Fisica 2

Angelo Perotti

1 Riassunto: Algebra di Boole, Introduzione al C e Codifica Algoritmi

1.1 1. Algebra di Boole

1.1.1 Concetti Fondamentali

- **Operatori logici binari**: operano su valori VERO (1) e FALSO (0)
- **Tre operazioni base**:
  - **NOT (unario)**: `¬A` o `!A` - inversione del valore
  - **AND (binario)**: `A  B` - vero solo se entrambi gli operandi sono veri
  - **OR (binario)**: `A  B` - vero se almeno uno degli operandi è vero

1.1.2 Proprietà Matematiche

- **Commutativa**: `A OR B = B OR A`, `A AND B = B AND A`
- **Distributiva**: `A AND (B OR C) = (A AND B) OR (A AND C)`
- **Precedenza operatori**: NOT → AND → OR

1.1.3 Leggi di De Morgan

1. `A AND B = NOT ((NOT A) OR (NOT B))`
2. `A OR B = NOT ((NOT A) AND (NOT B))`

1.1.4 Concetti Avanzati

- **Tautologia**: espressione sempre vera (es. `A OR NOT A`)
- **Contraddizione**: espressione sempre falsa (es. `A AND NOT A`)
- **Equivalenza**: due espressioni con identica tabella di verità

1.2 2. Linguaggio C - Fondamenti

1.2.1 Caratteristiche del C

- **Linguaggio di medio livello**: bilanciamento tra astrazione e controllo hardware
- **Case sensitive**: distinzione tra maiuscole e minuscole
- **Compilato**: tradotto in codice macchina prima dell'esecuzione
- **Portabile**: codice trasferibile tra diverse macchine

1.2.2 Struttura del Programma

[] #include <stdio.h> int main(int argc, char *argv[]) { // corpo del programma return 0; }

1.2.3 Elementi Sintattici

- **Identificatori**: nomi per variabili, funzioni (lettere, cifre, underscore)
- **Keywords**: 32 parole riservate del linguaggio
- **Commenti**: `/* multi-linea */` o `// singola linea`

1.2.4 Sistema di Tipi e Variabili

- **Dichiarazione**: `tipo identificatore [= valore];`
- **L-value**: locazione di memoria (sinistra assegnazione)
- **R-value**: valore contenuto (destra assegnazione)
- **Costanti**: `const tipo identificatore = valore;`

1.2.5 Operatori

Aritmetici

- Base: `+`, `-`, `*`, `/`, `%` (modulo)
- Assegnazione composta: `+=`, `-=`, `*=`, `/=`, `%=`
- Incremento/decremento: `++`, `--` (pre/post fisso)

Precedenza Operatori (dal più alto al più basso)

1. Parentesi `()`, array `[]`
2. Unari `!`, `++`, `--`
3. Moltiplicativi `*`, `/`, `%`
4. Additivi `+`, `-`
5. Relazionali <, <=, >, >=
6. Uguaglianza `==`, `!=`
7. AND logico `&&`
8. OR logico ||
9. Operatore ternario `?:`
10. Assegnazione `=`, `+=`, etc.

1.3 3. Strutture di Controllo

1.3.1 Istruzioni Condizionali

if-else [] if (condizione) { // istruzioni se vero } else { // istruzioni se falso }

Operatore Ternario [] risultato = condizione ? valore_se_vero : valore_se_falso;

Ambiguità if-else

- L'`else` si associa sempre all'`if` più vicino
- Uso delle parentesi graffe per disambiguare

1.3.2 Istruzioni Iterative

Ciclo while [] while (condizione) { // corpo del ciclo // aggiornamento variabili controllo }
Esecuzione:

1. Valutazione condizione
2. Se vera: esecuzione corpo + ritorno al passo 1
3. Se falsa: uscita dal ciclo

1.3.3 Istruzioni Composte

- Raggruppamento di più istruzioni con {}
- Equivalenti a singola istruzione
- Non richiedono ; dopo la parentesi chiusa

1.4 4. Input/Output

- **Input**: `scanf()` per leggere da tastiera
- **Output**: `printf()` per scrivere su schermo
- **Caratteri**: `getchar()` e `putchar()`
- **EOF**: costante per fine file

1.5 5. Esempi di Algoritmi Implementati

1.5.1 Massimo Comune Divisore

Metodo 1 - Definizione: ricerca divisori comuni fino al minimo **Metodo 2 - Euclide**: sottrazione iterativa `MCD(n,m) = MCD(n-m,m)`

1.5.2 Moltiplicazione Binaria

Algoritmo che usa solo somme e moltiplicazioni/divisioni per 2

1.5.3 Problema delle Scale

Calcolo del numero di modi per salire una scala con passi di 1, 2 o 3 gradini: `S(n) = S(n-1) + S(n-2) + S(n-3)`

1.6 6. Processo di Compilazione

1. **Edit**: scrittura codice sorgente
2. **Preprocess**: elaborazione direttive `#include`
3. **Compilation**: traduzione in codice oggetto
4. **Link**: collegamento librerie
5. **Load**: caricamento in memoria
6. **Execute**: esecuzione programma

