

Manuale Completo di Programmazione C per il Parziale Pratico

Questo manuale copre tutti gli argomenti fondamentali del corso, con esempi pratici basati su codice reale.

Indice

- Manuale Completo di Programmazione C per il Parziale Pratico
 - Indice
 - I. Concetti Base del Linguaggio C
 - * A. Sintassi e Semantica
 - * B. Variabili e Tipi di Dato
 - * C. Input e Output
 - * D. Operatori e Espressioni
 - * E. Controllo di Flusso
 - II. Strutture Dati Composte
 - * A. Stringhe in C
 - * B. Funzioni per Stringhe (string.h)
 - strcpy - Copia stringhe
 - strcmp - Confronta stringhe
 - strlen - Lunghezza stringa
 - strcat - Concatena stringhe
 - Altri funzioni utili
 - * C. Array di Stringhe
 - * D. Lettura Stringhe da Input
 - * E. Array e Macro
 - * B. Strutture (struct)
 - * C. Tipi Enumerativi (enum)
 - III. Funzioni e Passaggio Parametri
 - * A. Passaggio per Valore vs Riferimento
 - * B. Array come Parametri
 - IV. Array e Pattern Algoritmici
 - * A. Pattern ForEach (Scansione)
 - * B. Pattern Filter (Filtro)
 - * C. Pattern Reduce (Accumulazione)
 - * D. Algoritmi di Ordinamento
 - V. ADT Lista Sequenziale
 - * A. Concetto
 - * B. Operazioni di Inserimento
 - * C. Altre Operazioni
 - VI. ADT Lista Collegata
 - * A. Concetto
 - * B. Struttura del Nodo
 - * C. Operazioni Base
 - * D. Pattern di Scorrimento
 - * E. Memoria Dinamica
 - * F. Confronto Liste Sequenziali vs Collegate

- VII. Input/Output su File
 - * A. Apertura e Chiusura File
 - * B. File di Testo
 - * C. File Binari
 - * D. Pattern Comuni di Lettura File
 - 1. File Binario con Struct (dal progetto Calorie)
 - 2. File Testo + File Binario (pattern completo)
 - 3. Gestione Errori Multipli File
 - VIII. Makefile e Compilazione
 - * A. Struttura di un Progetto Multi-file
 - * B. Formato del Makefile
 - * C. Esempio Completo (dal codice reale)
 - * D. Uso del Makefile
 - IX. Debugging con GDB
 - * A. Compilazione per il Debug
 - * B. Comandi Base di GDB
 - * C. Esempio di Sessione Debug
 - * D. Debug di Bug Logici
 - X. Argomenti da Linea di Comando
 - * A. argc e argv
 - * B. Esempio Completo con Validazione (dal progetto)
 - XI. Consigli Pratici per il Parziale
 - * A. Checklist Pre-Compilazione
 - * B. Errori Comuni
 - * C. Pattern da Ricordare
 - XII. Pattern Avanzati e Tecniche Comuni
 - * A. Algoritmo MCD (Massimo Comun Divisore)
 - * B. Ordinamento con Criteri Multipli
 - * C. Contare Occorrenze in Array
 - * D. Copiare Lista in Array per Ordinamento
 - * E. Aggiornare Elementi in Lista
 - * F. Cicli con Condizioni Complesse
 - * G. Stampa Formattata
 - * H. Esempio Completo: Lettura File con Matricole
 - Riepilogo Rapido
 - * Tipi di Dato
 - * Puntatori
 - * Liste
 - * File
 - * Compilazione
 - **Buono lavoro!**
-

I. Concetti Base del Linguaggio C

A. Sintassi e Semantica

La **sintassi** definisce le regole grammaticali del C. La **semantica** definisce il significato dei programmi.

- **Flusso predefinito:** Le istruzioni vengono eseguite sequenzialmente dall'alto verso il basso
- **Macchina astratta C:** Modello definito dagli standard ANSI C per prevedere il comportamento dei programmi

B. Variabili e Tipi di Dato

Tipi di dato scalari:

Tipo	Dimensione	Intervallo	Note
char	8 bit	-128 a 127 (con segno)	Usato per caratteri, ma è un intero
int	16/32 bit	Dipende dal compilatore	Intero con segno
float	32 bit	Numeri reali	Singola precisione
double	64 bit	Numeri reali	Doppia precisione

Esempio: Definizione e inizializzazione

```
int a = 10;           // Definizione con inizializzazione
char c = 'A';        // Carattere
float media = 0.0;   // Numero reale
```

ATTENZIONE: Le variabili non inizializzate contengono valori casuali!

C. Input e Output

Printf - Output formattato:

```
int x = 42;
float pi = 3.14;
printf("Intero: %d, Float: %.2f\n", x, pi);
// Output: Intero: 42, Float: 3.14
```

Specificatori comuni: %d (int), %f (float), %c (char), %s (stringa)

Scanf - Input:

```
int eta;
scanf("%d", &eta); // IMPORTANTE: usare & per passare l'indirizzo
```

D. Operatori e Espressioni

Conversioni di tipo:

```
int a = 5, b = 2;
float media1 = (a + b) / 2;      // Risultato: 3.0 (divisione intera!)
float media2 = (a + b) / 2.0;    // Risultato: 3.5 (divisione reale)
float media3 = (float)(a + b) / 2; // Risultato: 3.5 (cast esplicito)
```

Gerarchia delle conversioni: int < long < float < double

E. Controllo di Flusso

Istruzioni condizionali:

```
if (voto >= 18) {  
    printf("Promosso\n");  
} else {  
    printf("Bocciato\n");  
}
```

Operatori logici:

- `&&` (AND logico)
- `||` (OR logico)
- `!` (NOT logico)

Valutazione in cortocircuito: in `a && b`, se `a` è falso, `b` non viene valutato

Switch:

```
switch (giorno) {  
    case 1:  
        printf("Lunedì\n");  
        break;  
    case 2:  
        printf("Martedì\n");  
        break;  
    default:  
        printf("Altro giorno\n");  
}
```

Cicli:

```
// While - controllo prima dell'esecuzione  
while (i < 10) {  
    printf("%d ", i);  
    i++;  
}  
  
// Do-While - esecuzione garantita almeno una volta  
do {  
    printf("Inserisci numero: ");  
    scanf("%d", &n);  
} while (n < 0);  
  
// For - quando si conosce il numero di iterazioni  
for (int i = 0; i < 10; i++) {  
    printf("%d ", i);  
}
```

Break e Continue:

- `break`: esce immediatamente dal ciclo

- continue: salta alla prossima iterazione
-

II. Strutture Dati Composte

A. Stringhe in C

In C, le stringhe sono **array di caratteri terminati da '\0'** (carattere nullo).

