



NoiStudiamo

DISPENSA DI LABORATORIO



Programmazione I Laboratorio

Compilazione, debug, file, memoria dinamica, liste,
ordinamento ed esempi svolti di programmi completi.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I diciassette capitoli della dispensa, dall'ambiente di lavoro agli esempi svolti. Tutti i programmi sono stati compilati e provati; nei listati il segno meno è tipografico.

1. Ambiente di lavoro e ciclo di sviluppo	3
2. Compilazione separata e file di intestazione	7
3. Input e output formattato in pratica	9
4. Schemi di programmazione ricorrenti	12
5. Funzioni e numeri casuali in pratica	15
6. Array e matrici in pratica	18
7. Puntatori in pratica	21
8. Stringhe in pratica	24
9. Memoria dinamica in pratica	27
10. Parametri da riga di comando	31
11. Lettura e scrittura di file	33
12. Liste, pile e code: implementazioni complete	37
13. Ordinare e cercare in pratica	41
14. Metodo per un progetto completo	45
15. Esempio svolto: matrice da file binario	47
16. Esempio svolto: archivio ordinato in una lista	53
17. Esempio svolto: coda di parole	59

Ambiente di lavoro e ciclo di sviluppo

Gli strumenti con cui si scrive, si compila, si esegue e si corregge un programma C su un sistema Linux, e il ciclo di lavoro che li collega.

Gli strumenti

Per programmare in C servono tre strumenti: un **editor di testo** con cui scrivere il file sorgente, un **compilatore** che lo traduce in un eseguibile e un **terminale** da cui lanciare i comandi. In ambiente Linux il compilatore di riferimento è **gcc** (GNU Compiler Collection); come editor va bene qualunque editor per programmatori, purché salvi testo semplice. Un ambiente integrato non è necessario: lavorare dal terminale obbliga a capire che cosa succede a ogni passo, ed è la condizione in cui si lavora quando non si dispone di altro.

I file sorgente hanno estensione `.c`, i file di intestazione `.h`. È buona norma creare una cartella per ogni esercizio e tenervi il sorgente, l'eseguibile e i file di dati usati per le prove.

COMANDI ESSENZIALI DEL TERMINALE

COMANDO	EFFETTO
<code>pwd</code>	mostra la cartella corrente
<code>ls -l</code>	elenca i file della cartella con dimensione e data
<code>cd nome</code>	entra nella cartella nome; <code>cd ..</code> risale di un livello
<code>mkdir nome</code>	crea una cartella
<code>cp a b, mv a b</code>	copia o sposta (rinomina) un file
<code>rm nome</code>	cancella un file, senza cestino
<code>cat nome</code>	stampa il contenuto di un file di testo
<code>man 3 printf</code>	apre la documentazione di una funzione della libreria C

Compilare con gcc

Il comando minimo per compilare un programma è `gcc prova.c`, che produce un eseguibile chiamato `a.out`. Nella pratica si usano sempre alcune opzioni che rendono il compilatore più severo e il risultato più facile da correggere.

OPZIONE	SIGNIFICATO
-o nome	dà un nome all'eseguibile al posto di a.out
-Wall -Wextra	attiva quasi tutti gli avvisi (warning) del compilatore
-std=c11	compila secondo lo standard C11
-g	inserisce le informazioni per il debugger
-lm	collega la libreria matematica (necessaria con math.h)
-c	compila senza collegare e produce il file oggetto .o
-fsanitize=address	controlla in esecuzione gli accessi alla memoria

La riga di comando consigliata per tutti gli esercizi di questa dispensa è quindi:

```
gcc -Wall -Wextra -std=c11 -g prova.c -o prova
./prova
```

ATTENZIONE

Il segno meno nei listati

Nei listati di questa dispensa le opzioni e gli operatori sono stampati con il segno meno tipografico, per chiarezza di lettura. Quando si digitano i comandi o il codice si usa il normale tasto meno della tastiera: copiando e incollando il carattere tipografico, il compilatore e il terminale non lo riconoscono.

ERRORI E AVVISI

Il compilatore distingue due categorie di messaggi. Gli **errori** (error) impediscono di produrre l'eseguibile: una parentesi non chiusa, un punto e virgola mancante, una variabile non dichiarata. Gli **avvisi** (warning) non bloccano la compilazione, ma segnalano costrutti quasi sempre sbagliati: una variabile usata prima di essere inizializzata, un formato di printf che non corrisponde al tipo dell'argomento, un = usato al posto di == in una condizione. Un programma corretto si compila **senza alcun avviso**: un avviso ignorato è spesso il bug che si cercherà per un'ora.

Ogni messaggio indica file, riga e colonna, per esempio prova.c:12:5: error: expected ' ; '. Si corregge sempre **il primo messaggio** e si ricompila: un solo errore di sintassi può generarne a cascata molti altri, che scompaiono da soli quando il primo è risolto.

ESEMPIO SVOLTO

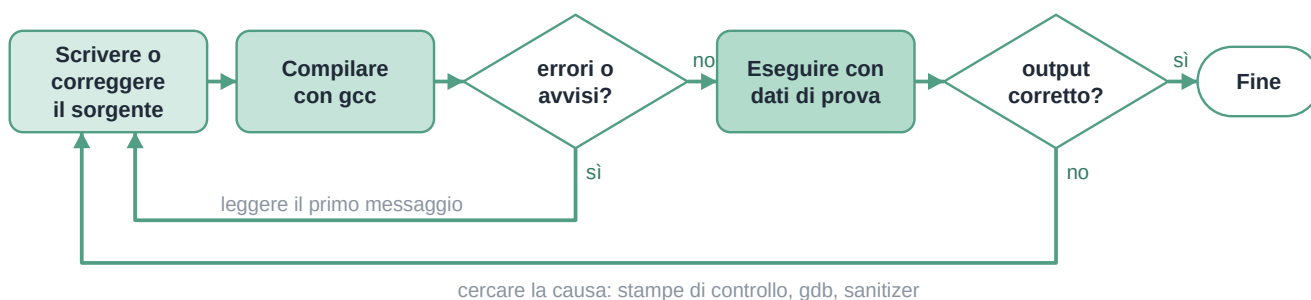
Leggere un avviso

```
int x;
if (x = 0)
    printf("zero\n");
```

gcc con -Wall segnala che si sta usando il risultato di un assegnamento come condizione e che x è usata senza essere inizializzata. Le correzioni sono due: scrivere x == 0 e dare un valore a x prima del confronto. Senza le opzioni di avviso il programma si compila in silenzio e stampa sempre nulla, perché l'assegnamento vale 0.

Il ciclo di lavoro

Lo sviluppo di un programma è un ciclo che si ripete molte volte: si scrive o si modifica il sorgente, si compila, si correggono errori e avvisi, si esegue con dati di prova, si confronta l'output con quello atteso e, se non corrisponde, si cerca la causa e si torna al sorgente.



Il ciclo di lavoro: si torna al sorgente sia dopo un errore di compilazione sia dopo un risultato sbagliato.

Due abitudini rendono il ciclo molto più breve. La prima è **compilare spesso**: dopo ogni funzione scritta, non alla fine del programma intero, in modo che gli errori siano sempre in poche righe appena scritte. La seconda è **preparare i dati di prova prima di eseguire**, calcolando a mano il risultato atteso: solo così si può dire se l'output è giusto.

Correggere gli errori in esecuzione

Un programma che si compila senza avvisi può ancora essere sbagliato. Gli strumenti per trovare l'errore sono, in ordine di semplicità, le stampe di controllo, il debugger e i controlli automatici sulla memoria.

STAMPE DI CONTROLLO

Il metodo più immediato è aggiungere stampe temporanee che mostrano il valore delle variabili nei punti critici, per esempio all'inizio di ogni iterazione di un ciclo. Conviene scriverle sul canale degli errori, così non si mescolano con l'output del programma e si eliminano facilmente al termine:

```
fprintf(stderr, "DEBUG i=%d somma=%d\n", i, somma);
```

IL DEBUGGER GDB

Il debugger permette di fermare il programma in un punto, eseguirlo un'istruzione alla volta e ispezionare le variabili. Il programma va compilato con `-g`; poi si avvia con `gdb ./prova`.

COMANDO GDB	EFFETTO
<code>break main, break 25</code>	punto di arresto su una funzione o su una riga
<code>run arg1 arg2</code>	avvia il programma con i parametri indicati
<code>next, step</code>	esegue una riga; step entra nelle funzioni chiamate
<code>print x, print v[i]</code>	mostra il valore di un'espressione
<code>display i</code>	mostra i a ogni passo
<code>backtrace</code>	mostra la catena delle chiamate attive
<code>continue, quit</code>	prosegue fino al prossimo arresto; esce

Quando un programma termina con **Segmentation fault**, eseguirlo dentro gdb e poi digitare `backtrace` mostra la riga esatta in cui è avvenuto l'accesso illegale e le funzioni che vi hanno portato.

I CONTROLLI SULLA MEMORIA

Molti errori del C, come uscire dai limiti di un array, usare memoria già liberata o dimenticare una `free`, non producono subito un crash: il programma continua con dati corrotti. L'opzione `-fsanitize=address` di gcc inserisce nel programma controlli che lo fermano **nel momento esatto** dell'accesso sbagliato, indicando riga e tipo di errore, e che al termine segnalano la memoria allocata e mai liberata. Tutti i programmi di questa dispensa sono stati provati con questa opzione.

DA RICORDARE

Come si procede quando un programma non funziona?

Prima si compila con `-Wall -Wextra` e si eliminano tutti gli avvisi. Poi si esegue con dati di prova di cui si conosce il risultato. Se l'output è sbagliato si localizza il punto in cui i valori smettono di essere quelli attesi, con stampe su `stderr` o con gdb. Se il programma si interrompe o si comporta in modo casuale si ricompila con `-g -fsanitize=address`, che indica l'accesso illegale alla memoria.

Compilazione separata e file di intestazione

Come si divide un programma in più file sorgente, che cosa va in un file di intestazione e come si compila e si collega il tutto.

Perché dividere il programma

Un programma di poche centinaia di righe si può tenere in un unico file. Quando cresce conviene dividerlo in **moduli**: ogni modulo raccoglie le funzioni che riguardano lo stesso argomento (per esempio la gestione di una lista) e può essere compilato, provato e riusato in modo indipendente. La divisione rende anche più veloce la ricompilazione, perché dopo una modifica si ricompila solo il modulo toccato.

Un modulo è formato da due file. Il **file di intestazione** `lista.h` contiene l'**interfaccia**: le definizioni di tipo e i prototipi delle funzioni che gli altri file possono usare. Il **file di implementazione** `lista.c` contiene il corpo delle funzioni. Chi usa il modulo include soltanto l'intestazione.

Un esempio completo

Il modulo seguente fornisce due funzioni sugli array di interi.

```
/* vettori.h: interfaccia del modulo */
#ifndef VETTORI_H
#define VETTORI_H

void stampaVettore(const int v[], int n);
int massimo(const int v[], int n);

#endif
```

```
/* vettori.c: implementazione del modulo */
#include <stdio.h>
#include "vettori.h"

void stampaVettore(const int v[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}

int massimo(const int v[], int n)
{
    int m = v[0];
    for (int i = 1; i < n; i++)
        if (v[i] > m)
            m = v[i];
    return m;
}
/* il primo elemento è il massimo provvisorio */
```

```

/* main.c: usa il modulo */
#include <stdio.h>
#include "vettori.h"

int main(void)
{
    int dati[5] = {4, 9, 2, 7, 5};
    stampaVettore(dati, 5);
    printf("Massimo: %d\n", massimo(dati, 5));
    return 0;
}

```

Le intestazioni di sistema si includono con le parentesi angolari, `#include <stdio.h>`, e il preprocessore le cerca nelle cartelle di sistema; quelle del programma si includono con le virgolette, `#include "vettori.h"`, e vengono cercate prima nella cartella corrente.

LA GUARDIA DI INCLUSIONE

Le tre righe `#ifndef`, `#define` ed `#endif` costituiscono la **guardia di inclusione**. Se lo stesso file di intestazione viene incluso due volte nella stessa unità di compilazione, cosa che succede facilmente quando un'intestazione ne include un'altra, la seconda volta il simbolo `VETTORI_H` è già definito e il contenuto viene saltato. Senza guardia le definizioni di tipo verrebbero ripetute e il compilatore segnalerebbe un errore.

Compilare e collegare

Si può compilare tutto con un solo comando, che compila ogni file e poi collega i risultati:

```
gcc -Wall -Wextra -std=c11 main.c vettori.c -o programma
```

Oppure si possono separare le due fasi: prima si produce un file oggetto per ogni sorgente con `-c`, poi si collegano i file oggetto. Dopo una modifica a `main.c` basta ricompilare quel solo file.

```

gcc -Wall -Wextra -std=c11 -c vettori.c      # produce vettori.o
gcc -Wall -Wextra -std=c11 -c main.c         # produce main.o
gcc main.o vettori.o -o programma           # collegamento

```

ATTENZIONE

Che cosa non va nel file di intestazione

Il file di intestazione contiene dichiarazioni, non definizioni che occupano memoria o generano codice. Se vi si scrive il corpo di una funzione o si definisce una variabile, e l'intestazione è inclusa da due file, il collegamento fallisce con un errore di **definizione multipla** (multiple definition). Allo stesso modo non si include mai un file `.c` con `#include`.

