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

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

```c
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:

```c
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:

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

D. Operatori e Espressioni

Conversioni di tipo:

```c
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:

```c
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:

```c
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. 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

Esempio: Struttura della Lista

```c
#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):

```c
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.dati[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.dati[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):

```c
// 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
    }
}
```

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;
}
```

```c
// ... 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:

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

Scrittura con fprintf:

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

C. File Binari

Lettura con fread:

```c
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:

```c
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

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
(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:

```c
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

```c
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)

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

- Con `scanf()` uso sempre `&variabile`
- I puntatori sono inizializzati prima dell'uso
- 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:

```
Nodo *p = (Nodo *)malloc(sizeof(Nodo));
// ... uso di p ...
free(p);  // NON dimenticare!
```

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
}
```

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!