Dichiarazione e inizializzazione:

```
char nome1[20] = "Mario";      // Terminatore automatico
char nome2[] = "Luigi";        // Dimensione automatica (6 char)
char nome3[20];                // Non inizializzata (contiene spazzatura)

// Caratteri individuali
nome1[0] = 'M';   // Primo carattere
nome1[1] = 'a';   // Secondo carattere
```

Rappresentazione in memoria:

```
char nome[6] = "Mario";
// Memoria: ['M'][ 'a' ][ 'r' ][ 'i' ][ 'o' ][ '\0' ]
//           [0]  [1]  [2]  [3]  [4]  [5]
```

IMPORTANTE: Il terminatore '\0' è fondamentale! Senza di esso, le funzioni di stringa non funzionano correttamente.

B. Funzioni per Stringhe (string.h)

Devi includere: #include <string.h>

strcpy - Copia stringhe

```
char dest[50];
char src[] = "Ciao";
strcpy(dest, src); // dest diventa "Ciao"

// ATTENZIONE: dest deve essere abbastanza grande!
char troppo_piccolo[3];
strcpy(troppo_piccolo, "Ciao"); // ERRORE! Buffer overflow
```

strcmp - Confronta stringhe

```
char s1[] = "Mario";
char s2[] = "Mario";
char s3[] = "Luigi";

// NON usare == per confrontare stringhe!
if (s1 == s2) { ... } // SBAGLIATO! Confronta indirizzi
```

```

// Usare strcmp:
if (strcmp(s1, s2) == 0) {
    printf("Stringhe uguali\n");
}

if (strcmp(s1, s3) < 0) {
    printf("s1 viene prima di s3\n"); // Ordine alfabetico
}

```

Valori di ritorno strcmp: - == 0: stringhe uguali - < 0: prima stringa precede la seconda (alfabeticamente) - > 0: prima stringa segue la seconda

strlen - Lunghezza stringa

```

char nome[] = "Mario";
int len = strlen(nome); // len = 5 (NON conta il '\0')

// Ciclo su ogni carattere
for (int i = 0; i < strlen(nome); i++) {
    printf("%c ", nome[i]);
}
// Output: M a r i o

```

strcat - Concatena stringhe

```

char dest[50] = "Ciao ";
char src[] = "Mondo";
strcat(dest, src);
printf("%s\n", dest); // Output: Ciao Mondo

```

// ATTENZIONE: dest deve avere spazio sufficiente!

Altri funzioni utili

```

// strchr - Trova un carattere nella stringa
char str[] = "Mario";
char *pos = strchr(str, 'r');
if (pos != NULL) {
    printf("Trovato alla posizione: %ld\n", pos - str); // 2
}

// strstr - Trova una sottostringa
char testo[] = "Ciao Mario";
char *pos2 = strstr(testo, "Mario");
if (pos2 != NULL) {
    printf("Trovato: %s\n", pos2); // Output: Mario
}

```

C. Array di Stringhe

```
// Array di 5 stringhe, ognuna lunga max 20 caratteri
char nomi[5][20] = {
    "Mario",
    "Luigi",
    "Anna",
    "Paolo",
    "Giulia"
};

// Accesso
printf("%s\n", nomi[0]); // Mario
printf("%c\n", nomi[0][0]); // M (primo carattere)

// Ciclo su tutte le stringhe
for (int i = 0; i < 5; i++) {
    printf("%s\n", nomi[i]);
}
```

D. Lettura Stringhe da Input

```
char nome[50];

// scanf - ATTENZIONE: si ferma al primo spazio!
scanf("%s", nome); // Legge fino a spazio/newline
// Input: "Mario Rossi" → nome = "Mario"

// fgets - Legge anche gli spazi
fgets(nome, 50, stdin); // Legge max 49 caratteri + '\0'
// Input: "Mario Rossi" → nome = "Mario Rossi\n"

// Rimuovere newline da fgets:
nome[strcspn(nome, "\n")] = '\0';
```

E. Array e Macro

Le macro (`#define`) sono fondamentali per definire costanti.

```
#include <stdio.h>
#define DIM 5

int main(void) {
    int numeri[DIM]; // Array di 5 interi

    // Inizializzazione
    for (int i = 0; i < DIM; i++) {
        numeri[i] = i * 10;
    }
```

```

// Stampa
for (int i = 0; i < DIM; i++) {
    printf("%d ", numeri[i]);
}
// Output: 0 10 20 30 40
}

```

B. Strutture (struct)

Aggregano dati eterogenei.

Esempio: Struttura Persona

```

typedef struct {
    char nome[50];
    int eta;
    float altezza;
} Persona;

```

```

Persona studente;
studente.eta = 20; // Accesso con notazione puntata
strcpy(studente.nome, "Mario");

```

C. Tipi Enumerativi (enum)

Definiscono un insieme di valori simbolici.

Esempio: Enum Vaccini (dal codice reale)

```

typedef enum {
    cimurro = 'C',
    epatite = 'E',
    parvovirosi = 'P',
    null = 'N'
} Vaccino;

```

```
Vaccino v = cimurro; // v vale 'C'
```

Gli enum sono totalmente ordinati e possono essere confrontati con <, >, ecc.