DA RICORDARE

Qual è la differenza tra un errore di compilazione e un errore di collegamento?

L'errore di compilazione riguarda un singolo file sorgente: sintassi sbagliata, tipi incompatibili, identificatori non dichiarati. L'errore di collegamento (linker) emerge solo quando si uniscono i file oggetto: una funzione dichiarata ma mai definita (undefined reference), una funzione definita due volte (multiple definition), una libreria non indicata, come la libreria matematica senza `-lm`.

Input e output formattato in pratica

L'uso corretto di `printf` e `scanf` nei casi che si incontrano davvero: formati, larghezze, controllo del valore restituito e gestione dell'input non valido.

Stampare con `printf`

`printf` riceve una **stringa di formato** e gli argomenti da stampare; ogni specificatore di conversione, introdotto da %, deve corrispondere per tipo all'argomento nella stessa posizione. Gli avvisi di gcc segnalano le incongruenze, che altrimenti producono valori senza senso.

FORMATO	TIPO	ESEMPIO DI OUTPUT
%d	int	42
%5d	int, larghezza 5 a destra	42
%-5d	int, larghezza 5 a sinistra	42
%05d	int, riempito di zeri	00042
%f	double (o float promosso)	3.141593
%.2f	due cifre decimali	3.14
%8.2f	larghezza 8, due decimali	3.14
%e	notazione esponenziale	3.141593e+00
%c	char	A
%s	stringa	ciao
%x, %o	int in esadecimale, ottale	2a, 52
%%	il carattere percento	%

La **larghezza** è il numero minimo di caratteri occupati dal campo e serve a incolonnare i dati: stampando ogni elemento di una matrice con `%5d` le colonne risultano allineate anche se i numeri hanno un numero diverso di cifre.

```
#include <stdio.h>

int main(void)
{
    double prezzi[3] = {2.5, 13.75, 120.0};
    printf("%-10s%10s\n", "Articolo", "Prezzo");
    for (int i = 0; i < 3; i++)
        printf("%-10d%10.2f\n", i + 1, prezzi[i]); /* colonne di 10 caratteri */
    return 0;
}
```

Leggere con scanf

scanf legge dal canale di input secondo un formato e scrive i valori letti **negli indirizzi** che riceve: per questo le variabili si passano con l'operatore &, tranne gli array di caratteri, il cui nome è già un indirizzo.

FORMATO	LETTURA
%d	int, in una variabile int
%f	float; per un double serve %lf
%c	un carattere qualsiasi, anche spazio o a capo
%C	uno spazio prima salta gli spazi e gli a capo
%s	una parola, fino al primo spazio
%10s	una parola di al massimo 10 caratteri

ATTENZIONE

Formati diversi tra printf e scanf

In printf un float viene promosso a double, quindi %f stampa sia float sia double. In scanf invece la funzione deve sapere quanti byte scrivere: %f per float, %lf per double. Leggere un double con %f lascia nella variabile un valore privo di significato.

IL VALORE RESTITUITO DA SCANF

scanf restituisce il **numero di campi letti e assegnati con successo**, oppure EOF se l'input è terminato prima della prima conversione. Controllare questo valore è l'unico modo per sapere se la lettura è riuscita: se l'utente digita lettere dove ci si aspetta un numero la funzione restituisce 0, la variabile resta invariata e i caratteri non validi restano nel buffer di input.

```
#include <stdio.h>

int main(void)
{
    int n;
    printf("Inserisci un intero positivo: ");
    while (scanf("%d", &n) != 1 || n <= 0) {
        int c;
        while ((c = getchar()) != '\n' && c != EOF) /* scarta il resto della riga */
            ;
        if (c == EOF)
            return 1;
        printf("Valore non valido, riprova: ");
    }
    printf("Hai inserito %d\n", n);
    return 0;
}
```

Il ciclo interno svuota la riga rimasta nel buffer; senza di esso scanf proverebbe all'infinito a convertire gli stessi caratteri non validi. La variabile c è di tipo int perché deve poter contenere anche il valore EOF, che non è un carattere.

LEGGERE UN CARATTERE DOPO UN NUMERO

Dopo `scanf("%d", &n)` nel buffer resta il carattere a capo digitato con Invio. Una successiva `scanf("%c", &c)` legge proprio quell'a capo, e il programma sembra saltare la domanda. La soluzione è mettere uno spazio prima di `%c`, che consuma tutti gli spazi bianchi precedenti:

```
scanf("%d", &n);  
scanf(" %c", &risposta);    /* lo spazio salta l'a capo rimasto */
```

Leggere una riga intera

Per leggere una riga che può contenere spazi si usa `fgets(s, dim, stdin)`, che legge al massimo `dim` meno 1 caratteri, si ferma dopo l'a capo e termina sempre la stringa. L'a capo, se c'è, resta nella stringa e va tolto. La funzione `gets` non controlla la dimensione, è stata eliminata dallo standard e non va mai usata.

```
char nome[50];  
if (fgets(nome, sizeof nome, stdin) != NULL)  
    nome[strcspn(nome, "\n")] = '\0';    /* strcspn: posizione del primo '\n' */
```

DA RICORDARE

Perché si controlla sempre il valore restituito da `scanf`?

Perché è l'unica informazione su come è andata la lettura: il numero di campi assegnati, oppure EOF a fine input. Se l'input non è valido le variabili restano non inizializzate o con il valore precedente, e un programma che non controlla prosegue con dati sbagliati; in un ciclo di lettura, inoltre, i caratteri non validi restano nel buffer e provocano un ciclo infinito.

Schemi di programmazione ricorrenti

I modelli di ciclo che compaiono in quasi ogni programma: contatori, accumulatori, ricerca del massimo, flag, sentinelle e convalida dei dati, con esercizi svolti.

Contatori e accumulatori

Un **contatore** è una variabile che parte da 0 e aumenta di 1 ogni volta che si verifica un evento; un **accumulatore** parte da un valore neutro (0 per le somme, 1 per i prodotti) e a ogni iterazione incorpora un nuovo valore. Entrambi vanno **inizializzati prima del ciclo**: una variabile locale non inizializzata contiene un valore casuale.

```
int conta = 0, somma = 0;
for (int i = 0; i < n; i++) {
    if (v[i] % 2 == 0)
        conta++;           /* contatore: quanti pari */
    somma += v[i];         /* accumulatore: somma di tutti */
}
```

Massimo e minimo

Per trovare il massimo si prende il **primo elemento come massimo provvisorio** e lo si confronta con tutti gli altri. Inizializzare il massimo a 0 è un errore classico: se tutti i valori sono negativi il risultato è 0, che non compare nei dati. Quando serve anche la posizione si memorizza l'indice invece del valore.

```
int imax = 0;                /* indice del massimo provvisorio */
for (int i = 1; i < n; i++)
    if (v[i] > v[imax])      /* con > si tiene la prima occorrenza */
        imax = i;
```

Flag e ricerca con uscita anticipata

Un **flag** è una variabile logica che registra se una condizione si è verificata almeno una volta. Per verificare una proprietà che deve valere per **tutti** gli elementi si parte da vero e si mette a falso al primo controesempio; per una proprietà che deve valere per **almeno uno** si parte da falso. Spesso il ciclo può terminare appena l'esito è deciso.

```
int ordinato = 1;            /* ipotesi: è ordinato */
for (int i = 0; i < n - 1 && ordinato; i++)
    if (v[i] > v[i + 1])
        ordinato = 0;       /* trovato un controesempio */
```

Sentinella e convalida

Quando il numero di dati non è noto in anticipo si legge fino a un **valore sentinella**, scelto in modo che non possa essere un dato valido, oppure fino alla fine dell'input. Lo schema corretto legge il primo valore **prima** del ciclo e il successivo **alla fine** del corpo, così la sentinella non viene mai elaborata.

ESEMPIO SVOLTO

Media dei voti con sentinella

Si leggono voti interi da 18 a 30 fino all'inserimento di 0; i voti fuori intervallo vengono rifiutati e non contano. Al termine si stampano numero di voti, media e voto massimo.

```
#include <stdio.h>

int main(void)
{
    int voto, conta = 0, somma = 0, max = 0;
    printf("Voto (0 per terminare): ");
    while (scanf("%d", &voto) == 1 && voto != 0) {
        if (voto < 18 || voto > 30)
            printf("Voto non valido\n");          /* convalida */
        else {
            conta++;
            somma += voto;
            if (voto > max)                        /* qui 0 va bene: i voti sono >= 18 */
                max = voto;
        }
        printf("Voto (0 per terminare): ");
    }
    if (conta > 0)
        printf("%d voti, media %.2f, massimo %d\n", conta, (double)somma / conta, max);
    else
        printf("Nessun voto inserito\n");
    return 0;
}
```

La conversione `(double)somma / conta` è necessaria: tra due interi la divisione è intera e la parte decimale della media andrebbe persa. Il caso senza voti è gestito a parte per non dividere per zero.

Contare caratteri, parole e righe

L'esempio seguente legge l'input un carattere alla volta fino a EOF (da tastiera si genera con Ctrl+D su Linux) e conta caratteri, righe e parole. Una parola inizia quando si passa da uno spazio bianco a un carattere non bianco: lo **stato** «dentro una parola» è un flag.

```

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int c, caratteri = 0, righe = 0, parole = 0;
    int dentro = 0; /* flag: siamo dentro una parola? */
    while ((c = getchar()) != EOF) {
        caratteri++;
        if (c == '\n')
            righe++;
        if (isspace(c))
            dentro = 0;
        else if (!dentro) { /* inizio di una nuova parola */
            dentro = 1;
            parole++;
        }
    }
    printf("%d righe, %d parole, %d caratteri\n", righe, parole, caratteri);
    return 0;
}

```

Lanciato come `./conta < testo.txt` il programma legge il file al posto della tastiera: è la **ridirezione dell'input**, che permette di provare un programma sempre con gli stessi dati senza digitarli ogni volta. Allo stesso modo `./conta > risultato.txt` scrive l'output in un file.

DA RICORDARE

Quali sono gli errori più comuni negli schemi di ciclo?

Contatori e accumulatori non inizializzati; massimo inizializzato a 0 invece che al primo elemento; sentinella elaborata come dato perché la lettura avviene all'inizio del corpo invece che alla fine; divisione intera nel calcolo di una media; indice che arriva a n invece di fermarsi a $n - 1$ quando nel corpo si accede a $v[i + 1]$.

Funzioni e numeri casuali in pratica

Come si scompone un esercizio in funzioni, come si progettano parametri e valori di ritorno e come si generano numeri pseudocasuali in un intervallo.

Progettare le funzioni

Una buona funzione svolge **un solo compito**, ha un nome che lo descrive e comunica con il resto del programma soltanto attraverso parametri e valore di ritorno, senza variabili globali. Prima di scriverne il corpo conviene fissarne il **prototipo**: che cosa riceve, che cosa restituisce e che cosa modifica.

SITUAZIONE	SCELTA
calcola un solo valore	lo restituisce con return
verifica una proprietà	restituisce int, 1 vero e 0 falso
deve produrre più risultati	riceve puntatori alle variabili da riempire
lavora su un array	riceve l'array e la sua dimensione
legge un array senza modificarlo	lo riceve come <code>const int v[]</code>
può fallire	restituisce un codice di esito

ESEMPIO SVOLTO

Verificare una data

Una funzione stabilisce se giorno, mese e anno formano una data valida; usa una seconda funzione per gli anni bisestili. Un anno è bisestile se è divisibile per 4 e non per 100, oppure se è divisibile per 400.

```
#include <stdio.h>

int bisestile(int a)
{
    return (a % 4 == 0 && a % 100 != 0) || a % 400 == 0;
}

int giorniNelMese(int m, int a)
{
    switch (m) {
        case 2:
            return bisestile(a) ? 29 : 28;
        case 4: case 6: case 9: case 11:
            return 30;
        default:
            return 31;
    }
}
```

```

int dataValida(int g, int m, int a)
{
    return m >= 1 && m <= 12 && g >= 1 && g <= giorniNelMese(m, a);
}

int main(void)
{
    int g, m, a;
    printf("Data (gg mm aaaa): ");
    if (scanf("%d %d %d", &g, &m, &a) != 3)
        return 1;
    printf("%02d/%02d/%d %s valida\n", g, m, a, dataValida(g, m, a) ? "è" : "non è");
    return 0;
}

```

Ogni funzione è breve e verificabile da sola: `bisestile(1900)` deve dare 0, `bisestile(2000)` deve dare 1. La condizione in `dataValida` controlla il mese prima di usarlo in `giorniNelMese`, sfruttando la valutazione in cortocircuito di `&&`.

Numeri pseudocasuali

La funzione `rand`, dichiarata in `stdlib.h`, restituisce un intero pseudocasuale da 0 a `RAND_MAX`. La sequenza è determinata da un **seme**, impostato con `srand`: con lo stesso seme si ottiene sempre la stessa sequenza, il che è utile per riprodurre una prova. Per avere sequenze diverse a ogni esecuzione si usa come seme l'ora corrente, `srand(time(NULL))`, chiamata **una sola volta** all'inizio di `main`.