1.7 7. Note Teoriche Importanti

1.7.1 Macchina Astratta

- **Concetto**: astrazione dell'hardware che nasconde dettagli implementativi
- **Interprete**: componente essenziale che esegue il ciclo fetch-decode-execute
- **Livelli di astrazione**: bilanciamento tra facilità programmazione ed efficienza

1.7.2 Compilazione vs Interpretazione

- **Compilazione**: traduzione completa prima dell'esecuzione (maggiore efficienza)
- **Interpretazione**: traduzione ed esecuzione simultanee (maggiore flessibilità)

1.7.3 Standard e Portabilità

- **ANSI C (1989)**: primo standard ufficiale
- **C99**: revisione del 1999
- **Portabilità**: codice eseguibile su diverse architetture hardware

2 Array in C: Teoria e Concetti Fondamentali

2.1 8. Definizione e Motivazioni

2.1.1 Variabili Strutturate

- Gli **array** sono un arricchimento della macchina astratta C dopo le istruzioni
- Rappresentano **variabili che memorizzano insiemi di elementi** definiti da una relazione
- Analoghe ai vettori e matrici in matematica
- **Esempio**: insieme dei numeri di matricola degli studenti

2.1.2 Caratteristiche Fondamentali

- **Dato strutturato** con modalità di accesso pre-definite per celle di memoria
- **Sequenza di celle consecutive e omogenee** in memoria
- Ogni variabile nell'array è accessibile tramite un **indice** (intero 0)

2.2 9. Implementazione in C

2.2.1 Struttura Base

[] int a[100]; // Dichiarazione di array di 100 interi

Caratteristiche dell'Implementazione:

- **Contenitore** di variabili dello stesso tipo
- **Nome unico** (identificatore) + indice tra parentesi quadre `[]`
- **Accesso agli elementi**: `a[i]` per l'elemento in posizione i
- **Primo elemento**: sempre a indice 0 (`a[0]`)
- **Range di indici**: da 0 a n-1 per array di n elementi

2.2.2 Operatori e Precedenza

- Le **parentesi quadre** `[]` hanno alta precedenza (come le parentesi tonde)
- Associano da sinistra a destra
- Tra parentesi quadre può esserci qualsiasi espressione che restituisce un intero

---

2.3 10. Allocazione di Memoria

2.3.1 Memoria Fisica

- Gli elementi sono memorizzati in **celle consecutive**
- Esempio di allocazione:

```
Elemento: a[0] a[1] a[2] a[3] ...Indirizzo: 1000 1001 1002 1003 ...
```

2.3.2 Accesso agli Elementi

La macchina astratta accede tramite:

1. **Calcolo del valore dell'indice**
2. **Calcolo dell'indirizzo** rispetto alla prima cella (indice 0)

2.4 11. Dichiarazione e Inizializzazione

2.4.1 Array Statici

[] int a[100]; // Spazio riservato a compile-time

- Lo spazio in memoria è **predeterminato** a tempo di compilazione
- La dimensione **non può essere variata** dopo la dichiarazione

2.4.2 Inizializzazione

[] int a[5] = {5, 2, -5, 10, 234}; // Inizializzazione completa int b[4] = {5, 2, -5}; // Quarto elemento = 0 int b[4] = {0}; // Tutti inizializzati a zero

2.4.3 Inizializzazione Programmatica

[] int voti[5]; int i = 0; while (i < 5) { scanf("%d", &voti[i]); i++; }

2.5 12. Array Multidimensionali

2.5.1 Dichiarazione

[] int a[10][5]; // Matrice 10 righe × 5 colonne int b[10][5][20]; // Array tridimensionale

2.5.2 Caratteristiche

- **Accesso**: `a[i][j]` per elemento in riga i, colonna j
- **Memoria**: memorizzazione **lineare** (una riga dopo l'altra)
- **Numero totale di elementi**: N×M per array bidimensionale

2.5.3 Inizializzazione Multidimensionale

[] int a[4][5] = { {2, 5, -8, 7, 6}, {3, 10, 7, 6, 1}, {-1, 8, -8, 5, 3}, {2, 5, 8, 4, 2} };

Regole di Inizializzazione

- `int D[][]={1,2,3,4};` // ERRORE
- `int E[2][]={1,2,3,4};` // ERRORE (manca numero colonne)
- `int F[][2]={1,2,3,4};` // OK (righe calcolate automaticamente)

2.6 13. Array Dinamici

2.6.1 Variable Length Arrays (C99)

[] int n; scanf("%d", &n); int array[n]; // Dimensione calcolata a runtime

2.6.2 Caratteristiche

- **Flessibilità**: dimensione determinata durante l'esecuzione
- **Memoria**: spazio allocato dinamicamente
- **Limitazione**: n deve essere inizializzato **prima** della dichiarazione