III. Funzioni e Passaggio Parametri

A. Passaggio per Valore vs Riferimento

Il C usa **passaggio per valore** per default. Per modificare variabili del chiamante, si usano **puntatori**.

Esempio: Swap (scambio di valori)

```

void swap(int *pm, int *pn) {
    int temp = *pm; // Legge il valore puntato da pm

```

```

    *pm = *pn;           // Scrive nella memoria di pm
    *pn = temp;
}

int main() {
    int a = 2, b = 3;
    swap(&a, &b);   // Passa gli indirizzi con &
    printf("%d %d\n", a, b); // Output: 3 2
    return 0;
}

```

B. Array come Parametri

Gli array sono **sempre passati per riferimento** (si passa l'indirizzo del primo elemento).

```

void azzera(int v[], int dim) {
    for (int i = 0; i < dim; i++) {
        v[i] = 0; // Modifica diretta nel chiamante
    }
}

int main() {
    int numeri[5] = {1, 2, 3, 4, 5};
    azzera(numeri, 5); // numeri viene modificato
    // Ora numeri = {0, 0, 0, 0, 0}
}

```

Nota: Passare sempre la dimensione dell'array come parametro separato!

IV. Array e Pattern Algoritmici

Tutti i pattern usano cicli **for** che iterano sugli indici dell'array.

A. Pattern ForEach (Scansione)

Applica un'operazione su ogni elemento.

```

void stampa(int a[], int dim) {
    for (int i = 0; i < dim; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

```

B. Pattern Filter (Filtro)

Crea un nuovo array contenente solo elementi che soddisfano una condizione.

Esempio: Filtrare numeri positivi

```

int a[5] = {-2, 5, -1, 8, 3};
int b[5]; // Array di destinazione
int dl = 0; // Dimensione logica di b

for (int i = 0; i < 5; i++) {
    if (a[i] > 0) {
        b[dl] = a[i]; // Inserisce nell'indice dl
        dl++;           // Incrementa dimensione logica
    }
}
// b = {5, 8, 3}, dl = 3

```

C. Pattern Reduce (Accumulazione)

Calcola un valore singolo da un array.

Esempio: Somma e massimo

```

// Somma
int somma = 0;
for (int i = 0; i < DIM; i++) {
    somma = somma + a[i];
}

// Massimo
int max = a[0]; // Inizializza col primo elemento
for (int i = 1; i < DIM; i++) {
    if (a[i] > max) {
        max = a[i];
    }
}

```

D. Algoritmi di Ordinamento

Insertion Sort:

```

void insertionSort(int a[], int dim) {
    for (int dlo = 0; dlo < dim; dlo++) {
        int j = dlo, m = a[dlo];
        // Sposta a destra gli elementi maggiori di m
        while (j > 0 && m < a[j - 1]) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = m; // Inserisce m nella posizione corretta
    }
}

```

Selection Sort:

```

void selectionSort(int a[], int dim) {

```

```

for (int i = 0; i < dim - 1; i++) {
    int min_idx = i;
    // Trova il minimo nella parte non ordinata
    for (int j = i + 1; j < dim; j++) {
        if (a[j] < a[min_idx]) {
            min_idx = j;
        }
    }
    // Scambia
    int temp = a[i];
    a[i] = a[min_idx];
    a[min_idx] = temp;
}
}

```

V. ADT Lista Sequenziale

A. Concetto

L'implementazione sequenziale usa un **array** come contenitore fisico.

Caratteristiche:

- Dimensione fisica (DIMENSIONE): massimo prefissato
- Dimensione logica (n_elementi): numero effettivo di elementi

Esempio: Struttura della Lista

```

#define DIMENSIONE 100

typedef struct {
    int n_elementi;          // Dimensione logica
    int dati[DIMENSIONE];   // Array fisico
} Lista;

void nuova_lista(Lista *pl) {
    pl->n_elementi = 0;    // Lista vuota
}

int vuota(Lista l) {
    return l.n_elementi == 0;
}

int piena(Lista l) {
    return l.n_elementi == DIMENSIONE;
}

```

B. Operazioni di Inserimento

Inserimento in testa (richiede scorrimento):

```
void insTesta(Lista *pl, int numero) {
    if (piena(*pl)) {
        printf("Errore: lista piena\n");
        return;
    }

    // Sposta tutti gli elementi a destra
    for (int i = pl->n_elementi; i > 0; i--) {
        pl->dati[i] = pl->dati[i - 1];
    }

    pl->dati[0] = numero; // Inserisce in testa
    pl->n_elementi++; // Incrementa dimensione logica
}
```

Inserimento ordinato:

```
void insOrd(Lista *pl, int numero) {
    if (piena(*pl)) {
        printf("Errore: lista piena\n");
        return;
    }

    int i = pl->n_elementi;
    // Sposta a destra gli elementi maggiori di 'numero'
    while (i > 0 && pl->dati[i - 1] > numero) {
        pl->dati[i] = pl->dati[i - 1];
        i--;
    }

    pl->dati[i] = numero; // Inserisce nella posizione corretta
    pl->n_elementi++;
}
```

C. Altre Operazioni

Ricerca:

```
int ricerca(Lista l, int valore) {
    for (int i = 0; i < l.n_elementi; i++) {
        if (l.dat[i] == valore) {
            return i; // Ritorna l'indice
        }
    }
    return -1; // Non trovato
}
```

Stampa:

```
void stampa(Lista l) {
    for (int i = 0; i < l.n_elementi; i++) {
        printf("%d ", l.dat[i]);
    }
    printf("\n");
}
```

Limiti delle liste sequenziali:

- Dimensione massima fissata a priori
 - Inserimento ed eliminazione costosi (richiedono scorrimenti)
 - Spreco di memoria se la lista è poco piena
-

VI. ADT Lista Collegata

A. Concetto

Nelle liste collegate, gli elementi **non sono adiacenti in memoria**. La connessione avviene tramite **puntatori**.