ESPRESSIONE	INTERVALLO
<code>rand() % n</code>	da 0 a n meno 1
<code>a + rand() % (b - a + 1)</code>	da a a b compresi
<code>(double)rand() / RAND_MAX</code>	reale da 0 a 1

ESEMPIO SVOLTO

Indovinare un numero

Il programma sceglie un numero da 1 a 100 e l'utente prova a indovinarlo ricevendo ogni volta l'indicazione «più alto» o «più basso».

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int casuale(int a, int b)
{
    return a + rand() % (b - a + 1);    /* intero da a a b */
}

int main(void)
{
    srand(time(NULL));                /* seme: una volta sola */
    int segreto = casuale(1, 100), tentativo, tentativi = 0;
    do {
        printf("Tentativo: ");
        if (scanf("%d", &tentativo) != 1)
            return 1;
        tentativi++;
        if (tentativo < segreto)
            printf("Più alto\n");
        else if (tentativo > segreto)
            printf("Più basso\n");
    } while (tentativo != segreto);
    printf("Indovinato in %d tentativi\n", tentativi);

    return 0;
}

```

La strategia migliore per l'utente è proporre ogni volta il valore centrale dell'intervallo rimasto: è la ricerca dicotomica, che trova qualunque numero da 1 a 100 in al massimo 7 tentativi.

ESEMPIO SVOLTO

Simulare il lancio di due dadi

Si lanciano due dadi 6000 volte e si conta quante volte esce ciascuna somma da 2 a 12, usando un array di contatori indicizzato dalla somma.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    int freq[13] = {0};                /* indici da 2 a 12 usati */
    srand(time(NULL));
    for (int i = 0; i < 6000; i++) {
        int s = (1 + rand() % 6) + (1 + rand() % 6);
        freq[s]++;
    }
    for (int s = 2; s <= 12; s++)
        printf("%2d: %4d\n", s, freq[s]);
    return 0;
}

```

Il 7 risulta la somma più frequente (circa 1000 volte, cioè 6 casi su 36), mentre 2 e 12 escono circa 167 volte ciascuno.

Array e matrici in pratica

Esercizi svolti sugli array monodimensionali e bidimensionali: lettura, stampa incolonnata, operazioni per righe e colonne e passaggio alle funzioni.

Array monodimensionali

Un array ha dimensione fissata alla compilazione; il numero di elementi effettivamente usati va tenuto in una variabile separata e passato a ogni funzione insieme all'array. Gli indici vanno da 0 a n meno 1, e il C **non controlla** che l'indice sia nei limiti: scrivere oltre la fine corrompe altre variabili senza alcun messaggio.

ESEMPIO SVOLTO

Invertire un array sul posto

Per invertire un array senza usare un secondo array si scambiano il primo e l'ultimo elemento, poi il secondo e il penultimo, e così via fino al centro.

```
void inverti(int v[], int n)
{
    for (int i = 0; i < n / 2; i++) {        /* basta arrivare a metà */
        int t = v[i];
        v[i] = v[n - 1 - i];
        v[n - 1 - i] = t;
    }
}
```

Se il ciclo arrivasse fino a n, ogni coppia verrebbe scambiata due volte e l'array tornerebbe identico all'originale.

ESEMPIO SVOLTO

Eliminare un elemento

Per eliminare l'elemento di indice k si spostano indietro di una posizione tutti gli elementi successivi e si riduce di uno il numero di elementi usati, passato per indirizzo.

```
void elimina(int v[], int *n, int k)
{
    for (int i = k; i < *n - 1; i++)
        v[i] = v[i + 1];                /* sposta a sinistra */
    (*n)--;
}
```

Matrici

Una matrice dichiarata come `int m[R][C]` è memorizzata per righe, una dopo l'altra. Quando la si passa a una funzione, il numero di colonne deve essere noto al compilatore per calcolare la posizione di `m[i][j]`, quindi va indicato nel parametro: `void stampa(int m[][C], int righe)`.

ESEMPIO SVOLTO

Somme per righe e per colonne

Il programma riempie una matrice 3×4 con valori casuali da 0 a 9, la stampa incolonnata aggiungendo a destra la somma di ogni riga e in basso la somma di ogni colonna.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define R 3
#define C 4

void riempi(int m[][C])
{
    for (int i = 0; i < R; i++)
        for (int j = 0; j < C; j++)
            m[i][j] = rand() % 10;
}

void stampaConSomme(int m[][C])
{
    int sommaCol[C] = {0};
    for (int i = 0; i < R; i++) {
        int sommaRiga = 0;
        for (int j = 0; j < C; j++) {
            printf("%4d", m[i][j]);
            sommaRiga += m[i][j];
            sommaCol[j] += m[i][j];
        }

        printf(" |%4d\n", sommaRiga);
    }
    for (int j = 0; j < C; j++)
        printf("%4d", sommaCol[j]);
    printf("\n");
}

int main(void)
{
    int m[R][C];
    srand(time(NULL));
    riempi(m);
    stampaConSomme(m);
    return 0;
}
```

ESEMPIO SVOLTO

Matrice simmetrica e trasposta

Una matrice quadrata è simmetrica se $m[i][j]$ è uguale a $m[j][i]$ per ogni coppia di indici. Basta controllare gli elementi sopra la diagonale, e ci si può fermare al primo controesempio. Per trasporre sul posto si scambia ogni elemento sopra la diagonale con il suo simmetrico.

```
#define N 4

int simmetrica(int m[][N])
{
    for (int i = 0; i < N; i++)
        for (int j = i + 1; j < N; j++) /* solo sopra la diagonale */
            if (m[i][j] != m[j][i])
                return 0;
    return 1;
}

void trasponi(int m[][N])
{
    for (int i = 0; i < N; i++)
        for (int j = i + 1; j < N; j++) {
            int t = m[i][j];
            m[i][j] = m[j][i];
            m[j][i] = t;
        }
}
```

Anche qui la scelta di partire da $j = i + 1$ è essenziale: scorrendo tutta la matrice ogni coppia verrebbe scambiata due volte.

DA RICORDARE

Come si scorrono le diagonali di una matrice quadrata $N \times N$?

La diagonale principale è formata dagli elementi con $i == j$, e si scorre con un solo ciclo su $m[i][i]$. La diagonale secondaria è formata dagli elementi con $i + j == N - 1$, cioè $m[i][N - 1 - i]$. Gli elementi sopra la diagonale principale hanno $j > i$, quelli sotto hanno $j < i$.

Puntatori in pratica

Esercizi svolti sull'uso dei puntatori come parametri di uscita, per restituire più risultati da una funzione, e sulla scansione di array con l'aritmetica dei puntatori.

Parametri di uscita

Una funzione C può restituire un solo valore con `return`. Quando deve produrre più risultati, chi chiama le passa gli **indirizzi** delle variabili da riempire e la funzione vi scrive attraverso i puntatori. Il valore di ritorno resta libero per comunicare l'**esito** dell'operazione.

ATTENZIONE

Errori tipici con i puntatori

Un puntatore dichiarato e non inizializzato contiene un indirizzo casuale: scriverci attraverso corrompe la memoria. Un puntatore a una variabile locale non deve essere restituito da una funzione, perché la variabile cessa di esistere al ritorno. Prima di dereferenziare un puntatore che può valere `NULL`, per esempio il risultato di `malloc` o di `fopen`, lo si confronta con `NULL`.

ESEMPIO SVOLTO

Equazione di secondo grado

La funzione riceve i coefficienti di $ax^2 + bx + c = 0$, restituisce il numero di soluzioni reali e scrive le soluzioni nelle variabili indicate da `x1` e `x2`.

```
#include <stdio.h>
#include <math.h>

/* restituisce il numero di soluzioni reali: 0, 1 o 2 (a diverso da 0) */
int risolvi(double a, double b, double c, double *x1, double *x2)
{
    double delta = b * b - 4 * a * c;
    if (delta < 0)
        return 0;
    *x1 = (-b - sqrt(delta)) / (2 * a);
    *x2 = (-b + sqrt(delta)) / (2 * a);
    return delta == 0 ? 1 : 2;
}

int main(void)
{
    double a, b, c, x1, x2;
    printf("Coefficienti a b c: ");
    if (scanf("%lf %lf %lf", &a, &b, &c) != 3 || a == 0)
        return 1;
    int n = risolvi(a, b, c, &x1, &x2);
    if (n == 0)
        printf("Nessuna soluzione reale\n");
    else if (n == 1)
```

```

        printf("Soluzione doppia: %.3f\n", x1);
    else
        printf("Soluzioni: %.3f e %.3f\n", x1, x2);
    return 0;
}

```

Il programma usa `sqrt`, quindi va compilato con `-lm`. Con i coefficienti 1, meno 3 e 2 stampa le soluzioni 1 e 2.

ESEMPIO SVOLTO

Retta per due punti

Dati due punti (x_1, y_1) e (x_2, y_2) con ascisse diverse, la funzione calcola coefficiente angolare m e termine noto q della retta $y = m x + q$ che li attraversa.

```

/* restituisce 0 se la retta è verticale, 1 altrimenti */
int retta(double x1, double y1, double x2, double y2, double *m, double *q)
{
    if (x1 == x2)
        return 0;
    *m = (y2 - y1) / (x2 - x1);
    *q = y1 - *m * x1;
    return 1;
}

```

Nella seconda assegnazione `*m * x1` si legge «il valore puntato da m , moltiplicato per $x1$ »: lo stesso simbolo indica la dereferenziazione quando è unario e la moltiplicazione quando è binario.

ESEMPIO SVOLTO

Contare positivi e inversioni

La funzione scorre un array e restituisce attraverso due puntatori il numero di elementi positivi e il numero di **inversioni**, cioè di coppie di elementi adiacenti in cui il primo è maggiore del secondo.

```

void analizza(const int v[], int n, int *positivi, int *inversioni)
{
    *positivi = 0; /* i risultati si azzerano qui */
    *inversioni = 0;
    for (int i = 0; i < n; i++) {
        if (v[i] > 0)
            (*positivi)++;
        if (i < n - 1 && v[i] > v[i + 1])
            (*inversioni)++;
    }
}

```

Le parentesi in `(*positivi)++` sono obbligatorie: `*positivi++` incrementerebbe il puntatore e non il valore puntato, perché il postincremento ha precedenza sulla dereferenziazione.

Scorrere un array con un puntatore

Il nome di un array, usato in un'espressione, è l'indirizzo del suo primo elemento, e `v + i` è l'indirizzo dell'elemento di indice i . Si può quindi scorrere un array con un puntatore che parte dall'inizio e avanza fino a un puntatore che indica la posizione successiva all'ultimo elemento.

```
int somma(const int *v, int n)
{
    int s = 0;
    for (const int *p = v; p < v + n; p++)    /* p scorre gli elementi */
        s += *p;
    return s;
}
```

Lo stesso schema, con un puntatore in lettura e uno in scrittura, permette di scorrere due array in parallelo. La funzione seguente copia in `dst` i soli elementi positivi di `src` e restituisce quanti ne ha copiati; il puntatore `q` avanza solo quando un elemento viene scritto.

```
int copiaPositivi(const int *src, int n, int *dst)
{
    int *q = dst;                                /* prossima posizione libera */
    for (const int *p = src; p < src + n; p++)
        if (*p > 0)
            *q++ = *p;                            /* scrivi, poi avanza q */
    return q - dst;                                /* differenza tra puntatori */
}
```

DA RICORDARE

Quando un parametro di una funzione deve essere un puntatore?

Quando la funzione deve **modificare** una variabile di chi la chiama, perché in C i parametri si passano per valore e la funzione riceve una copia. È il caso dei risultati multipli e dei contatori da aggiornare. Un array si passa sempre come puntatore al primo elemento: per dichiarare che la funzione non lo modifica si usa `const`. La differenza tra due puntatori allo stesso array è il numero di elementi che li separano.

Stringhe in pratica

Le operazioni sulle stringhe che ricorrono negli esercizi: lunghezza e copia sicura, confronto, ricerca di caratteri, estensioni dei file, scomposizione in parole e conversione in numero.

Promemoria sulle funzioni di string.h

FUNZIONE	EFFETTO
strlen(s)	numero di caratteri, terminatore escluso
strcpy(d, s)	copia s in d, che deve essere abbastanza grande
strncpy(d, s, n)	copia al massimo n caratteri (vedi sotto)
strcat(d, s)	accoda s a d
strcmp(a, b)	negativo, zero o positivo secondo l'ordine
strchr(s, c)	puntatore alla prima occorrenza di c, o NULL
strrchr(s, c)	puntatore all'ultima occorrenza di c, o NULL
strstr(s, t)	puntatore alla prima occorrenza di t in s, o NULL
strcspn(s, t)	lunghezza del tratto iniziale privo di caratteri di t

Una stringa è un array di char terminato dal carattere '\0': l'array deve avere **un elemento in più** rispetto alla lunghezza massima prevista. strncpy non aggiunge il terminatore se la sorgente è lunga n caratteri o più; quando la si usa si termina la stringa a mano. Due stringhe non si confrontano con ==, che confronterebbe gli indirizzi, ma con strcmp.

Controllare l'estensione di un file

Molti programmi ricevono nomi di file da controllare: l'estensione è la parte che segue l'**ultimo** punto. strrchr restituisce un puntatore a quel punto, e strcmp confronta la parte finale con l'estensione attesa.