---

2.7 14. Limitazioni e Considerazioni

2.7.1 Operazioni Non Permesse

[] array = 5; // Non si può assegnare all'intero array array1 = array2; // Non si può copiare un array in un altro

2.7.2 Controllo degli Indici

- **Responsabilità del programmatore**: C non controlla automaticamente i limiti
- **Accesso fuori range**: `a[100]` per array `a[100]` causa comportamento indefinito

2.7.3 Visualizzazione

[] // SBAGLIATO printf("%d", voti);
// CORRETTO for(int i = 0; i < 5; i++) { printf("%d ", voti[i]); }

2.8 15. Terminologia Tecnica

2.8.1 Classificazione

- **Array in C**: non è un tipo ma un **costruttore di tipo**
- **Forma corretta**: "variabile di tipo array di int" (non "variabile di tipo array")

2.8.2 Tipi di Allocazione

- **Array Statici**: dimensione nota a compile-time
- **Array Dinamici**: dimensione calcolata a runtime (C99)
- **Allocazione automatica**: inizializzazione all'esecuzione del blocco
- **Allocazione statica**: inizializzazione prima dell'avvio del programma

---

2.9 16. Algoritmi Comuni con Array

2.9.1 Ricerca e Confronto

- **Ricerca lineare** negli elementi
- **Confronto di array** elemento per elemento
- **Verifica proprietà** (es. matrice simmetrica: `a[i][j] == a[j][i]`)

2.9.2 Ottimizzazioni

- **Early termination**: interrompere cicli quando condizione è soddisfatta
- **Evitare confronti ridondanti**: nelle matrici simmetriche, controllare solo triangolo superiore

2.9.3 Matrici Speciali

- **Matrici Magiche**: somma costante per righe, colonne e diagonali
- **Formula somma magica**: `n * (n² + 1) / 2`

3 Riassunto: Stringhe in C e Rappresentazione delle Informazioni

3.1 17. Stringhe in C

3.1.1 Concetti fondamentali

- **Definizione**: Una sequenza di caratteri trattati come un oggetto singolo
- **Rappresentazione**: Il C usa array di caratteri (`char`) per rappresentare stringhe
- **Carattere terminatore**: Ogni stringa deve terminare con il carattere nullo `'\0'`

3.1.2 Proprietà delle stringhe in C

- Per memorizzare una stringa di n caratteri servono n+1 posizioni (per il carattere `'\0'`)
- Il carattere terminatore serve alle funzioni di manipolazione per identificare la fine della stringa

3.1.3 Dichiarazione e inizializzazione

Forma semplificata [] char mia_stringa[] = "Ciao a tutti!";

- Il compilatore calcola automaticamente la dimensione (14 caratteri = 13 + 1)
- Aggiunge automaticamente il carattere `'\0'`

Forma esplicita [] char mia_stringa[] = {'C','i','a','o',' ','a',' ','t','u','t','t','i','!','\0'};

- Il programmatore deve inserire manualmente `'\0'`

Con dimensione predefinita [] char frase[20] = "Ciao a tutti!";

- Uno dei 20 elementi viene riservato automaticamente per `'\0'`

3.1.4 Visualizzazione

- Utilizzo di `printf("%s", nome_stringa)` per stampare l'intera stringa
- `%s` è lo specificatore di formato per le stringhe
- La stampa continua fino al carattere `'\0'`

3.1.5 Lettura

- `scanf("%s", nome_stringa)` - **NON** serve l'operatore `&`
- Per singoli caratteri: `printf("%c", stringa[indice])`

3.1.6 Limitazioni

- Non è possibile assegnare direttamente: `stringa = "nuovo valore";`
- Gestione dinamica complessa (richiede allocazione manuale)

3.2 18. Rappresentazione delle Informazioni

3.2.1 Processo di codifica/decodifica

- **Informazioni → Codifica → Dati → Decodifica → Informazioni**
- I calcolatori memorizzano e manipolano informazioni sotto forma di dati organizzati secondo rappresentazioni specifiche

3.2.2 Tipi di dati

- **Tipi semplici**: interi, caratteri, numeri frazionari
- **Tipi complessi**: costruiti a partire da quelli semplici

3.2.3 Rappresentazione dei numeri interi

Sistemi numerici posizionali

- **Sistema decimale** (base 10)
- **Sistema binario** (base 2): fondamentale per i calcolatori

Interi relativi - Complemento a 2 (CA2)

- **Vantaggi**: rappresentazione uniforme di numeri positivi e negativi
- **Processo di inversione**:
  1. Inversione di tutti i bit
  2. Aggiunta di 1 al risultato
- **Esempio**: +6 = 0110 → inversione: 1001 → +1: 1010 = -6

3.2.4 Rappresentazione dei numeri frazionari

Virgola fissa

- Posizione decimale fissata in anticipo

Virgola mobile (Standard IEEE-754)

- **Componenti**: segno, esponente, mantissa
- **Precisione singola** (32 bit): 1 bit segno + 8 bit esponente + 23 bit mantissa
- **Mantissa normalizzata**: compresa tra [1,2)
- **Bias dell'esponente**: 2^(N-1) per N bit