Vantaggi:

- Nessun limite di dimensione prefissato
- Inserimento/eliminazione efficienti (no scorrimenti)
- Memoria allocata “al bisogno”

Svantaggio:

- Accesso sequenziale (non si può accedere direttamente all’i-esimo elemento)

B. Struttura del Nodo

Esempio dal codice reale (listaCani):

```
// Struttura del dato
typedef struct {
    int nChip;
    Vaccino vaccini[3];
} Cane;

// Nodo della lista
typedef struct nodo {
    Cane dato;           // Dati contenuti nel nodo
    struct nodo *next;  // Puntatore al nodo successivo
} Nodo;

// Lista come puntatore al primo nodo
typedef Nodo *Lista;
```

Spiegazione:

- Ogni nodo contiene:
 - Il dato (**Cane dato**)
 - Un puntatore al nodo successivo (**struct nodo *next**)
- La lista è un puntatore al primo nodo (**testa**)
- Lista vuota = **NULL**

C. Operazioni Base

1. Inizializzazione (lista vuota):

```
void nuova_lista(Lista *pl) {
    *pl = NULL; // Puntatore a NULL = lista vuota
}
```

2. Inserimento in testa:

```
void insTesta(Lista *pl, Cane d) {
    // Alloca memoria per il nuovo nodo
    Nodo *aux = (Nodo *)malloc(sizeof(Nodo));
    if (aux == NULL) {
        printf("Errore allocazione memoria\n");
        exit(100);
    }

    // Inserisce i dati nel nuovo nodo
    aux->dato = d;

    // Il nuovo nodo punta alla vecchia testa
    aux->next = *pl;

    // Aggiorna la testa della lista
    *pl = aux;
}
```

Passi dell'inserimento in testa:

1. Alloca memoria per un nuovo nodo con **malloc()**
2. Copia i dati nel nuovo nodo
3. Fa puntare il nuovo nodo alla vecchia testa
4. Aggiorna la testa della lista

3. Inserimento in coda:

```
void insInCoda(Lista *pl, Cane d) {
    // Scorre fino alla fine della lista
    while (*pl != NULL) {
        pl = &(*pl)->next;
    }

    // Quando trova NULL (fine lista), inserisce in testa
    insTesta(pl, d);
}
```

4. Ricerca:

```
int ricerca(Lista *pl, int numero) {
    Lista l = *pl;
    int index = 0;

    // Scorre la lista
    while (l != NULL) {
        // Se trova il chip cercato, ritorna l'indice
        if (l->dato.nChip == numero) {
            return index;
        }
        l = l->next; // Passa al nodo successivo
        index++;
    }

    return -1; // Non trovato
}
```

5. Stampa:

```
void stampa(Lista l) {
    // Scorre tutti i nodi
    while (l != NULL) {
        printf("%d ", l->dato.nChip);
        l = l->next; // Passa al nodo successivo
    }
    printf("\n");
}
```

6. Eliminazione di un nodo:

```
void elimina(Lista *pl, int valore) {
    // Cerca il nodo da eliminare
    while (*pl != NULL && (*pl)->dato.nChip != valore) {
        pl = &(*pl)->next;
    }

    if (*pl != NULL) { // Trovato
        Nodo *temp = *pl; // Salva il nodo da eliminare
        *pl = (*pl)->next; // "Salta" il nodo da eliminare
        free(temp); // Libera la memoria
    }
}
```

7. Conversione enum a stringa (utility comune):

```
// Utile per stampare valori enum in forma leggibile
const char* vaccino_to_string(Vaccino v) {
    switch (v) {
        case cimurro: return "cimurro";
```

```

        case epatite:      return "epatite";
        case parvovirosi: return "parvovirosi";
        case null:         return " ";
        default:           return "sconosciuto";
    }
}

// Uso:
Vaccino v = cimurro;
printf("Vaccino: %s\n", vaccino_to_string(v));
// Output: Vaccino: cimurro

```

D. Pattern di Scorrimento

Pattern fondamentale per scorrere una lista:

```

Lista l = testa; // Parte dalla testa
while (l != NULL) {
    // Elabora l->dato
    printf("%d ", l->dato.valore);

    l = l->next; // Passa al successivo
}

```

Scorrimento con puntatore al puntatore:

```

Lista *pl = &testa;
while (*pl != NULL) {
    // Elabora (*pl)->dato

    pl = &(*pl)->next; // Passa al successivo
}

```

E. Memoria Dinamica

Allocazione con malloc:

```

Nodo *nuovo = (Nodo *)malloc(sizeof(Nodo));
if (nuovo == NULL) {
    // Gestione errore
    exit(1);
}

```

Deallocazione con free:

```
free(nuovo); // Libera la memoria allocata
```

IMPORTANTE: Ogni malloc() deve avere un corrispondente free() per evitare memory leak!

F. Confronto Liste Sequenziali vs Collegate

Operazione	Lista Sequenziale	Lista Collegata
Accesso i-esimo elemento	$O(1)$ diretto	$O(n)$ sequenziale
Inserimento in testa	$O(n)$ scorrimento	$O(1)$
Inserimento in coda	$O(1)$	$O(n)$ scorrimento
Ricerca	$O(n)$	$O(n)$
Dimensione	Fissa	Dinamica
Memoria	Fissa (spreco possibile)	Solo necessaria

VII. Input/Output su File

A. Apertura e Chiusura File

```
FILE *fp;
```

```
// Apertura in lettura
fp = fopen("dati.txt", "r");
if (fp == NULL) {
    printf("Errore apertura file\n");
    return 1;
}

// ... operazioni sul file ...

fclose(fp); // Chiusura
```

Modi di apertura:

- "r": lettura (read)
- "w": scrittura (write, sovrascrive)
- "a": append (aggiunge in fondo)
- "rb", "wb": binario

B. File di Testo

Lettura con fscanf:

```
int numero;
char nome[50];
while (fscanf(fp, "%d %s", &numero, nome) == 2) {
    printf("%d %s\n", numero, nome);
}
```

IMPORTANTE: fscanf ritorna il numero di elementi letti con successo: - Ritorna 2 se ha letto correttamente sia numero che nome - Ritorna EOF (-1) alla fine del file - Ritorna meno elementi se la lettura fallisce