```
#include <string.h>

int haEstensione(const char *nome, const char *est)
{
    const char *punto = strrchr(nome, '.');
    return punto != NULL && strcmp(punto, est) == 0;
}
```

haEstensione("dati.txt", ".txt") restituisce 1; haEstensione("dati.txt.bak", ".txt") restituisce 0, perché l'ultimo punto precede «bak»; haEstensione("dati", ".txt") restituisce 0 senza accedere a un puntatore nullo, grazie al cortocircuito di &&.

Scomporre una riga in parole

`strtok(s, sep)` divide una stringa in **token** separati da uno qualunque dei caratteri di `sep`. Alla prima chiamata si passa la stringa, alle successive NULL, per continuare dallo stesso punto; quando non ci sono più token restituisce NULL. La funzione **modifica** la stringa, sostituendo i separatori con terminatori, quindi non si applica a una stringa costante.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char riga[] = "rosso, verde;blu  giallo";
    int n = 0;
    for (char *t = strtok(riga, " ,;"); t != NULL; t = strtok(NULL, " ,;"))
        printf("%d: %s\n", ++n, t);      /* stampa le quattro parole */
    return 0;
}
```

Trasformare i caratteri

Le funzioni di `ctype.h` classificano e convertono i singoli caratteri: `isalpha`, `isdigit`, `isspace`, `isupper`, `islower`, `toupper`, `tolower`. La funzione seguente porta una stringa in maiuscolo e conta le vocali.

```
#include <ctype.h>
#include <string.h>

int maiuscoloEVocali(char *s)
{
    int vocali = 0;
    for (int i = 0; s[i] != '\0'; i++) {
        s[i] = toupper((unsigned char)s[i]);
        if (strchr("AEIOU", s[i]) != NULL)
            vocali++;
    }
    return vocali;
}
```

Il cast a `unsigned char` evita un comportamento indefinito con i caratteri di codice superiore a 127. La condizione `s[i] != '\0'` evita di chiamare `strlen` a ogni iterazione.

Da stringa a numero

I parametri della riga di comando e le righe lette con `fgets` sono stringhe: per usarli come numeri vanno convertiti. `atoi` e `atof` sono semplici ma non segnalano errori, e restituiscono 0 sia per «0» sia per «abc». `strtol` permette invece di controllare che la stringa fosse davvero un numero:

```
#include <stdlib.h>

/* converte s in intero; restituisce 1 se s era un intero valido */
int convertiIntero(const char *s, int *valore)
{
    char *fine;
    long x = strtol(s, &fine, 10);          /* base 10 */
    if (fine == s || *fine != '\0')        /* nessuna cifra o caratteri in più */
        return 0;
    *valore = (int)x;
    return 1;
}
```

Dopo la chiamata, `fine` indica il primo carattere non convertito: se coincide con l'inizio la stringa non conteneva cifre, se non è il terminatore la stringa conteneva altri caratteri dopo il numero. Il procedimento inverso, da numero a stringa, si ottiene con `sprintf(s, "%d", x)`, o meglio con `snprintf`, che non supera la dimensione indicata.

DA RICORDARE

Come si costruisce il nome di un file aggiungendo un'estensione?

Si dichiara un array abbastanza grande da contenere nome, estensione e terminatore e lo si riempie con `snprintf(nome, sizeof nome, "%s.txt", base)`, oppure con `strcpy` seguita da `strcat`. Prima va controllato che la base non superi la lunghezza massima prevista, altrimenti la copia uscirebbe dai limiti dell'array.

Memoria dinamica in pratica

L'allocazione di array e matrici di dimensione nota solo durante l'esecuzione, il ridimensionamento con `realloc` e gli errori da evitare nella gestione dell'heap.

Array dinamici

Quando la dimensione di un array si conosce solo durante l'esecuzione, per esempio perché è scritta all'inizio di un file, l'array si alloca nell'heap con `malloc`. Lo schema è sempre lo stesso: calcolare i byte con `sizeof`, controllare che il risultato non sia `NULL`, usare la memoria come un normale array, liberarla con `free` quando non serve più.

```
int *v = malloc(n * sizeof(int));
if (v == NULL) {
    fprintf(stderr, "Memoria insufficiente\n");
    exit(EXIT_FAILURE);
}
for (int i = 0; i < n; i++)
    v[i] = 0;
/* ... uso di v ... */
free(v);
```

`calloc(n, sizeof(int))` alloca lo stesso spazio e lo **azzerà**; `malloc` lascia invece valori indeterminati.

Crescere con `realloc`

Se il numero di elementi non è noto neppure all'inizio, come quando si leggono numeri fino alla fine dell'input, si parte da una capacità piccola e la si **raddoppia** con `realloc` ogni volta che si esaurisce. Raddoppiare, invece di aggiungere un elemento alla volta, rende trascurabile il costo delle copie.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int capacita = 4, n = 0, x;
    int *v = malloc(capacita * sizeof(int));
    if (v == NULL)
        return 1;
    while (scanf("%d", &x) == 1) {
        if (n == capacita) {
            /* spazio esaurito */
            capacita *= 2;
            int *nuovo = realloc(v, capacita * sizeof(int));
            if (nuovo == NULL) {
                /* v è ancora valido */
                free(v);
                return 1;
            }
            v = nuovo;
        }
        v[n++] = x;
    }
    printf("Letti %d numeri, capacità finale %d\n", n, capacita);
    free(v);
    return 0;
}

```

```

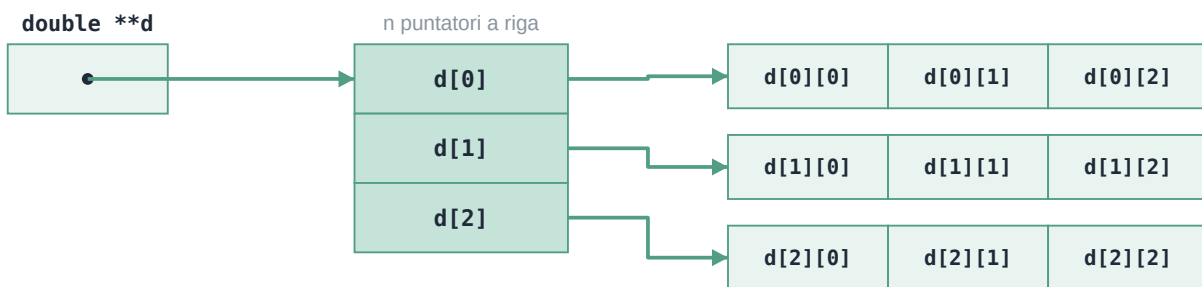
}

```

Il risultato di `realloc` si assegna a un puntatore di appoggio: se la funzione fallisce restituisce `NULL` e lascia intatto il blocco originale, che con `v = realloc(v, ...)` andrebbe perso senza poter essere liberato.

Matrici dinamiche

Esistono tre modi di allocare una matrice di R righe e C colonne note solo in esecuzione.



Matrice dinamica con un array di puntatori a riga: ogni riga è un blocco separato nell'heap.

PRIMO METODO: ARRAY DI PUNTATORI A RIGHE

Si alloca un array di R puntatori e poi una riga di C elementi per ciascuno. L'accesso è il consueto `m[i][j]`, ma la liberazione richiede un ciclo: prima ogni riga, poi l'array di puntatori.

```

int **creaMatrice(int r, int c)
{
    int **m = malloc(r * sizeof(int *));
    if (m == NULL)
        return NULL;
    for (int i = 0; i < r; i++) {
        m[i] = malloc(c * sizeof(int));
        if (m[i] == NULL) {
            /* libera quanto già allocato */
            for (int k = 0; k < i; k++)
                free(m[k]);
            free(m);
            return NULL;
        }
    }
    return m;
}

```

```

void liberaMatrice(int **m, int r)
{
    for (int i = 0; i < r; i++)
        free(m[i]);
    free(m);
}

```

SECONDO METODO: UN UNICO BLOCCO CON INDICE CALCOLATO

Si alloca un solo blocco di $R \times C$ elementi e si calcola a mano la posizione dell'elemento (i, j) , che nella memorizzazione per righe è $i * C + j$. Una sola malloc, una sola free e la matrice è contigua, quindi può essere scritta o letta da un file binario con un'unica chiamata.

```

int *m = malloc(r * c * sizeof(int));
m[i * c + j] = 5;
free(m);
/* elemento di riga i e colonna j */

```

TERZO METODO: BLOCCO UNICO CON PUNTATORI A RIGHE

Si combinano i due precedenti: un blocco contiguo per i dati e un array di puntatori che indicano l'inizio di ogni riga. Si scrive $m[i][j]$, i dati sono contigui e bastano due free.

```

int **m = malloc(r * sizeof(int *));
int *dati = malloc(r * c * sizeof(int));
for (int i = 0; i < r; i++)
    m[i] = dati + i * c;
/* ... uso di m[i][j] ... */
/* inizio della riga i */
free(m[0]);
free(m);
/* coincide con dati */

```

ATTENZIONE

Gli errori nella gestione dell'heap

Perdita di memoria (memory leak): un blocco non più raggiungibile e mai liberato, per esempio perché il puntatore è stato sovrascritto. **Puntatore pendente** (dangling pointer): un puntatore che indica memoria già liberata; usarlo produce risultati imprevedibili, ed è prudente assegnargli NULL dopo la free. **Doppia liberazione**: chiamare free due volte sullo stesso blocco. **Liberazione impropria**: chiamare free su un indirizzo non ottenuto da malloc, calloc o realloc, come quello di un array locale. L'opzione `-fsanitize=address` rileva tutti questi casi.

DA RICORDARE

Quali sono i vantaggi e gli svantaggi dei tre metodi per le matrici dinamiche?

L'array di puntatori a righe permette di scrivere `m[i][j]` e di avere righe di lunghezza diversa, ma richiede $R + 1$ allocazioni e un ciclo di liberazione, e le righe non sono contigue. Il blocco unico richiede di calcolare l'indice $i * C + j$, ma usa una sola allocazione ed è contiguo. Il terzo metodo unisce la notazione `m[i][j]` alla contiguità dei dati, con due sole allocazioni.

Parametri da riga di comando

Come un programma riceve i parametri con `argc` e `argv`, come li controlla e li raccoglie in una struttura, e come segnala gli errori con `stderr` e i codici di uscita.

argc e argv

Quando `main` è dichiarata come `int main(int argc, char *argv[])`, riceve il **numero di parametri** in `argc` e le **stringhe** dei parametri in `argv`. `argv[0]` è il nome con cui il programma è stato lanciato, quindi `argc` vale sempre almeno 1; i parametri veri vanno da `argv[1]` ad `argv[argc - 1]`, e `argv[argc]` vale `NULL`.

COMANDO	ARGC	ARGV
<code>./prog</code>	1	<code>./prog</code>
<code>./prog dati.txt</code>	2	<code>./prog, dati.txt</code>
<code>./prog dati.txt 10 out.txt</code>	4	<code>./prog, dati.txt, 10, out.txt</code>

Tutti i parametri sono **stringhe**: il 10 dell'ultima riga è la stringa "10", che va convertita in intero con `atoi` o, con controllo, con `strtol`.

Raccogliere i parametri in una struttura

Uno schema che rende ordinato il programma consiste nel definire una struttura con un campo per ogni parametro e una funzione `leggiParametri` che controlla tutto e la restituisce già riempita. Se un controllo fallisce, la funzione stampa un messaggio d'uso su `stderr` e termina il programma con un codice di errore.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNOME 20

typedef struct {
    char input[MAXNOME + 1];
    int n;
    char output[MAXNOME + 1];
} Parametri;

void errore(const char *prog)
{
    fprintf(stderr, "Uso: %s input.txt n output.txt (n da 1 a 100)\n", prog);
    exit(EXIT_FAILURE);
}
```

```

Parametri leggiParametri(int argc, char *argv[])
{
    Parametri p;
    if (argc != 4)
        errore(argv[0]);
    if (strlen(argv[1]) > MAXNOME || strlen(argv[3]) > MAXNOME)
        errore(argv[0]);
    char *fine;
    long n = strtol(argv[2], &fine, 10);
    if (*fine != '\0' || n < 1 || n > 100)
        errore(argv[0]);
    strcpy(p.input, argv[1]);
    p.n = (int)n;
    strcpy(p.output, argv[3]);
    return p;
}

```

```

int main(int argc, char *argv[])
{
    Parametri p = leggiParametri(argc, argv);
    printf("input = %s, n = %d, output = %s\n", p.input, p.n, p.output);
    return EXIT_SUCCESS;
}

```

Le lunghezze vanno controllate **prima** di `strcpy`, perché i campi della struttura hanno dimensione fissa. Il messaggio d'uso dice all'utente come lanciare correttamente il programma.

stderr e codici di uscita

I messaggi di errore si scrivono sul canale **stderr** con `fprintf(stderr, ...)` e non su `stdout`: così restano visibili sul terminale anche quando l'output normale è ridiretto su un file. Il valore restituito da `main`, o passato a `exit`, è il **codice di uscita** del programma: 0, cioè `EXIT_SUCCESS`, indica successo, un valore diverso da 0, come `EXIT_FAILURE`, indica un errore. Il terminale lo mostra con `echo $?` subito dopo l'esecuzione, e gli script lo usano per decidere come proseguire.

DA RICORDARE

Che cosa deve controllare una funzione che legge i parametri?