3.2.5 Rappresentazione dei caratteri

Standard ASCII

- **American Standard Code for Information Interchange**
- Definito dal 1963
- **Esempi**:
  - 'A' = 65
  - 'a' = 97
  - Differenza tra maiuscole e minuscole: 'a' - 'A' = 32

Manipolazione caratteri

- Conversione maiuscolo/minuscolo basata sulla differenza ASCII
- Confronti diretti possibili: `if(C >= 'a' && C <= 'z')`

3.3 19. Gestione e manipolazione stringhe

3.3.1 Funzioni di libreria (cenni)

- Calcolo lunghezza
- Concatenazione
- Ricerca di caratteri o sottostringhe
- Conversioni (es. stringa → float)

3.3.2 Limitazioni del C

- Gestione dinamica complessa
- Necessità di calcolare e allocare manualmente la memoria
- **Soluzione C++**: classe String per gestione ad alto livello

3.4 Concetti chiave da ricordare

1. Le stringhe in C sono array di char terminati da `'\0'`
2. La rappresentazione binaria è fondamentale nei calcolatori
3. Il complemento a 2 è lo standard per i numeri relativi
4. IEEE-754 standardizza la rappresentazione floating point
5. ASCII fornisce la codifica standard per i caratteri
6. La codifica/decodifica è essenziale per l'interpretazione dei dati # Riassunto: Tipi di Dato in C

3.5 20. Concetti Fondamentali

3.5.1 Tipo di Dato Astratto

- **Definizione**: Un tipo caratterizzato da un insieme di valori e operazioni definite su di essi
- **Astrazione**: Visione esterna (chi usa il tipo) vs implementazione interna
- **Utilità dei tipi**:
  - Determinano l'interpretazione dei bit in memoria (es. 01100001 → 97 o 'a')
  - Definiscono la quantità di memoria da riservare
  - Permettono di rilevare errori di tipo in fase di compilazione
  - Linguaggi **statically typed**: C, C++, Java
  - Linguaggi **dynamically typed**: Python, JavaScript, PHP

3.5.2 Classificazione dei Tipi

1. **Tipi predefiniti (built-in)**: forniti dal linguaggio
   - Semplici: int, float, double, char
   - Strutturati: array
2. **Tipi definiti dal programmatore (user-defined)**: creati dall'utente

3.6 21. Tipi Predefiniti

3.6.1 Tipo int

- **Rappresentazione**: numeri interi con un sottoinsieme finito
- **Operatori**: `=`, `+`, `-`, `*`, `/`, `%`, `==`, `!=`, <, >, <=, >=
- **Qualificatori di dimensione**:
  - `short int long`
  - Tipicamente: `long` 32 bit, `short` 16 bit, `int` 16-32 bit

- **Qualificatori di segno**:
  - `signed` (con segno, MSB per il segno)
  - `unsigned` (senza segno, tutti i bit per i valori)
- **Esempio**: su 32 bit
  - `signed int`: $\{-2^{31}, \ldots, 2^{31}-1\}$
  - `unsigned int`: $\{0, \ldots, 2^{32}-1\}$

3.6.2 Tipi float e double

- **Rappresentazione**: numeri razionali in virgola mobile (IEEE-754)
- **Formato**: m $\times$ 10 (mantissa $\times$ 10ˆesponente)
- **Precisione**:
  - `float`: 4 byte, $10^3 \ldots 10^3$, 6 cifre di precisione
  - `double`: 8 byte, $10^3 \ldots 10^3$, 15 cifre di precisione
  - `long double`: spazio double
- **Avvertenze**:
  - Evitare confronti diretti `if (a == b)`
  - Usare soglie: `if (x >= y - EPSILON && x <= y + EPSILON)`

3.6.3 Tipo char

- **Rappresentazione**: caratteri ASCII come numeri interi (1 byte)
- **Ordinamento**: definito automaticamente ('0' < '1', 'A' < 'R')
- **Caratteri di controllo**: `\n`, `\b`, `\t`, `\r`
- **Operazioni**: tutte quelle degli interi (aritmetiche e relazionali)

3.6.4 Insiemi di Tipi

- **Tipi integral**: char, signed/unsigned int, short, long
  - Rappresentazione discreta, corrispondenza biunivoca con sottoinsiemi dei naturali
- **Tipi floating**: float, double, long double
  - Rappresentazione di insieme denso, numero molto grande di valori tra due reali

3.7 22. Tipi Strutturati

3.7.1 struct

- **Scopo**: aggregare informazioni eterogenee in una singola variabile
- **Sintassi di definizione**:

[] struct IdStruttura { Tipo1 nomeCampo1; Tipo2 nomeCampo2; ... TipoN nomeCampoN; };

- **Dichiarazione variabili**:

[] struct IdStruttura var1, var2;