Scrittura con fprintf:

```
fprintf(fp, "%d %s\n", 42, "Mario");
```

Formati comuni per fscanf/fprintf: - %d - intero - %f - float - %s - stringa (si ferma a spazio/tab/newline) - %c - singolo carattere - %lf - double (in lettura)

Esempio: File con stringhe e numeri (dal progetto pasto.txt)

File pasto.txt:

```
Pane 80
Fontina 40
Mela 200
```

Lettura:

```
FILE *pft = fopen("pasto.txt", "rt");
if (pft == NULL) {
    printf("Errore apertura file\n");
    exit(1);
}

char nome_cibo[31];
float grammi;

// Legge coppie (stringa, numero)
while (fscanf(pft, "%s%f", nome_cibo, &grammi) == 2) {
    printf("Cibo: %s, Grammi: %.1f\n", nome_cibo, grammi);
}
// Output:
// Cibo: Pane, Grammi: 80.0
// Cibo: Fontina, Grammi: 40.0
// Cibo: Mela, Grammi: 200.0

fclose(pft);
```

ATTENZIONE con %s: - Non legge spazi! “Pane tostato” → legge solo “Pane” - Non serve & perché il nome dell’array è già un puntatore - Assicurati che l’array sia abbastanza grande

Esempio completo: Lettura e scrittura file testo (dal progetto Studenti)

```
typedef struct {
    char cognome[20];
    char nome[20];
    int matricola;
} Studente;

int main() {
    FILE *fp = fopen("nomi.txt", "r");
    if (fp == NULL) {
        printf("Errore apertura file\n");
        return 1;
    }

    Studente studenti[100];
```

```

int count = 0;

// Legge cognome, nome e matricola (separati da spazi)
while (fscanf(fp, "%s %s %d",
              studenti[count].cognome,
              studenti[count].nome,
              &studenti[count].matricola) == 3) {
    count++;
}
fclose(fp);

// Ordina per matricola (selection sort)
for (int i = 0; i < count - 1; i++) {
    for (int j = i + 1; j < count; j++) {
        if (studenti[i].matricola > studenti[j].matricola) {
            Studente temp = studenti[i];
            studenti[i] = studenti[j];
            studenti[j] = temp;
        }
    }
}

// Scrive su file gli studenti ordinati
fp = fopen("ordinati.txt", "w");
if (fp == NULL) {
    printf("Errore apertura file in scrittura\n");
    return 1;
}

for (int i = 0; i < count; i++) {
    fprintf(fp, "%s %s %d\n",
            studenti[i].cognome,
            studenti[i].nome,
            studenti[i].matricola);
}
fclose(fp);

return 0;
}

```

C. File Binari

Lettura con fread:

```

int numero;
char carattere;

while (fread(&numero, sizeof(int), 1, fp) == 1 &&
      fread(&carattere, sizeof(char), 1, fp) == 1) {

```

```
// Elabora numero e carattere
}
```

Scrittura con fwrite:

```
int n = 42;
fwrite(&n, sizeof(int), 1, fp);
```

Parametri di fread/fwrite:

1. Indirizzo del dato
2. Dimensione di ogni elemento (`sizeof()`)
3. Numero di elementi da leggere
4. Puntatore al file

Esempio completo: Lettura file binario (dal progetto listaCani)

```
// File binario formato: [int nChip] [int vaccino_code] ripetuto
FILE *fp = fopen(argv[1], "rb");
if (fp == NULL) {
    printf("Errore apertura file\n");
    return 1;
}

int nChip;
int vaccino_temp;
char vaccino;

// Legge coppie (nChip, vaccino) dal file
while (fread(&nChip, sizeof(int), 1, fp) == 1 &&
       fread(&vaccino_temp, sizeof(int), 1, fp) == 1) {

    // Converte int in char (es: 67 → 'C')
    vaccino = (char)vaccino_temp;

    // Elabora i dati letti
    printf("Chip: %d, Vaccino: %c\n", nChip, vaccino);
}
fclose(fp);
```

IMPORTANTE: `fread()` ritorna il numero di elementi letti. Controllare sempre il valore di ritorno!

D. Pattern Comuni di Lettura File

1. File Binario con Struct (dal progetto Calorie) Struttura file binario `calorie.dat`:

[Cibo1] [Cibo2] [Cibo3] ...

Ogni Cibo è una struct di 35 byte (31 char + 4 float)

Lettura completa:

```
typedef struct {
    char nome[31];
```

```

    float calorie;
} Cibo;

FILE *pfb = fopen("calorie.dat", "rb");
if (pfb == NULL) {
    printf("Errore apertura file\n");
    exit(2);
}

Cibo r;
Lista lista;
nuovaLista(&lista);

// Legge tutte le struct dal file
while (fread(&r, sizeof(Cibo), 1, pfb) == 1) {
    insCoda(&lista, r);
}
fclose(pfb);

```

Cosa succede: 1. fread legge 35 byte (sizeof(Cibo)) dal file 2. Li copia nella struct r 3. r viene inserita nella lista 4. Ripete fino a fine file (fread ritorna 0)

2. File Testo + File Binario (pattern completo) Dal progetto Simulazione Esame Totale:

```

int main(int argc, char *argv[]) {
    // Controlla argomenti
    if (argc != 3) {
        printf("Uso: %s [fileCalorie.dat] [filePasto.txt]\n", argv[0]);
        exit(1);
    }

    Lista lista;
    nuovaLista(&lista);

    // === FASE 1: Carica database da file binario ===
    FILE *pfb = fopen(argv[1], "rb");
    if (pfb == NULL) {
        printf("Errore apertura %s\n", argv[1]);
        exit(2);
    }

    Cibo r;
    while (fread(&r, sizeof(Cibo), 1, pfb) == 1) {
        insCoda(&lista, r);
    }
    fclose(pfb);

    // === FASE 2: Leggi dati da file testo ===
    FILE *pft = fopen(argv[2], "rt");