Il numero dei parametri (`argc`); la lunghezza delle stringhe che verranno copiate in array di dimensione fissa; il formato dei valori, per esempio che un numero sia davvero un numero e che rientri nell'intervallo ammesso; le estensioni dei nomi di file, se richieste. A ogni errore stampa su `stderr` un messaggio d'uso e termina con un codice diverso da 0.

Lettura e scrittura di file

Gli schemi pronti per leggere e scrivere file di testo e binari, per l'accesso diretto e per il controllo degli errori, con l'uso corretto del valore restituito dalle funzioni di lettura.

Aprire e chiudere

Ogni file si apre con `fopen`, che restituisce un puntatore a `FILE` oppure `NULL` se l'apertura fallisce (file inesistente, permessi mancanti, cartella sbagliata). Il controllo è obbligatorio, e al termine il file si chiude con `fclose`, che scrive su disco i dati ancora nel buffer.

MODO	SIGNIFICATO
"r"	lettura; il file deve esistere
"w"	scrittura; crea il file o ne cancella il contenuto
"a"	aggiunta in fondo; crea il file se non esiste
"r+"	lettura e scrittura; il file deve esistere
"rb", "wb"	come sopra, in modalità binaria

```
FILE *f = fopen(nome, "r");
if (f == NULL) {
    fprintf(stderr, "Impossibile aprire %s\n", nome);
    exit(EXIT_FAILURE);
}
/* ... lettura ... */
fclose(f);
```

Il ciclo di lettura corretto

La regola fondamentale è che un ciclo di lettura si controlla con il **valore restituito dalla funzione che legge**, non con `feof`. `feof` diventa vera solo **dopo** che una lettura ha tentato di andare oltre la fine; un ciclo `while (!feof(f))` esegue quindi un'iterazione di troppo ed elabora l'ultimo dato due volte, oppure un dato non letto.

FUNZIONE	IL CICLO CONTINUA FINCHÉ
<code>fscanf(f, "%d", &x)</code>	restituisce il numero di campi attesi (qui 1)
<code>fgets(s, n, f)</code>	restituisce un valore diverso da <code>NULL</code>
<code>fgetc(f)</code>	restituisce un valore diverso da <code>EOF</code> (in un <code>int</code>)
<code>fread(&x, sizeof x, 1, f)</code>	restituisce il numero di elementi richiesti

FILE DI TESTO CON CAMPI

Le righe composte da più campi si leggono comodamente con `fgets` e si scompongono con `sscanf`, che lavora su una stringa come `fscanf` lavora su un file. In questo modo una riga malformata si può riconoscere e saltare senza perdere l'allineamento con le righe successive.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *f = fopen("studenti.txt", "r");    /* righe del tipo: Rossi 27 */
    if (f == NULL) {
        fprintf(stderr, "Impossibile aprire studenti.txt\n");
        return EXIT_FAILURE;
    }
    char riga[100], cognome[31];
    int voto, letti = 0;
    while (fgets(riga, sizeof riga, f) != NULL) {
        if (sscanf(riga, "%30s %d", cognome, &voto) != 2)
            continue;                      /* riga non valida: si salta */
        printf("%-15s %2d\n", cognome, voto);
        letti++;
    }
    fclose(f);
    printf("%d righe valide\n", letti);
    return EXIT_SUCCESS;
}
```

La larghezza in `%30s` impedisce di scrivere oltre la fine di cognome, che ha 31 elementi per lasciare posto al terminatore.

File binari

Un file binario contiene i byte della rappresentazione in memoria dei dati, senza conversione in caratteri. Si scrive con `fwrite` e si legge con `fread`, che ricevono l'indirizzo dei dati, la dimensione di un elemento, il numero di elementi e il file, e restituiscono il numero di elementi effettivamente trasferiti.

```

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int dati[5] = {10, 20, 30, 40, 50}, letti[5];
    FILE *f = fopen("numeri.bin", "wb");
    if (f == NULL)
        return EXIT_FAILURE;
    fwrite(dati, sizeof(int), 5, f);          /* un'unica scrittura */
    fclose(f);

    f = fopen("numeri.bin", "rb");
    if (f == NULL)
        return EXIT_FAILURE;
    size_t n = fread(letti, sizeof(int), 5, f);
    fclose(f);
    for (size_t i = 0; i < n; i++)
        printf("%d ", letti[i]);
    printf("\n");
    return EXIT_SUCCESS;
}

```

Il file prodotto occupa 20 byte (5 interi da 4 byte) e non è leggibile con un editor di testo. Un file binario va riletto sulla stessa architettura, perché dimensione dei tipi e ordine dei byte possono cambiare da una macchina all'altra.

DA RICORDARE

Che differenza c'è tra un file di testo e un file binario?

Un file di testo contiene caratteri: i numeri sono scritti come sequenze di cifre e la loro lunghezza varia, si legge con un editor e si elabora con `fprintf`, `fscanf` e `fgets`. Un file binario contiene la rappresentazione in memoria dei dati: ogni intero occupa sempre `sizeof(int)` byte, si elabora con `fwrite` e `fread`, è più compatto e veloce da leggere e permette l'accesso diretto a un record calcolandone la posizione.

Accesso diretto

Con `fseek(f, spostamento, origine)` ci si posiziona in un punto qualsiasi del file, contando i byte a partire dall'inizio (`SEEK_SET`), dalla posizione corrente (`SEEK_CUR`) o dalla fine (`SEEK_END`). `ftell(f)` restituisce la posizione corrente e `rewind(f)` torna all'inizio. In un file binario di record della stessa dimensione, il record di indice `k` inizia al byte `k * sizeof(record)`, e si può leggere o riscrivere senza toccare gli altri.

ESEMPIO SVOLTO

Modificare un record sul posto

Il file `numeri.bin` dell'esempio precedente contiene 5 interi. Si raddoppia il terzo (indice 2) e si calcola il numero di interi contenuti nel file dalla sua dimensione.

```
FILE *f = fopen("numeri.bin", "r+b");
int x;
fseek(f, 2 * sizeof(int), SEEK_SET);      /* inizio del terzo intero */
fread(&x, sizeof(int), 1, f);
x *= 2;
fseek(f, 2 * sizeof(int), SEEK_SET);      /* si torna indietro di un intero */
fwrite(&x, sizeof(int), 1, f);
fseek(f, 0, SEEK_END);
long quanti = ftell(f) / sizeof(int);      /* dimensione in byte / 4 */
fclose(f);
```

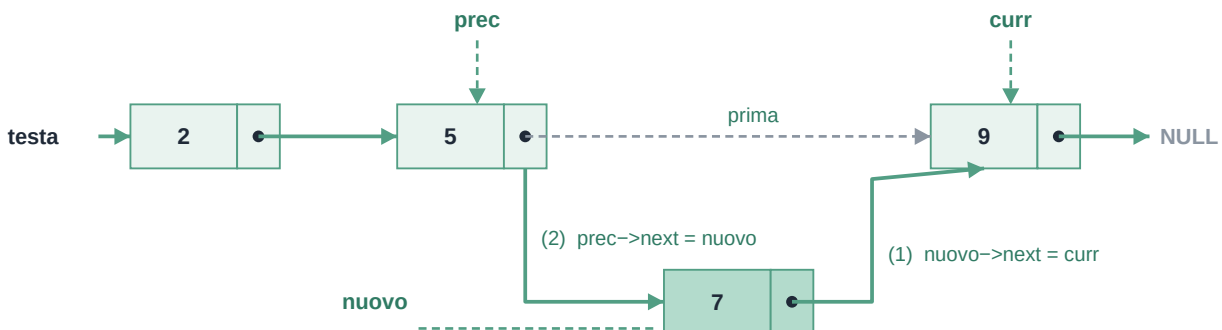
Il secondo `fseek` è necessario: dopo la lettura la posizione è avanzata di un intero, e una scrittura immediata modificherebbe il quarto. Lo standard richiede inoltre un posizionamento tra una lettura e una scrittura sullo stesso file.

Liste, pile e code: implementazioni complete

Le implementazioni complete delle strutture collegate: inserimento in testa, in coda e in ordine, cancellazione, pila e coda, provate con un programma di esempio.

La lista concatenata

Ogni nodo contiene un dato e il puntatore al nodo successivo; la lista è rappresentata dal puntatore al primo nodo, che vale NULL se la lista è vuota. Le funzioni che possono cambiare il primo nodo ricevono un **puntatore al puntatore di testa**, così da poterlo modificare.



Inserimento del valore 7 tra i nodi 5 e 9 di una lista ordinata: è il caso gestito da `inserisciOrdinato`.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Nodo {
    int val;
    struct Nodo *next;
} Nodo;

Nodo *nuovoNodo(int x)
{
    Nodo *n = malloc(sizeof(Nodo));
    if (n == NULL) {
        fprintf(stderr, "Memoria insufficiente\n");
        exit(EXIT_FAILURE);
    }
    n->val = x;
    n->next = NULL;
    return n;
}
```

```

void inserisciTesta(Nodo **testa, int x)
{
    Nodo *n = nuovoNodo(x);
    n->next = *testa;          /* il nuovo nodo precede il vecchio primo */
    *testa = n;
}

void inserisciCoda(Nodo **testa, int x)
{
    Nodo *n = nuovoNodo(x);
    if (*testa == NULL) {
        *testa = n;
        return;
    }
    Nodo *p = *testa;
    while (p->next != NULL)    /* si arriva all'ultimo nodo */
        p = p->next;
    p->next = n;
}

```

```

void inserisciOrdinato(Nodo **testa, int x)
{
    Nodo **pp = testa;        /* indirizzo del puntatore da modificare */
    while (*pp != NULL && (*pp)->val < x)
        pp = &(*pp)->next;
    Nodo *n = nuovoNodo(x);
    n->next = *pp;
    *pp = n;
}

int elimina(Nodo **testa, int x)    /* 1 se trovato e cancellato */
{
    Nodo **pp = testa;
    while (*pp != NULL && (*pp)->val != x)
        pp = &(*pp)->next;
    if (*pp == NULL)
        return 0;
    Nodo *da = *pp;
    *pp = da->next;            /* scavalca il nodo */
    free(da);
    return 1;
}

```

```

void stampa(const Nodo *p)
{
    for (; p != NULL; p = p->next)
        printf("%d ", p->val);
    printf("\n");
}

void libera(Nodo *p)
{
    while (p != NULL) {
        Nodo *succ = p->next;      /* salvato prima della free */
        free(p);
        p = succ;
    }
}

```

```

int main(void)
{
    Nodo *l = NULL;
    int dati[6] = {7, 3, 9, 1, 5, 3};
    for (int i = 0; i < 6; i++)
        inserisciOrdinato(&l, dati[i]);
    stampa(l);                      /* 1 3 3 5 7 9 */
    elimina(&l, 3);
    elimina(&l, 9);
    inserisciTesta(&l, 0);
    inserisciCoda(&l, 10);
    stampa(l);                      /* 0 1 3 5 7 10 */
    libera(l);
    return 0;
}

```

Le funzioni `inserisciOrdinato` ed `elimina` usano un **puntatore al puntatore** `pp`, che indica il campo da modificare: all'inizio il puntatore di testa, poi il campo `next` del nodo corrente. In questo modo il caso del primo nodo non richiede un trattamento a parte. In `libera` il successore va salvato prima di liberare il nodo, perché dopo la `free` il campo `next` non è più accessibile.

ATTENZIONE

Il caso della struttura vuota

Quasi tutti gli errori nelle strutture collegate riguardano i casi limite: lista vuota, lista di un solo elemento, inserimento o cancellazione del primo o dell'ultimo nodo. In una coda, quando si estrae l'ultimo elemento, va riportato a `NULL` anche il puntatore al fondo; dimenticarlo lascia un puntatore pendente, che il successivo inserimento userebbe.

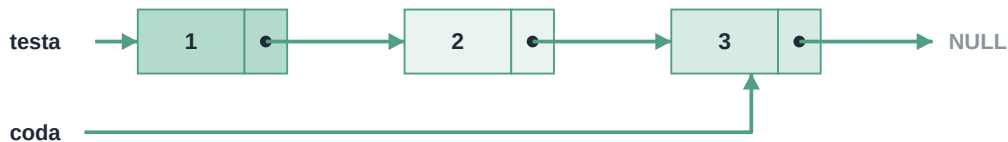
Pila e coda

La **pila** (stack) estrae per primo l'ultimo elemento inserito (LIFO): si realizza con una lista in cui inserimento ed estrazione avvengono entrambi in testa. La **coda** (queue) estrae per primo il primo inserito (FIFO): si inserisce in fondo e si estrae in testa, e per inserire in tempo costante si tiene anche un puntatore all'ultimo nodo.

PILA (LIFO): PUSH E POP IN CIMA



CODA (FIFO): ENQUEUE IN CODA, DEQUEUE IN TESTA



Pila e coda realizzate con liste concatenate dopo l'inserimento di 1, 2, 3: nella coda si mantengono due puntatori.

```
typedef struct {
    Nodo *testa, *fondo;
} Coda;

void accoda(Coda *q, int x)
{
    Nodo *n = nuovoNodo(x);
    if (q->fondo == NULL)
        q->testa = n;
    else
        q->fondo->next = n;
    q->fondo = n;
}

int estrai(Coda *q, int *x)
{
    if (q->testa == NULL)
        return 0;
    Nodo *primo = q->testa;
    *x = primo->val;
    q->testa = primo->next;
    if (q->testa == NULL)
        q->fondo = NULL;
    free(primo);
}

return 1;
}
```

Per la pila bastano `inserisciTesta` come operazione push e, come pop, una funzione identica a `estrai` ma senza il puntatore al fondo. Un'applicazione tipica della pila è la verifica del bilanciamento delle parentesi: si inserisce ogni parentesi aperta e, a ogni parentesi chiusa, si estrae e si controlla che sia del tipo corrispondente.