- **Accesso ai campi**: dot notation `var1.nomeCampo`
- **Inizializzazione**:
  - Statica: `struct Data d = {12, 2, 1999};`
  - Run-time: assegnazioni individuali ai campi

3.7.2 Matrici

- **Definizione semplice**: `int matrice[20][20];`
- **Definizione con typedef**:

[] typedef int Vettore[20]; typedef Vettore MatriceIntera20Per20[20]; // oppure typedef int MatriceIntera20Per20[20][20];

3.8 23. Tipi Definiti dall'Utente

3.8.1 typedef

- **Scopo**: definire sinonimi per tipi esistenti
- **Sintassi**: `typedef TipoEsistente NuovoNome;`
- **Esempi**:

[] typedef int intero; typedef char Stringa[20]; typedef struct { int giorno; int mese; int anno; } Data;

3.8.2 enum

- **Scopo**: definire un tipo con un numero finito e predeterminato di valori
- **Sintassi**:

[] enum NomeTipo {valore1, valore2, ..., valoreN};

- **Rappresentazione interna**: costanti intere (da 0 in avanti)
- **Inizializzazione personalizzata**:

[] enum months {Jan=1, Feb, Mar, Apr, ...}; // Feb=2, Mar=3, etc.

- **Combinazione con typedef**:

[] typedef enum {lun, mar, mer, gio, ven, sab, dom} GiornoSettimana;

3.9 24. Conversioni di Tipo

3.9.1 Conversioni Implicite

- **Quando avvengono**: quando un operando non è del tipo atteso
- **Regole**:
  1. `char` e `short` → `int`
  2. Gerarchia: `int` < `long` < `unsigned` < `unsigned long` < `float` < `double` < `long double`
  3. Conversione verso il tipo di livello gerarchico superiore

3.9.2 Conversioni negli Assegnamenti

- Il valore a destra viene convertito al tipo della variabile a sinistra
- **Rischi**: troncamento e perdita di informazione
- **Esempi**:

[] int a=2, b=3; float c=10.4; a = c; // troncamento: a = 10 c = a; // trasformazione: c = 2.0 a = a/b; // divisione intera: a = 0

3.9.3 Conversioni Esplicite (Casting)

- **Sintassi**: `(tipo) espressione`
- **Esempio**:

[] float average; int num1=5, num2=2; average = (float)(num1+num2)/2; // Corretto: 3.5

3.10 25. Precedenza Operatori (parziale)

tabella di precedenze

4 Puntatori in C - Riassunto Teorico

4.1 26. Concetto di Puntatore

Un **puntatore** è una variabile che memorizza come valore (right-value) un **indirizzo di memoria** che riferisce alla locazione di un'altra variabile.

Caratteristiche principali:

- Punto di forza del linguaggio C per costruzione di funzioni, allocazione dinamica ed efficienza
- Consente l'accesso indiretto alle variabili
- "La prima variabile punta alla seconda"

4.2 27. Dichiarazione di Puntatori

4.2.1 Sintassi base:

[] TipoDato *identificatore;

Elementi della dichiarazione:

- `TipoDato`: definisce il tipo di variabile che può essere referenziata
- `*`: operatore unario di dereferenziazione
- `identificatore`: nome della variabile puntatore

4.2.2 Esempi di dichiarazione:

[] int *totale, *intermedio; char *stringa; int NumUtenti, *NumChiamate;

4.2.3 Uso di typedef per maggiore leggibilità:

[] typedef int TipoDato; typedef TipoDato *TipoPuntatore; TipoPuntatore P;

4.3 28. Operatori sui Puntatori

4.3.1 Operatore di Dereferenziazione (*)Operatore di Dereferenziazione (*)

- `*P` denota la variabile il cui indirizzo è contenuto in P
- Consente di accedere al valore della variabile puntata

[] *P = x; // Assegna il valore di x alla variabile puntata da P x = *P; // Assegna alla variabile x il valore della variabile puntata da P

4.3.2 Operatore Indirizzo-di (&)Operatore Indirizzo-di (&)

- `&x` restituisce l'indirizzo della variabile x (left-value)
- Si applica SOLO a oggetti in memoria: variabili e array
- NON si applica a espressioni, costanti o variabili di tipo register

[] P = &x; // P punta alla variabile x

4.4 29. Puntatori e Strutture

4.4.1 Accesso ai campi tramite puntatori:

[] typedef struct { int PrimoCampo; char SecondoCampo; } TipoDato;
TipoDato x, *P; P = &x;
// Due sintassi equivalenti: (*P).PrimoCampo = 12; // Dot notation con parentesi necessarie P->PrimoCampo = 12; // Operatore freccia (sintassi abbreviata)

4.5 30. Array e Puntatori

4.5.1 Relazione fondamentale:

- L'identificatore di un array è considerato come l'indirizzo del primo elemento
- `a` è un puntatore "fisso" (costante) al primo elemento dell'array

4.5.2 Equivalenze importanti:

- `a[i]` è equivalente a `*(a+i)`
- `p[i]` è equivalente a `*(p+i)` (se p è un puntatore)
- `p = a` è equivalente a `p = &a[0]`