```

```

if (pft == NULL) {
    printf("Errore apertura %s\n", argv[2]);
    exit(3);
}

char nome_cibo[31];
float grammi;
float calorie_totali = 0.0;

// Legge coppie (nome, grammi) e calcola
while (fscanf(pft, "%s%f", nome_cibo, &grammi) == 2) {
    // Cerca il cibo nella lista e calcola calorie
    float cal_per_100g = calorie100(lista, nome_cibo);
    calorie_totali += grammi * cal_per_100g / 100.0;
}
fclose(pft);

printf("Calorie consumate: %.2f\n", calorie_totali);
return 0;
}

```

Pattern usato: 1. File binario → lista (database) 2. File testo → elaborazione (query) 3. Lista + query → risultato

3. Gestione Errori Multipli File

```

FILE *fp1 = fopen("file1.txt", "r");
if (fp1 == NULL) {
    printf("Errore apertura file1\n");
    exit(1);
}

FILE *fp2 = fopen("file2.txt", "r");
if (fp2 == NULL) {
    printf("Errore apertura file2\n");
    fclose(fp1); // Chiudi il primo prima di uscire!
    exit(2);
}

// Elaborazione...

fclose(fp1);
fclose(fp2);

```

BEST PRACTICE: - Codici di uscita diversi per errori diversi - Sempre chiudere i file già aperti prima di exit() - Controllare SEMPRE il valore di ritorno di fopen()

VIII. Makefile e Compilazione

A. Struttura di un Progetto Multi-file

Un progetto tipico è composto da:

- **File header (.h)**: dichiarazioni e prototipi (interfaccia)
- **File sorgente (.c)**: implementazione delle funzioni
- **File main (.c)**: programma principale

Esempio:

- lista.h - interfaccia dell'ADT Lista
- lista.c - implementazione delle funzioni
- main.c - programma principale

B. Formato del Makefile

Il Makefile definisce **target** (obiettivi) e **dipendenze**:

```
target: dipendenza1 dipendenza2  
        comando_compilazione
```

ATTENZIONE: Il comando deve iniziare con un TAB, non spazi!

C. Esempio Completo (dal codice reale)

```
# Compila il file oggetto listaCani.o  
# Dipendenze: listaCani.c e listaCani.h  
listaCani.o : listaCani.c listaCani.h  
        gcc -g -c listaCani.c  
  
# Compila il file oggetto main.o  
# Dipendenze: main.c e listaCani.h  
main.o : main.c listaCani.h  
        gcc -g -c main.c  
  
# Crea l'eseguibile finale  
# Dipendenze: tutti i file oggetto  
vaccinazioni : main.o listaCani.o  
        gcc -g -o vaccinazioni main.o listaCani.o
```

Opzioni gcc:

- **-g**: include informazioni di debug
- **-c**: compila senza linkare (crea .o)
- **-o nome**: specifica il nome dell'output

D. Uso del Makefile

```
make           # Compila il primo target  
make vaccinazioni # Compila target specifico  
make clean      # Pulisce i file compilati (se definito)
```

Vantaggi:

- Ricompila solo i file modificati
 - Automatizza il processo di build
 - Gestisce correttamente le dipendenze
-

IX. Debugging con GDB

A. Compilazione per il Debug

Usa sempre l'opzione -g:

```
gcc -g -o programma programma.c
```

B. Comandi Base di GDB

Avvio:

```
gdb ./programma
```

Comandi essenziali:

Comando	Abbreviazione	Descrizione
break main	b main	Breakpoint sulla funzione main
break 10	b 10	Breakpoint alla riga 10
run	r	Avvia il programma
next	n	Esegue la prossima istruzione (non entra nelle funzioni)
step	s	Esegue la prossima istruzione (entra nelle funzioni)
print variabile	p variabile	Stampa il valore di una variabile
continue	c	Continua l'esecuzione fino al prossimo breakpoint
quit	q	Esce da GDB

C. Esempio di Sessione Debug

```
$ gdb ./vaccinazioni
(gdb) break main          # Breakpoint su main
(gdb) run dati.dat        # Esegue con argomento
(gdb) print nChip          # Stampa valore di nChip
(gdb) next                 # Prossima istruzione

// ATTENZIONE: stringhe NON vogliono &
char nome[50];
scanf("%s", nome);    // CORRETTO (nome è già un puntatore)
scanf("%s", &nome);    // SBAGLIATO!
(gdb) step                 # Entra nella funzione
(gdb) continue              # Continua
```

D. Debug di Bug Logici

Strategia:

1. Identifica il primo **stato imprevisto** (valore sbagliato di una variabile)
2. Usa breakpoint prima di quel punto
3. Esamina i valori con `print`
4. Esegui passo-passo con `next` o `step`

Esempio:

```
int a, b, c;
scanf("%d%d", &a, &b);
c = a += b; // BUG: dovrebbe essere c = a + b
```

Con GDB:

```
(gdb) b 3
(gdb) r
(gdb) p a      # Controlla valore di a
(gdb) p b      # Controlla valore di b
(gdb) n        # Esegue la riga con il bug
(gdb) p a      # a è cambiato! (stato imprevisto)
(gdb) p c      # c ha valore sbagliato
```

X. Argomenti da Linea di Comando

A. argc e argv

```
int main(int argc, char *argv[]) {
    // argc: numero di argomenti (include il nome del programma)
    // argv: array di stringhe con gli argomenti

    if (argc != 2) {
        printf("Uso: %s <nome_file>\n", argv[0]);
        return 1;
    }

    printf("File: %s\n", argv[1]);
    return 0;
}
```

Esempio di esecuzione:

```
$ ./programma dati.txt
File: dati.txt
• argv[0]: nome del programma ("./programma")
• argv[1]: primo argomento ("dati.txt")
• argc: 2 (numero totale di argomenti)
```