Ordinare e cercare in pratica

Gli algoritmi di ordinamento e di ricerca scritti come funzioni riusabili, applicati ad array di interi, di stringhe e di strutture, e l'uso della funzione di libreria `qsort`.

Ordinamento per selezione e a bolle

Le due funzioni seguenti ordinano un array di interi in modo crescente. Entrambe richiedono un numero di confronti proporzionale a n^2 ; il bubble sort con flag si ferma però appena una passata non esegue scambi, quindi su un array già ordinato fa una sola passata.

```
void scambia(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void selectionSort(int v[], int n)
{
    for (int i = 0; i < n - 1; i++) {
        int imin = i;                                /* minimo della parte non ordinata */
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[imin])
                imin = j;
        scambia(&v[i], &v[imin]);
    }
}
```

```
void bubbleSort(int v[], int n)
{
    int scambiato = 1;
    for (int passo = 1; passo < n && scambiato; passo++) {
        scambiato = 0;
        for (int i = 0; i < n - passo; i++)          /* gli ultimi passo-1 sono a posto */
            if (v[i] > v[i + 1]) {
                scambia(&v[i], &v[i + 1]);
                scambiato = 1;
            }
    }
}
```

Il limite $i < n - \text{passo}$ del ciclo interno non è solo un'ottimizzazione: con $i < n$ l'accesso a $v[i + 1]$ uscirebbe dall'array all'ultima iterazione.

Insertion sort

L'insertion sort mantiene ordinata la parte iniziale dell'array e vi inserisce un elemento alla volta, spostando a destra gli elementi maggiori fino a trovare la posizione giusta. È efficiente su array quasi ordinati.

```
void insertionSort(int v[], int n)
{
    for (int i = 1; i < n; i++) {
        int x = v[i], j = i - 1;
        while (j >= 0 && v[j] > x) {          /* sposta a destra i maggiori */
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = x;
    }
}
```

Ordinare stringhe e strutture

Lo stesso algoritmo ordina dati di qualunque tipo: cambiano soltanto il **confronto** e lo **scambio**. Per le stringhe il confronto si fa con `strcmp`; se le stringhe sono raccolte in un array di puntatori, `char *v[]`, conviene scambiare i **puntatori** invece di copiare i caratteri. Per le strutture si confronta il campo scelto come chiave e si scambiano le strutture intere, oppure i puntatori a esse.

```
typedef struct {
    char cognome[31];
    int voto;
} Studente;

/* ordina per voto decrescente; a parità di voto, per cognome */
void ordinaStudenti(Studente s[], int n)
{
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - 1 - i; j++) {
            int fuoriOrdine = s[j].voto < s[j + 1].voto
                || (s[j].voto == s[j + 1].voto
                    && strcmp(s[j].cognome, s[j + 1].cognome) > 0);
            if (fuoriOrdine) {
                Studente t = s[j];          /* le strutture si copiano con = */
                s[j] = s[j + 1];
                s[j + 1] = t;
            }
        }
}
```

La funzione qsort

La libreria standard offre `qsort`, che ordina un array di qualunque tipo. Riceve l'array, il numero di elementi, la dimensione di un elemento e una **funzione di confronto**, che riceve due puntatori generici agli elementi e restituisce un valore negativo, zero o positivo.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int confrontaInt(const void *a, const void *b)
{
    int x = *(const int *)a, y = *(const int *)b;
    return (x > y) - (x < y);          /* evita l'overflow di x - y */
}

int confrontaStringhe(const void *a, const void *b)
{
    /* gli elementi sono char *, quindi a e b sono puntatori a char * */
    return strcmp(*(char *const *)a, *(char *const *)b);
}

```

```

int main(void)
{
    int v[6] = {5, -2, 9, 0, 3, -7};
    char *nomi[4] = {"Marco", "Anna", "Luca", "Bianca"};
    qsort(v, 6, sizeof(int), confrontaInt);
    qsort(nomi, 4, sizeof(char *), confrontaStringhe);
    for (int i = 0; i < 6; i++)
        printf("%d ", v[i]);
    printf("\n");
    for (int i = 0; i < 4; i++)
        printf("%s ", nomi[i]);
    printf("\n");
    return 0;
}

```

Ricerca sequenziale e dicotomica

La ricerca sequenziale funziona su qualunque array e richiede in media $n/2$ confronti. La ricerca dicotomica richiede un array **ordinato** e al massimo circa $\log_2 n$ confronti: con un milione di elementi, una ventina.

```

int ricercaSequenziale(const int v[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (v[i] == x)
            return i;
    return -1;                                /* non trovato */
}

int ricercaDicotomica(const int v[], int n, int x)
{
    int inizio = 0, fine = n - 1;
    while (inizio <= fine) {
        int medio = inizio + (fine - inizio) / 2;
        if (v[medio] == x)
            return medio;
        if (v[medio] < x)
            inizio = medio + 1;                /* si scarta la metà sinistra */
        else
            fine = medio - 1;                  /* si scarta la metà destra */
    }
    return -1;
}

```

Il valore restituito in caso di insuccesso è meno 1, che non può essere un indice valido. Il punto medio si calcola come $\text{inizio} + (\text{fine} - \text{inizio}) / 2$ invece di $(\text{inizio} + \text{fine}) / 2$: il risultato è lo stesso, ma la somma di due indici molto grandi potrebbe superare il massimo rappresentabile in un int.

DA RICORDARE

Come si sceglie l'algoritmo di ordinamento negli esercizi?

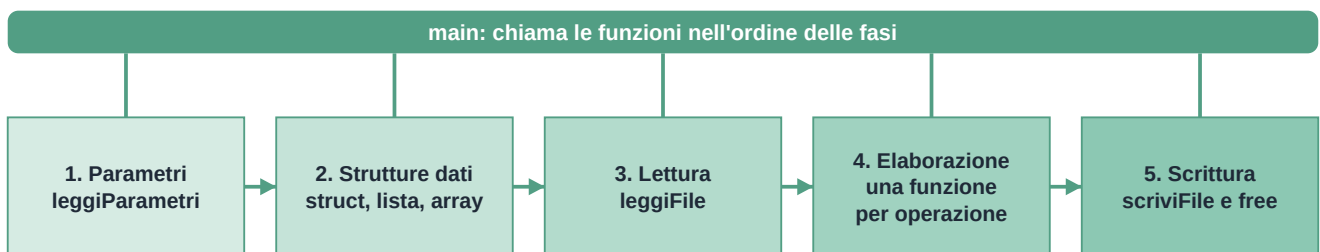
Per pochi elementi qualunque algoritmo semplice va bene; il bubble sort con flag e l'insertion sort sono rapidi su dati quasi ordinati. Se l'ordinamento deve essere **stabile**, cioè conservare l'ordine originale degli elementi con chiave uguale, vanno bene bubble sort e insertion sort con confronto stretto, mentre il selection sort non lo garantisce. Quando non è richiesto un algoritmo specifico, `qsort` evita di riscrivere il codice, ma non garantisce la stabilità.

Metodo per un progetto completo

Il procedimento in cinque fasi con cui si affronta un programma completo: dai parametri alla lettura dei dati, all'elaborazione e alla scrittura dei risultati, con le regole di stile che lo rendono leggibile e corretto.

Le cinque fasi

Un esercizio completo di programmazione in C ha quasi sempre la stessa struttura: riceve parametri dalla riga di comando, legge dati da un file, li memorizza in una struttura opportuna, li elabora e scrive i risultati a video o in un altro file. Conviene affrontarlo in cinque fasi, ciascuna realizzata da una o più funzioni e **provata prima di passare alla successiva**.



ogni fase si prova prima di scrivere la successiva

Le cinque fasi di un programma completo, ciascuna in una o più funzioni chiamate da main.

- 1. Parametri.** Definire la struttura `Parametri` e la funzione `leggiParametri`, che controlla numero, lunghezza e formato dei parametri e termina con un messaggio d'uso in caso di errore. Si prova con parametri giusti e sbagliati e si stampa la struttura.
- 2. Strutture dati.** Definire i tipi dei record (struct) e scegliere dove memorizzarli: array statico se la dimensione massima è nota, array dinamico se è scritta nel file, lista se i dati vanno mantenuti ordinati o se ne vanno estratti alcuni.
- 3. Lettura.** Scrivere la funzione che apre il file, legge i dati controllando il valore restituito dalla funzione di lettura, li inserisce nella struttura e chiude il file. Si prova stampando il contenuto della struttura.
- 4. Elaborazione.** Una funzione per ogni operazione richiesta: calcoli, ricerche, ordinamenti, estrazioni. Si prova confrontando l'output con i risultati calcolati a mano su un file di prova piccolo.
- 5. Scrittura e chiusura.** Scrivere i risultati nel formato richiesto, liberare tutta la memoria dinamica e restituire `EXIT_SUCCESS`.

Regole di stile

- **Nessuna variabile globale.** Ogni funzione riceve ciò che le serve come parametro; il flusso dei dati resta visibile nei prototipi.
- **main breve.** Il corpo di `main` si limita a chiamare le funzioni delle cinque fasi nell'ordine giusto, e si legge come un indice del programma.
- **Costanti con nome.** Le dimensioni massime si definiscono con `#define` all'inizio del file, così si cambiano in un solo punto.
- **Controlli sistematici.** Ogni `fopen` e ogni `malloc` è seguita dal controllo del risultato; ogni lettura è controllata con il valore restituito.

- **Errori su stderr.** I messaggi di errore vanno su stderr, il programma termina con un codice diverso da 0.
- **Memoria liberata.** Ogni blocco allocato viene liberato prima della fine, ed è buona prova eseguire con `-fsanitize=address`, che segnala ciò che resta allocato.
- **Commenti utili.** Un commento spiega il perché di una scelta, non ripete l'istruzione.

File di prova

Un file di prova va preparato **prima** di eseguire il programma, e deve essere abbastanza piccolo da poter calcolare a mano il risultato atteso. Oltre al caso normale conviene preparare i casi limite: file vuoto, un solo dato, dati tutti uguali, dati già ordinati o in ordine inverso, righe malformate. Un programma che supera solo il caso normale non è stato ancora verificato.

DA RICORDARE

Perché ogni fase va provata prima di scrivere la successiva?

Perché un errore resta confinato nelle poche righe appena scritte, dove è facile trovarlo. Se si scrive l'intero programma e lo si esegue solo alla fine, un risultato sbagliato può dipendere da parametri, lettura, struttura dati o elaborazione, e la ricerca della causa richiede molto più tempo. Stampare la struttura dati dopo la lettura è la verifica più utile di tutte.

Esempio svolto: matrice da file binario

Un programma completo che legge una matrice da un file binario, ne estrae un blocco quadrato, ne verifica una proprietà, ne cerca il minimo e scrive il risultato in un file di testo.

Il problema

IL PROBLEMA

Il programma riceve due parametri: il nome di un file binario con estensione «.raw» e il nome di un file di testo con estensione «.txt», ciascuno lungo al massimo 20 caratteri. Il file binario contiene due interi R e C , il numero di righe e di colonne, seguiti dagli $R \times C$ elementi interi della matrice A , memorizzati per righe.

Il programma deve: stampare A ; costruire la matrice quadrata B formata dalle ultime N righe e dalle ultime N colonne di A , con N uguale al minimo tra R e C ; stabilire se B è **triangolare superiore**, cioè se tutti gli elementi sotto la diagonale principale sono nulli; trovare il minimo di B e la posizione della sua prima occorrenza; scrivere B nel file di testo, riservando 7 caratteri a ogni valore.

Analisi e scelte

Le dimensioni della matrice si conoscono solo leggendo il file, quindi la matrice va allocata dinamicamente. Si sceglie il **blocco unico** con indice calcolato: un solo `fread` legge tutti gli elementi, e il calcolo $i * \text{colonne} + j$ si concentra in poche funzioni. Una struttura `Matrice` raccoglie dimensioni e puntatore ai dati, così ogni funzione riceve un solo parametro per matrice.

- `leggiParametri` controlla numero, lunghezza ed estensione dei parametri.
- `leggiMatrice` legge le dimensioni, alloca il blocco e legge gli elementi, controllando che il file contenga davvero $R \times C$ interi.
- `estraiTriangolare` copia il blocco in basso a destra, che inizia alla riga R meno N e alla colonna C meno N , e controlla gli elementi con indice di riga maggiore dell'indice di colonna.
- `minimo` restituisce il valore minimo e scrive riga e colonna attraverso due puntatori; il confronto stretto conserva la prima occorrenza nella scansione per righe.
- `scriviMatrice` scrive B con il formato `%7d`.

Il file di prova

Un file binario non si può scrivere con un editor di testo: lo si produce con un piccolo programma, che scrive le dimensioni e poi i valori.