4.5.3 Differenze tra array e puntatori:

[] char amesg[] = "It is the time"; // Array di caratteri char *pmesg = "It is the time"; // Puntatore a stringa costante

4.6 31. Aritmetica dei Puntatori

4.6.1 Operazioni consentite:

- Incremento (`++`) e decremento (`--`)
- Addizione di un intero (`+`, `+=`)
- Sottrazione di un intero (`-`, `-=`)
- Differenza tra puntatori dello stesso tipo

4.6.2 Comportamento dell'aritmetica:

- `p + i` restituisce l'indirizzo dell'i-esimo elemento successivo
- L'incremento non è di `i` byte, ma di `i * sizeof(tipo)`
- `p - q` restituisce il numero di elementi tra i due puntatori

4.7 32. Puntatore NULL e Inizializzazione

4.7.1 Regole importanti:

- I puntatori devono essere inizializzati prima dell'uso
- `NULL` indica che il puntatore non punta a nessuna informazione significativa
- `*P` è indefinito se `P` vale `NULL`

[] TipoPuntatore P = NULL; // Inizializzazione sicura

4.8 33. Qualificatore const con Puntatori

4.8.1 Principio del privilegio minimo:

- Dare alle funzioni il minimo accesso necessario per completare il loro compito
- Evitare modifiche accidentali dei dati

4.8.2 Tipi di const:

[] const int *ptr; // Puntatore a dato costante int * const ptr; // Puntatore costante const int * const ptr; // Puntatore costante a dato costante

4.9 34. Array di Puntatori

Utile per gestire stringhe di lunghezza variabile:
[] const char *semi[4] = {"Cuori", "Quadri", "Picche", "Fiori"};
Vantaggi: risparmio di memoria rispetto alle matrici bidimensionali a dimensione fissa.

4.10 35. Rischi e Attenzioni

4.10.1 Aliasing (effetti collaterali):

- Quando due puntatori puntano allo stesso oggetto
- Modifiche tramite un puntatore influenzano l'altro
- Da evitare quando possibile

4.10.2 Precedenza degli operatori:

- `*` e `++` hanno stessa priorità e associano a destra
- Usare parentesi per chiarezza: `(*p)++` vs `*p++`

4.11 36. Operatore sizeof

Restituisce il numero di byte occupati da una variabile o tipo:
[] int totale, a[5]; sizeof(totale) // 4 byte (tipicamente) sizeof(a[2]) // 4 byte sizeof(a) // 20 byte

4.12 Sommario delle Operazioni sui Puntatori

1. **Assegnazione di indirizzo**: `P = &x;`
2. **Dereferenziazione**: `*P = x;`
3. **Assegnazione tra puntatori**: `P = Q;`
4. **Inizializzazione**: `P = NULL;`
5. **Aritmetica**: `P++`, `P--`, `P+i`, `P-i`
6. **Confronto**: tra puntatori dello stesso tipo

I puntatori sono una caratteristica distintiva del C che lo rende un linguaggio di basso livello, consentendo manipolazione diretta della memoria, ma richiedono attenzione per evitare errori comuni.

5 Funzioni in C - Riassunto Teorico

5.1 37. Introduzione ai Sottoprogrammi

5.1.1 Motivazioni

- **Riutilizzo del codice**: evitare blocchi di istruzioni ripetute
- **Modularità**: separare logiche diverse (es. calcolo somma array, MCD)
- **Astrazione**: nascondere l'implementazione dietro un'interfaccia
- **Manutenibilità**: modifiche localizzate in un solo punto

5.1.2 Tipi di Sottoprogrammi in C

1. **Funzioni**: restituiscono un valore al chiamante
2. **Procedure**: svolgono un'azione ma non restituiscono valore (`void`)

5.2 38. Struttura delle Funzioni

5.2.1 Sintassi Generale

[] tipo_ritorno nome_funzione(lista_parametri_formali) { // Parte dichiarativa locale dichiarazioni_variabili_locali;
// Parte esecutiva istruzioni;
return espressione; // opzionale per void }

5.2.2 Componenti

- **Testata (Header)**: contiene tipo di ritorno, nome e parametri formali
- **Corpo (Body)**:
  - Parte dichiarativa locale: variabili necessarie all'esecuzione
  - Parte esecutiva: istruzioni che implementano l'algoritmo

5.3 39. Parametri e Invocazione

5.3.1 Parametri Formali vs Effettivi

- **Parametri formali**: definiti nella testata della funzione
- **Parametri effettivi**: valori passati durante l'invocazione
- All'invocazione, i parametri formali vengono inizializzati con i valori effettivi

5.3.2 Invocazione

[] risultato = nome_funzione(parametri_effettivi);

- Sintatticamente è un'espressione
- Viene valutata e sostituita dal valore restituito

5.4 40. Controllo del Flusso

5.4.1 Istruzione return

- `return;` - termina senza restituire valore (procedure)
- `return espressione;` - restituisce il valore dell'espressione
- Possono esistere multiple return in una funzione
- Se manca return, il risultato è indefinito

5.4.2 Funzione main

- Ha tipo di ritorno `int`
- `return 0` indica successo
- Valori diversi da 0 indicano errori (dipende dal sistema)

5.5 41. Prototipi e Dichiarazioni

5.5.1 Differenza Definizione vs Dichiarazione

- **Definizione**: include testata + corpo
- **Dichiarazione (Prototipo)**: solo testata, termina con **;**

5.5.2 Regola Fondamentale

Una funzione deve essere **definita o dichiarata** prima dell'uso per permettere al compilatore di verificare la correttezza.