B. Esempio Completo con Validazione (dal progetto)

```
int main(int argc, char *argv[]) {
    // Controlla numero di argomenti
    if (argc != 3) {
        printf("Uso: %s [fileCalorie] [filePasto]\n", argv[0]);
        exit(1);
    }

    // Apre primo file (binario)
    FILE *pfb = fopen(argv[1], "rb");
    if (pfb == NULL) {
        printf("Errore apertura file %s\n", argv[1]);
        exit(2);
    }

    // Apre secondo file (testo)
    FILE *pft = fopen(argv[2], "rt");
    if (pft == NULL) {
        printf("Errore apertura %s\n", argv[2]);
        fclose(pfb); // Chiudi il primo file
        exit(3);
    }

    // Elabora i file...

    fclose(pfb);
    fclose(pft);
    return 0;
}
```

BEST PRACTICE: - Sempre controllare `argc` prima di accedere a `argv[i]` - Usare `exit()` con codici diversi per errori diversi (0=successo, 1,2,3...=errori) - Chiudere i file aperti anche in caso di errore

XI. Consigli Pratici per il Parziale

A. Checklist Pre-Compilazione

1. • Ogni `malloc()` ha un controllo `if (ptr == NULL)`
2. • Ogni `fopen()` ha un controllo `if (fp == NULL)`
3. • Ogni array è passato con la sua dimensione
4. • Con `scanf()` uso sempre `&variabile`
5. • I puntatori sono inizializzati prima dell'uso
6. • Ogni `malloc()` ha un corrispondente `free()`

B. Errori Comuni

1. Dimenticare & in scanf:

```
int x;  
scanf("%d", x);    // SBAGLIATO  
scanf("%d", &x);   // CORRETTO
```

2. Dimenticare di controllare NULL:

```
FILE *fp = fopen("file.txt", "r");  
fprintf(fp, "test"); // CRASH se fp è NULL!
```

```
// CORRETTO:  
if (fp == NULL) {  
    printf("Errore\n");  
    return 1;  
}
```

3. Confondere dimensione fisica e logica:

```
// Stampa solo elementi validi (dimensione logica)  
for (int i = 0; i < lista.n_elementi; i++) { // CORRETTO  
// NON usare DIMENSIONE qui!
```

4. Non deallocare la memoria (Memory Leak):

```
// SBAGLIATO: Memory leak!  
void creaLista() {  
    Nodo *p = (Nodo *)malloc(sizeof(Nodo));  
    p->dato.valore = 10;  
    p->next = NULL;  
    // Esce dalla funzione senza fare free(p)  
    // La memoria allocata è persa!  
}
```

```
// CORRETTO: Libera sempre la memoria allocata  
void esempioDeallocazione() {  
    // Array dinamico  
    int *arr = (int *)malloc(100 * sizeof(int));  
    if (arr == NULL) exit(1);  
  
    // Usa arr...  
  
    free(arr); // Libera la memoria  
    arr = NULL; // Best practice: evita dangling pointer
```

```
// Lista collegata: serve una funzione apposita  
Lista lista;  
nuovaLista(&lista);  
// ... inserimenti ...
```

```

// Libera tutti i nodi (dal progetto listaCani)
while (lista != NULL) {
    Nodo *temp = lista;
    lista = lista->next;
    free(temp); // Libera ogni nodo
}
}

```

IMPORTANTE: Ogni `malloc()` deve avere un corrispondente `free()!`

5. Confrontare stringhe con `==`:

```

char s1[] = "Mario";
char s2[] = "Mario";

if (s1 == s2) { ... } // SBAGLIATO! Confronta indirizzi
if (strcmp(s1, s2) == 0) { ... } // CORRETTO

```

6. Non controllare il valore di ritorno di `fscanf`:

```

// SBAGLIATO: non controlla se la lettura ha successo
fscanf(fp, "%d", &x);

```

```

// CORRETTO: controlla il numero di elementi letti
if (fscanf(fp, "%d", &x) == 1) {
    // Lettura riuscita
}

// CORRETTO: ciclo di lettura
while (fscanf(fp, "%d %s", &num, nome) == 2) {
    // Elabora num e nome
}

```

7. Buffer overflow con stringhe:

```

char nome[5];
strcpy(nome, "Alessandro"); // ERRORE! Troppo lungo (11 caratteri)

// CORRETTO: usa strncpy
strncpy(nome, "Alessandro", 4);
nome[4] = '\0'; // Assicura il terminatore

```

C. Pattern da Ricordare

Scorrimento lista collegata:

```

while (lista != NULL) {
    // elabora lista->dato
    lista = lista->next;
}

```

Allocazione nodo:

```
Nodo *nuovo = (Nodo *)malloc(sizeof(Nodo));
if (nuovo == NULL) exit(1);
```

Lettura file binario:

```
while (fread(&dato, sizeof(tipo), 1, fp) == 1) {
    // elabora dato
}
```

Ricerca con indice:

```
int ricerca(Lista *pl, int valore) {
    Lista l = *pl;
    int index = 0;
    while (l != NULL) {
        if (l->dato.campo == valore)
            return index;
        l = l->next;
        index++;
    }
    return -1; // Non trovato
}
```

Inserimento ordinato con doppio ordinamento:

```
// Prima per campo1 DESC, poi per campo2 ASC
if (arr[i].campo1 < arr[j].campo1 ||
    (arr[i].campo1 == arr[j].campo1 && arr[i].campo2 > arr[j].campo2)) {
    // swap
}
```

XII. Pattern Avanzati e Tecniche Comuni

A. Algoritmo MCD (Massimo Comun Divisore)

Algoritmo di Euclide con sottrazione:

```
int MCD(int a, int b) {
    while (a != b) {
        if (a < b) {
            b = b - a;
        } else {
            a = a - b;
        }
    }
    return a;
}
```

```
// Esempio d'uso:
int main() {
```

```

int m = 12, n = 18;
printf("MCD(%d, %d) = %d\n", m, n, MCD(m, n));
// Output: MCD(12, 18) = 6
return 0;
}

