nome di un file e lo stampi a schermo byte per byte in formato esadecimale (esattamente come un hexdump elementare), stampando 16 byte esadecimali per ciascuna riga.

Soluzione

Spiegazione dell'algoritmo: Leggiamo singolarmente i byte dallo stream aperto in modalità binaria grezza con `fgetc`. Tramite l'operatore modulo `\{\}\%` 16 determiniamo la fine della riga visualizzata stampando l'offset in byte esadecimale all'inizio ed un ritorno a capo ogni 16 caratteri estratti.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          fprintf(stderr, "Uso: %s [nome_file]\n", argv[0]);
7          return 1;
8      }
9
10     FILE *fp = fopen(argv[1], "rb");
11     if (fp == NULL) {
12         fprintf(stderr, "Errore nell'aprire il file '%s'\n", argv
13         [1]);
14         return 1;
15     }
16     int byte;
17     int contatore = 0;
18
19     printf("Offset      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E
20     \n");
21     while ((byte = fgetc(fp)) != EOF) {
22         if (contatore % 16 == 0) {
23             printf("%08X ", contatore);
24         }
25         printf("%02X ", byte);
26         contatore++;
27         if (contatore % 16 == 0) {
28             printf("\n");
29         }
30     }
31     printf("\n");
32 }
33
34     if (contatore % 16 != 0) {
35         printf("\n");
36     }
37     fclose(fp);
38     return 0;
39 }
40
41 }
```

Esercizio 10: Moltiplicazione Russa Ricorsiva

Domanda

(russian\{}_multiplication.c)

L'algoritmo di moltiplicazione russa (o moltiplicazione contadina) calcola il prodotto di due interi positivi A e B dimezzando ripetutamente A (divisione intera) e raddoppiando B ad ogni iterazione, finché A diventa pari a 1. Il risultato finale è la somma di tutti i valori di B in corrispondenza delle righe in cui A è dispari. Si scriva una funzione C ricorsiva `int prod\_russo(int a, int b)` che implementi questo algoritmo.

Spiegazione dell'algoritmo: Sfruttiamo l'induzione matematica: se a è dispari, il prodotto è pari a $b + \text{prod_russo}(a/2, b \times 2)$. Se a è pari, equivale semplicemente a $\text{prod_russo}(a/2, b \times 2)$. Il caso limite di terminazione è $a == 1$ che restituisce il valore cumulato di b .

```

1  #include <stdio.h>
2
3  int prod_russo(int a, int b) {
4      // caso base
5      if (a == 1) {
6          return b;
7      }
8      // Se a è dispari, accumuliamo b piu' il risultato della
9      // chiamata ricorsiva
10     if (a % 2 != 0) {
11         return b + prod_russo(a / 2, b * 2);
12     } else {
13         // Se a è pari, passiamo alla chiamata ricorsiva
14         // dimezzando a e raddoppiando b
15         return prod_russo(a / 2, b * 2);
16     }
17 }
18
19 int main(void) {
20     int x = 27;
21     int y = 35;
22     printf("Prodotto russo di %d * %d = %d (Verifica matematica: %d\n", x, y, prod_russo(x, y), x * y);
23     return 0;

```

Soluzione