5.6 42. Modello di Esecuzione

5.6.1 Macchine Dedicate (Concettuale)

- Ogni invocazione crea una "macchina virtuale" dedicata
- Ogni macchina ha il proprio **ambiente** (memoria locale)
- Gli ambienti sono indipendenti tra loro

5.6.2 Implementazione Reale: Stack

- **Pila LIFO** (Last In First Out)
- Ad ogni invocazione: nuovo ambiente allocato in cima
- Al termine: ambiente rimosso dalla pila
- Gestisce automaticamente invocazioni annidate

5.6.3 Ambiente di Esecuzione

Contiene:

- Variabili locali
- Parametri formali (trattati come variabili locali inizializzate)
- Valore di ritorno

5.7 43. Passaggio dei Parametri

5.7.1 Passaggio per Valore (Default in C)

- Viene creata una **copia** del valore nell'ambiente della funzione
- Modifiche ai parametri non influenzano le variabili originali
- Efficiente per tipi semplici, costoso per strutture grandi

5.7.2 Passaggio per Riferimento

- Implementato usando **puntatori**
- Si passa l'indirizzo della variabile con `&variabile`
- Si accede al contenuto con `*puntatore`
- Permette di modificare la variabile originale

[] // Passaggio per valore void func1(int x) { x = 10; } // non modifica l'originale
// Passaggio per riferimento void func2(int *x) { *x = 10; } // modifica l'originale // Invocazione: func2(&variabile);

5.8 44. Strutture Dati come Parametri

5.8.1 Struct

- **Per valore**: l'intera struttura viene copiata (inclusi eventuali array)
- **Per riferimento**: si passa il puntatore alla struct
- Possono essere usate come valore di ritorno

5.8.2 Array

- **Non possono essere restituiti** direttamente da una funzione
- Si possono restituire **puntatori** ad array
- Gli array vengono sempre passati "per riferimento" (implicitamente come puntatori)

5.9 45. Struttura Completa di un Programma C

[] // 1. Direttive precompilatore #include <stdio.h> #define COSTANTE valore
// 2. Dichiarazioni globali typedef struct {...} TipoPersonalizzato; int variabile_globale;
// 3. Prototipi funzioni int funzione1(int param); void procedura1(int *param);
// 4. Definizioni funzioni int funzione1(int param) { // implementazione return risultato; }
void procedura1(int *param) { // implementazione }
// 5. Funzione principale int main() { // corpo del programma principale return 0; }

5.10 46. Vantaggi del Passaggio per Valore

- **Sicurezza**: impossibile modificare accidentalmente variabili del chiamante
- **Riduzione variabili locali**: i parametri possono essere modificati liberamente
- **Semplicità**: non richiede gestione esplicita di puntatori

5.11 Note Teoriche Fondamentali

5.11.1 Astrazione e Incapsulamento

- Le funzioni forniscono **astrazione procedurale**
- Separano il "cosa" (interfaccia) dal "come" (implementazione)
- Permettono **riutilizzo** e **composizione** di codice

5.11.2 Controllo Sintattico

- Il compilatore verifica:
  - Corrispondenza tipi tra parametri formali/effettivi
  - Esistenza della funzione prima dell'uso
  - Coerenza del tipo di ritorno

5.11.3 Gestione Memoria

- **Automatica** per variabili locali e parametri
- **Dinamica** attraverso lo stack di esecuzione
- **Efficiente** per chiamate annidate e ricorsione

5.12 11. Procedure vs Funzioni

5.12.1 Trasformazione Funzione → Procedura

- Qualsiasi funzione può essere trasformata in procedura
- Il risultato diventa un **parametro aggiuntivo** passato per riferimento

[] // Funzione int f(int p1) { return risultato; } y = f(x);
// Procedura equivalente void f(int p1, int *p2) { *p2 = risultato; } f(x, &y);

5.13 12. Uso di `const` con i Puntatori12. Uso di const con i Puntatori

5.13.1 Principio del Privilegio Minimo

Dare accesso ai dati **sufficiente** per il compito ma **null'altro**.