```

B. Ordinamento con Criteri Multipli

Ordinare prima per un criterio, poi per un altro (dal progetto listaCani):

```

typedef struct {
    int chip;
    int n_vaccini;
} CaneInfo;

// Ordina per:
// 1. Numero vaccini DECRESLENTE (più vaccini prima)
// 2. Numero chip CRESCENTE (chip minore prima)
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        // Condizione di scambio composta
        if (arr[i].n_vaccini < arr[j].n_vaccini ||
            (arr[i].n_vaccini == arr[j].n_vaccini &&
            arr[i].chip > arr[j].chip)) {

            CaneInfo temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
}

```

Pattern generale:

```

// Per ordinare per campo1 (crescente), poi campo2 (decrescente):
if (a.campo1 > b.campo1 ||
    (a.campo1 == b.campo1 && a.campo2 < b.campo2)) {
    // scambia
}

```

C. Contare Occorrenze in Array

Conta elementi che soddisfano una condizione:

```

// Conta quanti vaccini ha fatto un cane (esclude 'N')
int conta_vaccini(Vaccino vaccini[], int dim) {
    int count = 0;
    for (int i = 0; i < dim; i++) {
        if (vaccini[i] != null) {
            count++;
        }
    }
}

```

```

    }
    return count;
}

// Uso:
Vaccino v[3] = {cimurro, epatite, null};
int n = conta_vaccini(v, 3); // n = 2

```

D. Copiare Lista in Array per Ordinamento

Pattern comune: lista → array → ordina → elabora

```

// 1. Conta elementi nella lista
int n_elementi = 0;
Lista temp = lista;
while (temp != NULL) {
    n_elementi++;
    temp = temp->next;
}

// 2. Alloca array dinamico
CaneInfo *arr = (CaneInfo*)malloc(n_elementi * sizeof(CaneInfo));
if (arr == NULL) {
    printf("Errore allocazione\n");
    exit(1);
}

// 3. Copia dati da lista ad array
temp = lista;
int idx = 0;
while (temp != NULL) {
    arr[idx].chip = temp->dato.nChip;
    arr[idx].n_vaccini = /* calcola */;
    idx++;
    temp = temp->next;
}

// 4. Ordina l'array (algoritmi efficienti su array)
// ... selection sort, bubble sort, etc ...

// 5. Elabora array ordinato
for (int i = 0; i < n_elementi; i++) {
    printf("%d\n", arr[i].chip);
}

// 6. IMPORTANTE: Libera la memoria
free(arr);

```

E. Aggiornare Elementi in Lista

Modificare dati di un nodo esistente:

```
// Trova il cane con nChip e aggiorna i suoi vaccini
Lista l = listaCani;
while (l != NULL) {
    if (l->dato.nChip == nChip) {
        // Trova il primo slot libero e inserisci
        if (l->dato.vaccini[1] == null) {
            l->dato.vaccini[1] = nuovo_vaccino;
        } else if (l->dato.vaccini[2] == null) {
            l->dato.vaccini[2] = nuovo_vaccino;
        }
        break; // Esci dal ciclo
    }
    l = l->next;
}
```

F. Cicli con Condizioni Complesse

While con modulo e operatore ternario:

```
// Incrementa di 3 se i%4!=0, altrimenti di 5
int i = 1;
while (i < 1000000) {
    i += (i % 4 != 0) ? 3 : 5;
    // Equivale a:
    // if (i % 4 != 0)
    //     i = i + 3;
    // else
    //     i = i + 5;
}
```

G. Stampa Formattata

Printf con precisione e allineamento:

```
float cal = 350.5;
printf("Calorie: %.2f\n", cal);      // 2 decimali: 350.50
printf("Calorie: %.0f\n", cal);      // 0 decimali: 351 (arrotonda)
printf("Valore: %6.2f\n", cal);      // Larghezza 6: "350.50"
printf("Valore: %06.2f\n", cal);      // Padding zeri: "350.50"
```

H. Esempio Completo: Lettura File con Matricole

File nomi.txt:

```
Gabriele Bertelli 212120
Lorenzo Zela 212126
Matteo Rizzo 212128
```

Anna Rossi 212130
Carlo Bianchi 212135

Programma completo (dal progetto Nomi Ordinati):

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

typedef struct {
    char cognome[20];
    char nome[20];
    int matricola;
} Studente;

int main() {
    FILE *fp = fopen("nomi.txt", "r");
    if (fp == NULL) {
        printf("Errore apertura file\n");
        return 1;
    }

    Studente studenti[100];
    int count = 0;

    // Legge cognome, nome, matricola (separati da spazi)
    while (fscanf(fp, "%s %s %d",
                  studenti[count].cognome,
                  studenti[count].nome,
                  &studenti[count].matricola) == 3) {
        count++;
    }
    fclose(fp);

    // Selection sort per matricola (crescente)
    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            if (studenti[i].matricola > studenti[j].matricola) {
                Studente temp = studenti[i];
                studenti[i] = studenti[j];
                studenti[j] = temp;
            }
        }
    }

    // Scrive su file gli studenti ordinati
    fp = fopen("ordinati.txt", "w");
    if (fp == NULL) {
        printf("Errore apertura file in scrittura\n");
        return 1;
```

```

    }

    for (int i = 0; i < count; i++) {
        fprintf(fp, "%s %s %d\n",
                studenti[i].cognome,
                studenti[i].nome,
                studenti[i].matricola);
    }
    fclose(fp);

    printf("File ordinato creato con successo!\n");
    return 0;
}

```

Riepilogo Rapido

Tipi di Dato

- **Scalari:** int, char, float, double
- **Composti:** struct, enum, array

Puntatori

- **&variabile:** indirizzo
- ***puntatore:** dereferenziazione
- **malloc() / free():** allocazione dinamica

Liste

- **Sequenziale:** array + dimensione logica
- **Collegata:** nodi + puntatori

File

- **Testo:** fscanf() / fprintf()
- **Binario:** fread() / fwrite()

Compilazione

- Singolo file: gcc -g -o prog prog.c
 - Multi-file: usa Makefile
 - Debug: gdb ./prog
-

Buono lavoro!