```

/* creaRaw.c: scrive il file binario di prova dati.raw (3 righe, 5 colonne) */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int dim[2] = {3, 5};
    int valori[15] = {9, 8, 4, 2, 7,
                     5, 6, 0, -1, 3,
                     4, 1, 0, 0, 10};
    FILE *f = fopen("dati.raw", "wb");
    if (f == NULL)
        return EXIT_FAILURE;
    fwrite(dim, sizeof(int), 2, f);      /* prima righe e colonne */
    fwrite(valori, sizeof(int), 15, f);  /* poi i valori, per righe */
    fclose(f);
    return EXIT_SUCCESS;
}

```

Il file contiene la matrice 3×5 qui sotto. Con N uguale a 3 il blocco B è formato dalle colonne di indice da 2 a 4: sotto la sua diagonale ci sono solo zeri, quindi B è triangolare superiore, e il suo minimo è meno 1, in riga 1 e colonna 1.

A (3 x 5)	B (3 x 3)
9 8 4 2 7	4 2 7
5 6 0 -1 3	0 -1 3
4 1 0 0 10	0 0 10

Il programma

```

/* matrice.c: legge una matrice da un file binario, ne estrae il blocco quadrato
   in basso a destra, lo analizza e lo scrive in un file di testo */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNOME 20

typedef struct {
    char input[MAXNOME + 1];    /* file binario .raw */
    char output[MAXNOME + 1];  /* file di testo .txt */
} Parametri;

typedef struct {
    int righe, colonne;
    int *dati;                /* elemento (i, j) in dati[i * colonne + j] */
} Matrice;

```

```

/* 1 se il nome termina con l'estensione indicata */
int haEstensione(const char *nome, const char *est)
{
    const char *punto = strrchr(nome, '.');
    return punto != NULL && strcmp(punto, est) == 0;
}

Parametri leggiParametri(int argc, char *argv[])
{
    Parametri p;
    if (argc != 3) {
        fprintf(stderr, "Uso: %s dati.raw risultato.txt\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strlen(argv[1]) > MAXNOME || strlen(argv[2]) > MAXNOME) {
        fprintf(stderr, "Nomi dei file troppo lunghi\n");
        exit(EXIT_FAILURE);
    }
    if (!haEstensione(argv[1], ".raw") || !haEstensione(argv[2], ".txt")) {
        fprintf(stderr, "Estensioni richieste: .raw e .txt\n");
        exit(EXIT_FAILURE);
    }
    strcpy(p.input, argv[1]);
    strcpy(p.output, argv[2]);
}

```

```

    return p;
}

Matrice leggiMatrice(const char *nome)
{
    Matrice m;
    FILE *f = fopen(nome, "rb");
    if (f == NULL) {
        fprintf(stderr, "Impossibile aprire %s\n", nome);
        exit(EXIT_FAILURE);
    }
    /* i primi due interi sono righe e colonne */
    if (fread(&m.righe, sizeof(int), 1, f) != 1
        || fread(&m.colonne, sizeof(int), 1, f) != 1
        || m.righe <= 0 || m.colonne <= 0) {
        fprintf(stderr, "Dimensioni non valide\n");
        fclose(f);
        exit(EXIT_FAILURE);
    }
    int n = m.righe * m.colonne;
    m.dati = malloc(n * sizeof(int));
    if (m.dati == NULL) {
        fprintf(stderr, "Memoria insufficiente\n");
        fclose(f);
    }
}

```

```

        exit(EXIT_FAILURE);
    }
    if (fread(m.dati, sizeof(int), n, f) != (size_t) n) { /* lettura incompleta */
        fprintf(stderr, "File incompleto\n");
        free(m.dati);
        fclose(f);
        exit(EXIT_FAILURE);
    }
    fclose(f);
    return m;
}

void stampaMatrice(const Matrice *m)
{
    for (int i = 0; i < m->righe; i++) {
        for (int j = 0; j < m->colonne; j++)
            printf("%7d", m->dati[i * m->colonne + j]);
        printf("\n");
    }
}

```

```

/* estrae in b il blocco N x N in basso a destra di a;
   restituisce 1 se b è triangolare superiore */
int estraiTriangolare(const Matrice *a, Matrice *b)
{
    int n = a->righe < a->colonne ? a->righe : a->colonne; /* N = min(R, C) */
    b->righe = b->colonne = n;
    b->dati = malloc(n * n * sizeof(int));
    if (b->dati == NULL) {
        fprintf(stderr, "Memoria insufficiente\n");
        exit(EXIT_FAILURE);
    }
    int r0 = a->righe - n, c0 = a->colonne - n; /* angolo del blocco */
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            b->dati[i * n + j] = a->dati[(r0 + i) * a->colonne + (c0 + j)];
    int triangolare = 1;
    for (int i = 1; i < n; i++) /* sotto la diagonale: i > j */
        for (int j = 0; j < i; j++)
            if (b->dati[i * n + j] != 0)
                triangolare = 0;
    return triangolare;
}

```

```

/* minimo e posizione della sua prima occorrenza, scandendo per righe */
int minimo(const Matrice *m, int *riga, int *col)
{
    int min = m->dati[0];
    *riga = 0;
    *col = 0;
    for (int i = 0; i < m->righe; i++)
        for (int j = 0; j < m->colonne; j++)
            if (m->dati[i * m->colonne + j] < min) { /* resta la prima occorrenza */
                min = m->dati[i * m->colonne + j];
                *riga = i;
                *col = j;
            }
    return min;
}

```

```

void scriviMatrice(const char *nome, const Matrice *m)
{
    FILE *f = fopen(nome, "w");
    if (f == NULL) {
        fprintf(stderr, "Impossibile scrivere %s\n", nome);
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < m->righe; i++) {
        for (int j = 0; j < m->colonne; j++)
            /* 7 caratteri per valore */
            fprintf(f, "%7d", m->dati[i * m->colonne + j]);
        fprintf(f, "\n");
    }
    fclose(f);
}

```

```

int main(int argc, char *argv[])
{
    Parametri p = leggiParametri(argc, argv);
    printf("input = %s, output = %s\n", p.input, p.output);

    Matrice a = leggiMatrice(p.input);
    printf("R = %d, C = %d\n", a.righe, a.colonne);
    stampaMatrice(&a);

    Matrice b;
    int tri = estraiTriangolare(&a, &b);
    printf("Blocco B (%d x %d):\n", b.righe, b.colonne);
    stampaMatrice(&b);
    printf("B %s triangolare superiore.\n", tri ? "è" : "non è");

    int r, c;
    int min = minimo(&b, &r, &c);
    printf("Minimo in B: %d in posizione (%d, %d).\n", min, r, c);
}

```

```

scriviMatrice(p.output, &b);
printf("B scritta in %s.\n", p.output);

free(a.dati);
free(b.dati);
return EXIT_SUCCESS;
}

```

Esecuzione di prova

```

$ gcc -Wall -Wextra -std=c11 matrice.c -o matrice
$ ./matrice dati.raw risultato.txt
input = dati.raw, output = risultato.txt
R = 3, C = 5
    9    8    4    2    7
    5    6    0   -1    3
    4    1    0    0   10
Blocco B (3 x 3):
    4    2    7
    0   -1    3
    0    0   10
B è triangolare superiore.
Minimo in B: -1 in posizione (1, 1).
B scritta in risultato.txt.
$ ./matrice dati.raw risultato.dat
Estensioni richieste: .raw e .txt

```

L'output coincide con quanto calcolato a mano. Il secondo comando mostra il controllo dei parametri: il programma stampa il messaggio su stderr e termina con codice di uscita 1. Eseguito con `-fsanitize=address`, il programma non segnala accessi illegali né memoria non liberata.

ATTENZIONE

Il confronto con fread

`fread` restituisce un valore di tipo `size_t`, senza segno, mentre `n` è un `int`. Il cast `(size_t) n` nel confronto evita l'avviso di gcc sul confronto tra interi con e senza segno. Se il file è più corto del previsto, `fread` restituisce meno elementi di quelli richiesti e il programma lo segnala invece di elaborare valori mai letti.

Esempio svolto: archivio ordinato in una lista

Un programma completo che legge un archivio di record da un file di testo, lo memorizza in una lista ordinata, sposta in una seconda lista i record migliori secondo un indicatore calcolato e li salva in un file.

Il problema

IL PROBLEMA

Il programma riceve due parametri: i nomi del file di input e del file di output, **senza estensione** e lunghi al massimo 15 caratteri; il programma aggiunge a entrambi l'estensione «.txt» e verifica che il file di input esista. Ogni riga del file di input descrive un artigiano: nome e cognome (al massimo 10 caratteri ciascuno), età, numero di ordini evasi, numero di pezzi prodotti e turno (un carattere, P per pomeriggio o M per mattina).

Il programma deve: memorizzare i record in una lista A ordinata per età crescente e stamparla; calcolare per ciascuno la media di pezzi per ordine; spostare da A in una lista B, inserendoli **in testa**, i tre artigiani con la media più alta, uno alla volta; stampare A e B; scrivere B nel file di output, aggiungendo a ogni riga la media con due cifre decimali.

Analisi e scelte

I record hanno campi di tipo diverso, quindi si raccolgono in una struttura `Artigiano`, e il nodo della lista contiene una struttura completa. L'ordinamento per età si ottiene **inserendo ogni record nella posizione giusta** mano a mano che lo si legge, senza ordinare dopo.

- `leggiParametri` costruisce i nomi completi con `sprintf` e verifica l'esistenza del file aprendolo in lettura. Gli array sono lunghi $15 + 4 + 1$ caratteri: nome, estensione e terminatore.
- `media` restituisce un float; il cast prima della divisione evita la divisione intera.
- `leggiFile` legge una riga alla volta con `fgets` e la scompone con `sscanf`; le righe non valide, o con zero ordini, vengono saltate.
- `inserisciOrdinato` usa il confronto `<=`, così un artigiano con la stessa età di altri viene inserito **dopo** di loro e l'ordine del file è conservato tra i pari età.
- `estraiMax` scorre la lista ricordando il nodo con la media massima e il suo predecessore, poi lo **scollega** e lo restituisce: il nodo passa da A a B senza copie e senza nuove allocazioni.

Il programma

```
/* artigiani.c: legge un archivio di artigiani in una lista ordinata per età,
   sposta in una seconda lista i tre con la media di pezzi per ordine
   più alta e la salva in un file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNOME 15

typedef struct {
    char input[MAXNOME + 5];      /* nome + ".txt" + terminatore */
    char output[MAXNOME + 5];
} Parametri;

typedef struct {
    char nome[11], cognome[11];
    int eta, ordini, pezzi;
    char turno;
} Artigiano;

typedef struct Nodo {
    Artigiano dati;
    struct Nodo *next;
} Nodo;

Parametri leggiParametri(int argc, char *argv[])
{
    Parametri p;
    if (argc != 3 || strlen(argv[1]) > MAXNOME || strlen(argv[2]) > MAXNOME) {
        fprintf(stderr, "Uso: %s input output (senza estensione, max %d caratteri)\n",
            argv[0], MAXNOME);
        exit(EXIT_FAILURE);
    }
    sprintf(p.input, "%s.txt", argv[1]);      /* aggiunge l'estensione */
    sprintf(p.output, "%s.txt", argv[2]);
    FILE *f = fopen(p.input, "r");            /* il file di input deve esistere */
    if (f == NULL) {
        fprintf(stderr, "Il file %s non esiste\n", p.input);
        exit(EXIT_FAILURE);
    }
    fclose(f);
    return p;
}
```

```

float media(Artigiano a)
{
    return (float) a.pezzi / a.ordini;          /* cast: altrimenti divisione intera */
}

/* inserisce il nodo n in ordine di età crescente; a parità dopo gli uguali */
void inserisciOrdinato(Nodo **lista, Nodo *n)
{
    Nodo *prec = NULL, *curr = *lista;
    while (curr != NULL && curr->dati.eta <= n->dati.eta) {
        prec = curr;
        curr = curr->next;
    }
    n->next = curr;
    if (prec == NULL)
        *lista = n;
    else
        prec->next = n;
}

```

```

Nodo *leggiFile(const char *nome)
{
    FILE *f = fopen(nome, "r");
    if (f == NULL) {
        fprintf(stderr, "Impossibile aprire %s\n", nome);
        exit(EXIT_FAILURE);
    }
    Nodo *lista = NULL;
    char riga[256];
    while (fgets(riga, sizeof riga, f) != NULL) {
        Artigiano a;
        if (sscanf(riga, "%10s %10s %d %d %d %c", a.nome, a.cognome,
                    &a.eta, &a.ordini, &a.pezzi, &a.turno) != 6 || a.ordini <= 0)
            continue; /* riga non valida: si salta */
        Nodo *n = malloc(sizeof(Nodo));
        if (n == NULL) {
            fprintf(stderr, "Memoria insufficiente\n");
            exit(EXIT_FAILURE);
        }
        n->dati = a;
        inserisciOrdinato(&lista, n);
    }
    fclose(f);
    return lista;
}

```

```

}

void stampaArtigiano(Artigiano a)
{
    printf("%10s %10s %3d %4d %4d %c %5.2f\n", a.nome, a.cognome, a.eta,
        a.ordini, a.pezzi, a.turno, media(a));
}

void stampaLista(const Nodo *p)
{
    for (; p != NULL; p = p->next)
        stampaArtigiano(p->dati);
}