5.13.2 Quattro Modalità di Passaggio Puntatori

1. **Puntatore non costante a dati non costanti**
   - Massimo livello di accesso
   - Sia puntatore che dati modificabili

   [] void func(int *ptr);

2. **Puntatore non costante a dati costanti**
   - Dati non modificabili, puntatore modificabile

   [] void func(const int *ptr);

3. **Puntatore costante a dati non costanti**
   - Puntatore non modificabile, dati modificabili

   [] void func(int * const ptr);

4. **Puntatore costante a dati costanti**
   - Né puntatore né dati modificabili

   [] void func(const int * const ptr);

5.14 13. Array come Parametri

5.14.1 Caratteristiche Speciali

- Gli array sono **sempre passati per riferimento**
- Viene passato l'**indirizzo del primo elemento**
- Gli elementi **non vengono copiati**
- La dimensione non è richiesta (se presente, viene ignorata)

[] void ModificaArray(int b[], int size); // Equivalente a: void ModificaArray(int *b, int size);
// Invocazione int TemperatureOrarie[24]; ModificaArray(TemperatureOrarie, 24);

5.14.2 Accesso agli Elementi

- Normale sintassi array: `a[i]`
- Equivalente a dereferenziazione: `*(a+i)`

5.15 14. Passaggio Parametri: Riepilogo Teorico

5.15.1 Verità Fondamentale

In C il passaggio è SEMPRE per valore

- Per tipi semplici: si copia il valore
- Per il "passaggio per riferimento": si copia l'indirizzo (valore del puntatore)

5.15.2 Pro e Contro

Per Valore Vantaggi:

- Sicurezza: nessuna modifica accidentale
- Indipendenza tra chiamante e chiamato

Svantaggi:

- Inefficiente per dati voluminosi
- Impossibile restituire multiple modifiche

Per Indirizzo (Riferimento) Vantaggi:

- Efficiente: copia solo l'indirizzo
- Permette modifiche multiple
- Necessario per array

Svantaggi:

- Possibili effetti collaterali
- Maggiore complessità

5.16 15. Ambienti e Visibilità

5.16.1 Definizione di Ambiente

Un **ambiente** è definito da:

1. **Programma intero** (globale)
2. **Singola funzione** (locale)
3. **Blocco** {} (locale al blocco)

5.16.2 Regole di Visibilità

Ambiente Globale

- Variabili/funzioni dichiarate fuori da tutte le funzioni
- **Visibili da tutto il programma**
- Includono identificatori predefiniti del linguaggio

Ambiente Locale (Funzione)

- Variabili dichiarate nella funzione
- **Visibili solo nel corpo della funzione** e suoi blocchi interni

Ambiente Locale (Blocco)

- Variabili dichiarate in un blocco {}
- **Visibili nel blocco** e in blocchi annidati

Mascheramento (Shadowing)

- Dichiarazione in blocco interno **maschera** identica dichiarazione esterna
- La variabile esterna rimane inaccessibile fino all'uscita dal blocco

[] int x = 10; // globale void func() { int x = 20; // maschera la globale { int x = 30; // maschera la locale // qui x vale 30 } // qui x vale 20 } // qui x vale 10

5.17 16. Ciclo di Vita delle Variabili

5.17.1 Variabili Statiche

- **Allocate una volta** all'inizio del programma
- **Distrutte al termine** del programma
- **Globali** sono sempre statiche
- **Locali dichiarate** `static` mantengono valore tra invocazioni

5.17.2 Variabili Automatiche (Dinamiche)

- **Create** all'ingresso nell'ambito di visibilità
- **Distrutte** all'uscita dall'ambito
- Include: parametri formali, variabili locali di funzioni e blocchi
- **Allocate dinamicamente** sullo stack

5.17.3 Implicazioni

- Variabili automatiche **non conservano** valori tra invocazioni
- Ogni invocazione ha il **proprio ambiente separato**
- Gestione automatica della memoria

5.18 17. Effetti Collaterali

5.18.1 Definizione

Effetto collaterale: modifiche che oltrepassano i confini dell'ambiente locale della funzione

5.18.2 Cause Principali

1. **Accesso a variabili globali**
2. **Modifiche tramite puntatori**
3. **I/O** (printf, scanf, ecc.)

5.18.3 Problemi

- **Dipendenze nascoste** tra funzioni
- **Difficoltà nel debug**
- **Codice meno riutilizzabile**
- **Comportamento imprevedibile**

5.18.4 Buone Pratiche

- **Minimizzare** l'uso di variabili globali
- **Preferire** passaggio parametri esplicito
- **Documentare** eventuali effetti collaterali

5.19 18. Standard Library del C

5.19.1 Organizzazione

- **Header files** per diverse categorie di funzioni
- **Funzioni predefinite** per operazioni comuni

5.19.2 Principali Librerie

[] #include <stdio.h> // Input/Output #include <string.h> // Gestione stringhe #include <math.h> // Funzioni matematiche #include <stdlib.h> // Utilità generali #include <ctype.h> // Classificazione caratteri

5.19.3 Vantaggi

- **Riutilizzo** di codice testato
- **Standardizzazione** tra implementazioni
- **Efficienza** (implementazioni ottimizzate)