/* scollega dalla lista il nodo con la media più alta e lo restituisce */
Nodo *estraiMax(Nodo **lista)
{
    if (*lista == NULL)
        return NULL;
    Nodo *max = *lista, *precMax = NULL;
    for (Nodo *prec = *lista, *p = (*lista)->next; p != NULL; prec = p, p = p->next)
        if (media(p->dati) > media(max->dati)) {
            max = p;
            precMax = prec;

```

```

        }
    if (precMax == NULL)
        *lista = max->next;
    else
        precMax->next = max->next;
    max->next = NULL;
    return max;
}

void scriviFile(const char *nome, const Nodo *p)
{
    FILE *f = fopen(nome, "w");
    if (f == NULL) {
        fprintf(stderr, "Impossibile scrivere %s\n", nome);
        exit(EXIT_FAILURE);
    }
    for (; p != NULL; p = p->next)
        fprintf(f, "%s %s %d %d %d %c %5.2f\n", p->dati.nome, p->dati.cognome,
            p->dati.eta, p->dati.ordini, p->dati.pezzi, p->dati.turno,
            media(p->dati));
    fclose(f);
}

```

```

void libera(Nodo *p)
{
    while (p != NULL) {
        Nodo *succ = p->next;
        free(p);
        p = succ;
    }
}

int main(int argc, char *argv[])
{
    Parametri par = leggiParametri(argc, argv);
    printf("input = %s, output = %s\n", par.input, par.output);

    Nodo *a = leggiFile(par.input);
    printf("Lista A ordinata per età:\n");
    stampaLista(a);
}

```

```

    Nodo *b = NULL;
    for (int i = 0; i < 3 && a != NULL; i++) {    /* i tre migliori, in testa a B */
        Nodo *n = estraiMax(&a);
        n->next = b;
        b = n;
    }
    printf("Lista A:\n");
    stampaLista(a);
    printf("Lista B:\n");
    stampaLista(b);

    scriviFile(par.output, b);
    libera(a);
    libera(b);
    return EXIT_SUCCESS;
}

```

Esecuzione di prova

Il file input.txt contiene dieci artigiani:

```

Emma Jones 42 81 160 P
Bob Wilson 29 102 182 M
Alice Davis 35 70 175 P
Hannah Smith 55 63 149 P
Frank Brown 67 92 198 M
Charlie Williams 48 118 168 M
Grace Taylor 23 105 185 P
David Johnson 31 79 172 M
Julia Moore 41 48 150 P
Ian Miller 63 67 145 M

```

```
$ ./artigiani input uscita
input = input.txt, output = uscita.txt
Lista A ordinata per età:
  Grace      Taylor  23  105  185 P  1.76
  Bob        Wilson  29  102  182 M  1.78
  David      Johnson  31   79  172 M  2.18
  Alice      Davis   35   70  175 P  2.50
  Julia      Moore   41   48  150 P  3.12
  Emma       Jones   42   81  160 P  1.98
  Charlie    Williams  48  118  168 M  1.42
  Hannah     Smith   55   63  149 P  2.37
  Ian        Miller  63   67  145 M  2.16
  Frank      Brown   67   92  198 M  2.15
Lista A:
  Grace      Taylor  23  105  185 P  1.76
  Bob        Wilson  29  102  182 M  1.78
  David      Johnson  31   79  172 M  2.18
  Emma       Jones   42   81  160 P  1.98
  Charlie    Williams  48  118  168 M  1.42
  Ian        Miller  63   67  145 M  2.16
  Frank      Brown   67   92  198 M  2.15
Lista B:
  Hannah     Smith   55   63  149 P  2.37
  Alice      Davis   35   70  175 P  2.50
```

```
Julia      Moore   41   48  150 P  3.12
```

Le tre medie più alte sono quelle di Julia Moore ($150 / 48 = 3,12$), Alice Davis (2,50) e Hannah Smith (2,37). Sono estratte in quest'ordine e inserite ogni volta in testa a B, che quindi le contiene in ordine inverso di estrazione. Il file `uscita.txt` contiene:

```
Hannah Smith 55 63 149 P  2.37
Alice Davis 35 70 175 P  2.50
Julia Moore 41 48 150 P  3.12
```

ATTENZIONE

Spostare un nodo senza copiarlo

Per spostare un elemento da una lista all'altra non serve allocare un nuovo nodo e liberare il vecchio: basta scollegarlo dalla prima, collegando il predecessore al successore, e collegarlo nella seconda. Se il nodo da scollegare è il primo, cambia il puntatore di testa, per questo `estraiMax` riceve `Nodo **lista`. Il campo `next` del nodo estratto va riportato a `NULL` o reimpostato subito, altrimenti continuerebbe a indicare un nodo della prima lista.

Esempio svolto: coda di parole

Un programma completo che legge parole da un file in una coda di stringhe allocate dinamicamente, elabora la coda un numero fissato di volte, la trasferisce in un array di puntatori e la ordina.

Il problema

IL PROBLEMA

Il programma riceve i nomi di due file di testo con estensione «.txt»: il primo contiene una parola per riga, il secondo è il file di output. Il programma deve: leggere le parole in una **coda**, allocando per ciascuna esattamente lo spazio necessario; elaborare la coda estraendo e reinserendo ogni elemento **una sola volta**, e invertendo prima del reinserimento le parole di lunghezza dispari; trasferire le parole in un array di stringhe, ordinarlo alfabeticamente con il bubble sort e scriverlo nel file di output.

Analisi e scelte

La coda è realizzata con una lista e due puntatori, alla testa e al fondo, più un contatore degli elementi. Ogni nodo contiene un puntatore a una stringa allocata con `malloc(strlen(riga) + 1)`: lo spazio in più è per il terminatore.

- **accoda ed estrai non copiano le stringhe**: spostano soltanto il puntatore. La stessa memoria allocata in lettura passa dalla coda all'array e viene liberata una sola volta alla fine.
- **elaboraCoda** fissa il numero di iterazioni **prima** di iniziare. Usare come limite $q \rightarrow n$ mentre la coda cambia, o ciclare finché la coda non è vuota, darebbe un numero di passi sbagliato o un ciclo infinito, perché ogni parola estratta viene subito reinserita.
- **inverti** scambia il primo e l'ultimo carattere, poi il secondo e il penultimo, e così via fino al centro: è un'inversione vera, non una rotazione dei caratteri.
- **ordinaInArray** svuota la coda in un array di `char *` e lo ordina scambiando i **puntatori**, che costa sempre lo stesso indipendentemente dalla lunghezza delle parole.

Il programma

```
/* parole.c: legge parole da un file in una coda, inverte quelle di lunghezza dispari,
   le ordina alfabeticamente con il bubble sort e le scrive in un file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXNOME 100
#define MAXPAROLA 50

typedef struct {
    char input[MAXNOME + 1];
    char output[MAXNOME + 1];
} Parametri;

typedef struct Nodo {
    char *parola;                /* stringa allocata dinamicamente */
    struct Nodo *next;
} Nodo;
```

```
typedef struct {
    Nodo *testa, *coda;
    int n;                      /* numero di elementi */
} Coda;

int haEstensione(const char *nome, const char *est)
{
    const char *punto = strrchr(nome, '.');
    return punto != NULL && strcmp(punto, est) == 0;
}

Parametri leggiParametri(int argc, char *argv[])
{
    Parametri p;
    if (argc != 3 || strlen(argv[1]) > MAXNOME || strlen(argv[2]) > MAXNOME
        || !haEstensione(argv[1], ".txt") || !haEstensione(argv[2], ".txt")) {
        fprintf(stderr, "Uso: %s parole.txt risultato.txt\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    strcpy(p.input, argv[1]);
    strcpy(p.output, argv[2]);
    return p;
}
```

```

/* inserisce in fondo un nodo che contiene la stringa indicata (senza copiarla) */
void accoda(Coda *q, char *s)
{
    Nodo *n = malloc(sizeof(Nodo));
    if (n == NULL) {
        fprintf(stderr, "Memoria insufficiente\n");
        exit(EXIT_FAILURE);
    }
    n->parola = s;
    n->next = NULL;
    if (q->coda == NULL)
        q->testa = n;
    else
        q->coda->next = n;
    q->coda = n;
    q->n++;
}

```

```

/* estrae il primo elemento e restituisce la sua stringa (coda non vuota) */
char *estrai(Coda *q)
{
    Nodo *primo = q->testa;
    char *s = primo->parola;
    q->testa = primo->next;
    if (q->testa == NULL)
        q->coda = NULL;
    free(primo);
    q->n--;
    return s;
}

void stampaCoda(const Coda *q)
{
    for (const Nodo *p = q->testa; p != NULL; p = p->next)
        printf("%s\n", p->parola);
}

```

```

void leggiFile(const char *nome, Coda *q)
{
    FILE *f = fopen(nome, "r");
    if (f == NULL) {
        fprintf(stderr, "Impossibile aprire %s\n", nome);
        exit(EXIT_FAILURE);
    }
    char riga[MAXPAROLA + 2];
    while (fgets(riga, sizeof riga, f) != NULL) {
        riga[strcspn(riga, "\n")] = '\0';          /* toglie l'a capo */
        if (strlen(riga) == 0)
            continue;
        char *s = malloc(strlen(riga) + 1);        /* +1 per il terminatore */
        if (s == NULL) {
            fprintf(stderr, "Memoria insufficiente\n");
            exit(EXIT_FAILURE);
        }
        strcpy(s, riga);
        accoda(q, s);
    }
    fclose(f);
}

```

```

void inverti(char *s)
{
    int i = 0, j = strlen(s) - 1;
    while (i < j) {                                /* scambia gli estremi e si avvicina al centro */
        char t = s[i];
        s[i] = s[j];
        s[j] = t;
        i++;
        j--;
    }
}

/* ogni parola viene estratta e reinserita una sola volta,
   invertita se ha lunghezza dispari */
void elaboraCoda(Coda *q)
{
    int n = q->n;                                  /* numero fissato prima di iniziare */
    for (int i = 0; i < n; i++) {
        char *s = estrai(q);
        if (strlen(s) % 2 != 0)
            inverti(s);
        accoda(q, s);
    }
}

```

```

/* svuota la coda in un array di puntatori e lo ordina con il bubble sort */
char **ordinaInArray(Coda *q, int *n)
{
    *n = q->n;
    char **r = malloc(*n * sizeof(char *));
    if (r == NULL) {
        fprintf(stderr, "Memoria insufficiente\n");
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < *n; i++)
        r[i] = estrai(q);
    for (int passo = 1; passo < *n; passo++)
        for (int i = 0; i < *n - passo; i++)
            /* scambia i puntatori, non i caratteri */
            if (strcmp(r[i], r[i + 1]) > 0) {
                char *t = r[i];
                r[i] = r[i + 1];
                r[i + 1] = t;
            }
    return r;
}

```

```

void scriviFile(const char *nome, char **r, int n)
{
    FILE *f = fopen(nome, "w");
    if (f == NULL) {
        fprintf(stderr, "Impossibile scrivere %s\n", nome);
        exit(EXIT_FAILURE);
    }
    for (int i = 0; i < n; i++)
        fprintf(f, "%s\n", r[i]);
    fclose(f);
}

int main(int argc, char *argv[])
{
    Parametri p = leggiParametri(argc, argv);
    printf("input = %s, output = %s\n", p.input, p.output);
}

```

```

Coda q = {NULL, NULL, 0};
leggiFile(p.input, &q);
printf("Coda iniziale:\n");
stampaCoda(&q);

elaboraCoda(&q);
printf("Coda dopo l'elaborazione:\n");
stampaCoda(&q);

int n;
char **r = ordinaInArray(&q, &n);
printf("Array ordinato:\n");
for (int i = 0; i < n; i++)
    printf("%s\n", r[i]);

scriviFile(p.output, r, n);
for (int i = 0; i < n; i++)
    free(r[i]);
free(r);
return EXIT_SUCCESS;
}

```

Esecuzione di prova

Il file `parole.txt` contiene, una per riga: sole, cielo, albero, programma, mare, zio, luna, re. Le parole di lunghezza dispari sono cielo (5), programma (9) e zio (3), che diventano oleic, ammargorp e oiz.

```

$ ./parole parole.txt risultato.txt
input = parole.txt, output = risultato.txt
Coda iniziale:
sole
cielo
albero
programma
mare
zio
luna
re
Coda dopo l'elaborazione:
sole
oleic
albero
ammargorp
mare
oiz
luna
re
Array ordinato:
albero
ammargorp
luna

```

```
mare  
oiz  
oleic  
re  
sole
```

Il file `risultato.txt` contiene le stesse otto parole dell'array ordinato, una per riga. L'ordine alfabetico di `strcmp` confronta i codici dei caratteri: con parole tutte minuscole coincide con l'ordine del dizionario, mentre una lettera maiuscola precederebbe qualunque minuscola.

DA RICORDARE

Quando una stringa va allocata dinamicamente invece che in un array di dimensione fissa?

Quando la sua lunghezza non è nota in anticipo e cambia molto da un dato all'altro, oppure quando la stringa deve sopravvivere alla funzione che la crea, come qui, dove ogni parola letta in `leggiFile` resta nella coda e poi nell'array. Si alloca la lunghezza più 1 per il terminatore, si copia con `strcpy` e la si libera con `free` quando non serve più; chi la sposta tra strutture diverse deve sapere in ogni momento chi ne è responsabile.